Secrets of the
JavaScript
Ninja

SECOND EDITION

John Resig
Bear Bibeault
Josip Maras

MANNING

ES6 cheat sheet

Template literals embed expressions into strings: `` `${ninja}` ``.

Block-scoped variables:

- Use the new `let` keyword to create block-level variables: `let ninja = "Yoshi"`.
- Use the new `const` keyword to create block-level constant variables whose value can't be reassigned to a completely new value: `const ninja = "Yoshi"`.

Function parameters:

- *Rest parameters* create an array from arguments that weren't matched to parameters:

```
function multiMax(first, ...remaining){ /*...*/}multiMax(2, 3, 4, 5); //first: 2;
    remaining: [3, 4, 5]
```

- *Default parameters* specify default parameter values that are used if no value is supplied during invocation:

```
function do(ninja, action = "skulk"){ return ninja + " " + action;}
do("Fuma"); //"Fuma skulk"
```

Spread operators expand an expression where multiple items are needed: `[...items, 3, 4, 5]`.

Arrow functions let us create functions with less syntactic clutter. They don't have their own `this` parameter. Instead, they inherit it from the context in which they were created:

```
const values = [0, 3, 2, 5, 7, 4, 8, 1];
values.sort((v1,v2) => v1  - v2); /*OR*/ values.sort((v1,v2) => { return v1  - v2;});
value.forEach(value => console.log(value));
```

Generators generate sequences of values on a per-request basis. Once a value is generated, the generator suspends its execution without blocking. Use `yield` to generate values:

```
function *IdGenerator(){
  let id = 0;
  while(true){ yield ++id; }
}
```

Promises are placeholders for the result of a computation. A promise is a guarantee that eventually we'll know the result of some computation. The promise can either succeed or fail, and once it has done so, there will be no more changes:

- Create a new promise with `new Promise((resolve, reject) => {});`.
- Call the `resolve` function to explicitly resolve a promise. Call the `reject` function to explicitly reject a promise. A promise is implicitly rejected if an error occurs.
- The promise object has a `then` method that returns a promise and takes in two callbacks, a success callback and a failure callback:

```
myPromise.then(val => console.log("Success"), err => console.log("Error"));
```

- Chain in a `catch` method to catch promise failures: `myPromise.catch(e => alert(e));`.

(continued on inside back cover)

Praise for the First Edition

Finally, from a master, a book that delivers what an aspiring JavaScript developer requires to learn the art of crafting effective cross-browser JavaScript.

—Glenn Stokol, Senior Principal Curriculum Developer,
Oracle Corporation

Consistent with the jQuery motto, "Write less, do more."

—André Roberge, Université Saint-Anne

Interesting and original techniques.

—Scott Sauyet, Four Winds Software

Read this book, and you'll no longer blindly plug in a snippet of code and marvel at how it works—you'll understand "why" it works.

—Joe Litton, Collaborative Software Developer, JoeLitton.net

Will help you raise your JavaScript to the realm of the masters.

—Christopher Haupt, greenstack.com

The stuff ninjas need to know.

—Chad Davis, author of *Struts 2 in Action*

Required reading for any JavaScript Master.

—John J. Ryan III, Princigration LLC

This book is a must-have for any serious JS coder. Your knowledge of the language will dramatically expand.

—S., Amazon reader

Secrets of the JavaScript Ninja, Second Edition

JOHN RESIG
BEAR BIBEAULT
and JOSIP MARAS

MANNING

Shelter Island

contents

v

author's introduction

It's incredible to think of how much the world of JavaScript has changed since I first started writing *Secrets of the JavaScript Ninja* back in 2008. The world in which we write JavaScript now, while still being largely centered around the browser, is nearly unrecognizable.

The popularity of JavaScript as a full-featured, cross-platform language has exploded. Node.js is a formidable platform against which countless production applications are developed. Developers are actually writing applications in one language—JavaScript—that are capable of running in a browser, on a server, and even in a native app on a mobile device.

It's more important now, than ever before, that a developer's knowledge of the JavaScript language be at its absolute peak. Having a fundamental understanding of the language and the ways in which it can be best written will allow you to create applications that can work on virtually any platform, which is a claim that few other languages can legitimately boast.

Unlike previous eras in the growth of JavaScript, there hasn't been equal growth in platform incompatibilities. It used to be that you would salivate over the thought of using the most basic new browser features and yet be stymied by outdated browsers that had far too much market share. We've entered a harmonious time in which most users are on rapidly updated browsers that compete to be the most standards-compliant platform around. Browser vendors even go out of their way to develop features specifically targeted at developers, hoping to make their lives easier.

The tools that we have now, provided by browsers and the open source community, are light years ahead of old practices. We have a plethora of testing frameworks to choose from, the ability to do continuous integration testing, generate code-coverage reports, performance-test on real mobile devices around the globe, and even automatically load up virtual browsers on any platform to test from.

The first edition of the book benefited tremendously from the development insight that Bear Bibeault provided. This edition has received substantial help from Josip Maras to explore the concepts behind ECMAScript 6 and 7, dive into testing best practices, and understand the techniques employed by popular JavaScript frameworks.

All of this is a long way of saying: how we write JavaScript has changed substantially. Fortunately, this book is here to help you keep on top of the current best practices. Not only that, but it'll help you improve how you think about your development practices as a whole to ensure that you'll be ready for writing JavaScript well into the future.

JOHN RESIG

acknowledgments

The number of people involved in writing a book would surprise most people. It took a collaborative effort on the part of many contributors with a variety of talents to bring the volume you are holding (or ebook that you are reading onscreen) to fruition.

The staff at Manning worked tirelessly with us to make sure this book attained the level of quality we hoped for, and we thank them for their efforts. Without them, this book would not have been possible. The "end credits" for this book include not only our publisher, Marjan Bace, and editor Dan Maharry, but also the following contributors: Ozren Harlovic, Gregor Zurowski, Kevin Sullivan, Janet Vail, Tiffany Taylor, Sharon Wilkey, Alyson Brener, and Gordan Salinovic.

Enough cannot be said to thank our peer reviewers who helped mold the final form of the book, from catching simple typos to correcting errors in terminology and code, and helping to organize the chapters in the book. Each pass through a review cycle ended up vastly improving the final product. For taking the time to review the book, we'd like to thank Jacob Andresen, Tidjani Belmansour, Francesco Bianchi, Matthew Halverson, Becky Huett, Daniel Lamb, Michael Lund, Kariem Ali Elkoush, Elyse Kolker Gordon, Christopher Haupt, Mike Hatfield, Gerd Klevesaat, Alex Lucas, Arun Noronha, Adam Scheller, David Starkey, and Gregor Zurowski.

Special thanks go to Mathias Bynens and Jon Borgman, who served as the book's technical proofreaders. In addition to checking each and every sample of example code in multiple environments, they also offered invaluable contributions to the technical accuracy of the text, located information that was originally missing, and kept abreast of the rapid changes to JavaScript and HTML5 support in the browsers.

John Resig

I would like to thank my parents for their constant support and encouragement over the years. They provided me with the resources and tools that I needed to spark my initial interest in programming—and they have been encouraging me ever since.

Bear Bibeault

The cast of characters I'd like to thank for this seventh go-around has a long list of "usual suspects," including, once again, the membership and staff at coderanch.com (formerly JavaRanch). Without my involvement in CodeRanch, I'd never have gotten the opportunity to begin writing in the first place, and so I sincerely thank Paul Wheaton and Kathy Sierra for starting the whole thing, as well as fellow staffers who gave me encouragement and support, including (but certainly not limited to) Eric Pascarello, Ernest Friedman-Hill, Andrew Monkhouse, Jeanne Boyarsky, Bert Bates, and Max Habibi.

My husband, Jay, and my dogs, Little Bear and Cozmo, get the usual warm thanks for putting up with the shadowy presence who shared their home and rarely looked up from his keyboard except to curse Word, the browsers, my fat-fingered lack of typing skills, or anything else that attracted my ire while I was working on this project.

And finally, I'd like to thank my coauthors, John Resig and Josip Maras, without whom this project would not exist.

Josip Maras

The biggest thanks go to my wife, Josipa, for putting up with all the hours that went into writing this book.

I would also like to thank Maja Stula, Darko Stipanicev, Ivica Crnkovic, Jan Carlson, and Bert Bates: all of them for guidance and useful advice, and some of them for being lenient on my "day job" assignments as book deadlines were approaching.

Finally, I would like to thank the rest of my family—Jere, two Marijas, Vitomir, and Zdenka—for always being there for me.

about this book

JavaScript is important. That wasn't always so, but it's true now. JavaScript has become one of the most important and most widely used programming languages today.

Web applications are expected to give users a rich user interface experience, and without JavaScript, you might as well just be showing pictures of kittens. More than ever, web developers need to have a sound grasp of the language that brings life to web applications.

And like orange juice and breakfast, JavaScript isn't just for browsers anymore. The language has long ago knocked down the walls of the browser and is being used on the server thanks to Node.js, on desktop devices and mobiles through platforms such as Apache Cordova, and even on embedded devices with Espruino and Tessel.

Although this book is primarily focused on JavaScript executed in the browser, the fundamentals of the language presented in this book are applicable across the board. Truly understanding the concepts and learning various tips and tricks will make you a better all-around JavaScript developer.

With more and more developers using JavaScript in an increasingly JavaScript world, it's more important than ever to grasp its fundamentals so you can become an expert ninja of the language.

Audience

If you aren't at all familiar with JavaScript, this probably shouldn't be your first book. Even if it is, don't worry too much; we try to present fundamental JavaScript concepts in a way that should be understandable even for relative beginners. But, to be honest,

this book will probably best fit people who already know some JavaScript and who wish to deepen their understanding of JavaScript as a language, as well as the browser as the environment in which JavaScript code is executed.

Roadmap

This book is organized to take you from an apprentice to a ninja in four parts.

Part 1 introduces the topic and sets the stage so that you can easily progress through the rest of the book:

- Chapter 1 introduces JavaScript the language and its most important features, while suggesting current best practices we should follow when developing applications, including testing and performance analysis.
- Because our exploration of JavaScript is made in the context of browsers, in chapter 2 we'll set the stage by introducing the lifecycle of client-side web applications. That will help us understand JavaScript's role in the process of developing web applications.

Part 2 focuses on one of the pillars of JavaScript: functions. We'll study why functions are so important in JavaScript, the different kinds of functions, as well as the nitty-gritty details of defining and invoking functions. We'll put a special focus on a new type of function—generator functions—which are especially helpful when dealing with asynchronous code:

- Chapter 3 begins our foray into the fundamentals of the language, starting, perhaps to your surprise, with a thorough examination of the *function* as defined by JavaScript. Although you may have expected the *object* to be the target of our first focus, a solid understanding of the function, and JavaScript as a functional language, begins our transformation from run-of-the-mill JavaScript coders to JavaScript ninjas!
- We continue this functional thread in chapter 4, by exploring the exact mechanism of invoking functions, as well as the ins and outs of implicit function parameters.
- Not being done with functions quite yet, in chapter 5 we take our discussion to the next level by studying two closely related concepts: *scopes* and *closures.* A key concept in functional programming, closures allow us to exert fine-grained control over the scope of objects that we declare and create in our programs. The control of these scopes is the key factor in writing code worthy of a ninja. Even if you stop reading after this chapter (but we hope you don't), you'll be a far better JavaScript developer than when you started.
- We conclude our exploration of functions in chapter 6, by taking a look at a completely new type of function (*generator functions*) and a new type of object (*promises)* that help us deal with asynchronous values. We'll also show you how to combine generators and promises to achieve elegance when dealing with asynchronous code.

Part 3 deals with the second pillar of JavaScript: objects. We'll thoroughly explore object orientation in JavaScript, and we'll study how to guard access to objects and how to deal with collections and regular expressions:

- Objects are finally addressed in chapter 7, where we learn exactly how JavaScript's slightly strange flavor of object orientation works. We'll also introduce a new addition to JavaScript: classes, which, deep under the hood, may not be exactly what you expect.
- We'll continue our exploration of objects in chapter 8, where we'll study different techniques for guarding access to our objects.
- In chapter 9, we'll put a special focus on different types of collections that exist in JavaScript; on arrays, which have been a part of JavaScript since its beginnings; and on maps and sets, which are recent addition to JavaScript.
- Chapter 10 focuses on regular expressions, an often-overlooked feature of the language that can do the work of scores of lines of code when used correctly. We'll learn how to construct and use regular expressions and how to solve some recurring problems elegantly, using regular expressions and the methods that work with them.
- In chapter 11, we'll learn different techniques for organizing our code into modules: smaller, relatively loosely coupled segments that improve the structure and organization of our code.

Finally, part 4 wraps up the book by studying how JavaScript interacts with our web pages and how events are processed by the browser. We'll finish the book by looking at an important topic, cross-browser development:

- Chapter 12 explores how we can dynamically modify our pages through DOM-manipulation APIs, and how we can handle element attributes, properties, and styles, as well as some important performance considerations.
- Chapter 13 discusses the importance of JavaScript's single-threaded execution model and the consequences this model has on the event loop. We'll also learn how timers and intervals work and how we can use them to improve the perceived performance of our web applications.
- Chapter 14 concludes the book by examining the five key development concerns with regard to these cross-browser issues: browser differences, bugs and bug fixes, external code and markup, missing features, and regressions. Strategies such as feature simulation and object detection are discussed at length to help us deal with these cross-browser challenges.

Code conventions

All source code in listings or in the text is in a `fixed-width font like this` to separate it from ordinary text. Method and function names, properties, XML elements, and attributes in the text are also presented in this same font.

In some cases, the original source code has been reformatted to fit on the pages. In general, the original code was written with page-width limitations in mind, but sometimes you may find a slight formatting difference between the code in the book and that provided in the source download. In a few rare cases, where long lines could not be reformatted without changing their meaning, the book listings contain line-continuation markers.

Code annotations accompany many of the listings; these highlight important concepts.

Code downloads

Source code for all the working examples in this book (along with some extras that never made it into the text) is available for download from the book's web page at https://manning.com/books/secrets-of-the-javascript-ninja-second-edition.

The code examples for this book are organized by chapter, with separate folders for each chapter. The layout is ready to be served by a local web server, such as the Apache HTTP Server. Unzip the downloaded code into a folder of your choice, and make that folder the document root of the application.

With a few exceptions, most of the examples don't require the presence of a web server and can be loaded directly into a browser for execution, if you so desire.
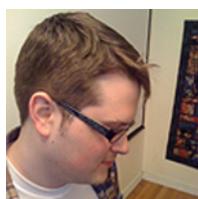
Author Online

The authors and Manning Publications invite you to the book's forum, run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the authors and other users. To access and subscribe to the forum, point your browser to https://manning.com/books/secrets-of-the-javascript-ninja-second-edition and click the Author Online link. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct in the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It's not a commitment to any specific amount of participation on the part of the authors, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions, lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the authors

John Resig is a staff engineer at Khan Academy and the creator of the jQuery JavaScript library. In addition to the first edition of *Secrets of the JavaScript Ninja*, he's also the author of the book *Pro JavaScript Techniques.*

 John has developed a comprehensive Japanese woodblock print database and image search engine: Ukiyo-e.org. He's a board member of the Japanese Art Society of America and is a Visiting Researcher at Ritsumeikan University working on the study of Ukiyo-e.

John is located in Brooklyn, NY.

Bear Bibeault has been writing software for well over three decades, starting with a Tic-Tac-Toe program written on a Control Data Cyber supercomputer via a 100-baud teletype. Having two degrees in electrical engineering, Bear should be designing antennas or something like that, but since his first job with Digital Equipment Corporation, he has always been much more fascinated with programming.

 Bear has also served stints with companies such as Dragon Systems, Works.com, Spredfast, Logitech, Caringo, and more than a handful of others. Bear even served in the U.S. military, leading and training a platoon of anti-tank infantry soldiers—skills that come in handy during scrum meetings. "That's *Sergeant* Bear to you, trainee!"

Bear is currently a senior front-end developer for a leading provider of object storage software that provides massive storage scalability and content protection.

In addition to the first edition of this book, Bear is also the author of a number of other Manning books, including *jQuery in Action* (first, second, and third editions), *Ajax in Practice*, and *Prototype and Scriptaculous in Action*; and he has been a technical reviewer for many of the web-focused "Head First" books by O'Reilly Publishing, such as *Head First Ajax*, *Head Rush Ajax*, and *Head First Servlets and JSP.*

In addition to his day job, Bear also writes books (duh!), runs a small business that creates web applications and offers other media services (but not wedding videography—never, ever wedding videography), and helps out at CodeRanch.com as a "marshal" (uber moderator).

 When not planted in front of a computer, Bear likes to cook *big* food (which accounts for his jeans size), dabble in photography and video, ride his Yamaha V-Star, and wear tropical print shirts.

He works and resides in Austin, Texas, a city he loves, except for the completely insane traffic and drivers.

Josip Maras is a post-doctoral researcher in the faculty of electrical engineering, mechanical engineering, and naval architecture, University of Split, Croatia. He has a PhD in software engineering, with the thesis "Automating Reuse in Web Application Development," which among other things included implementing a JavaScript interpreter in JavaScript. During his research, he has published more than a dozen scientific conference and journal papers, mostly dealing with program analysis of client-side web applications.

When not doing research, Josip spends his time teaching web development, systems analysis and design, and Windows development (a couple hundred students over the last six years). He also owns a small software development business.

In his spare time, Josip enjoys reading, long runs, and, if the weather allows, swimming in the Adriatic.

about the cover illustration

The figure on the cover of *Secrets of the JavaScript Ninja, Second Edition* is captioned "Noh Actor, Samurai," from a woodblock print by an unknown Japanese artist of the mid-19th century. Derived from the Japanese word for *talent* or *skill*, Noh is a form of classical Japanese musical drama that has been performed since the 14th century. Many characters are masked, with men playing male and female roles. The samurai, a hero figure in Japan for hundreds of years, was often featured in the performances, and in this print the artist renders with great skill the beauty of the costume and the ferocity of the samurai.

Samurai and ninjas were both warriors excelling in the Japanese art of war, known for their bravery and cunning. Samurai were elite soldiers, well-educated men who knew how to read and write as well as fight, and they were bound by a strict code of honor called Bushido (The Way of the Warrior), which was passed down orally from generation to generation, starting in the 10th century. Recruited from the aristocracy and upper classes, analogous to European knights, samurai went into battle in large formations, wearing elaborate armor and colorful dress meant to impress and intimidate. Ninjas were chosen for their martial arts skills rather than their social standing or education. Dressed in black and with their faces covered, they were sent on missions alone or in small groups to attack the enemy with subterfuge and stealth, using any tactics to assure success; their only code was one of secrecy.

The cover illustration is from a set of three Japanese prints owned for many years by a Manning editor, and when we were looking for a ninja for the cover of this book, the

striking samurai print came to our attention and was selected for its intricate details, vibrant colors, and vivid depiction of a ferocious warrior ready to strike—and win.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on two-hundred-year-old illustrations that depict the rich diversity of traditional costumes from around the world, brought back to life by prints such as this one.

Part 1

Warming up

This part of the book will set the stage for your JavaScript ninja training. In chapter 1, we'll look at the current state of JavaScript and explore some of the environments in which JavaScript code can be executed. We'll put a special focus on the environment where it all began—*the browser*—and we'll discuss some of the best practices when developing JavaScript applications.

Because our exploration of JavaScript will be done in the context of browsers, in chapter 2 we'll teach you about the lifecycle of client-side web applications and how executing JavaScript code fits into this lifecycle.

When you're finished with this part of the book, you'll be ready to embark on your training as a JavaScript ninja!

JavaScript is everywhere 1

This chapter covers
- The core language features of JavaScript
- The core items of a JavaScript engine
- Three best practices in JavaScript development

Let's talk about Bob. After spending a few years learning how to create desktop applications in C++, he graduated as a software developer in the early 2000s and then went out into the wide world. At that point, the web had just hit its stride, and everybody wanted to be the next Amazon. So the first thing he did was learn web development.

He learned some PHP so that he could dynamically generate web pages, which he usually sprinkled with JavaScript in order to achieve complex functionality such as form validation and even dynamic on-page clocks! Fast-forward a couple of years, and smartphones had become a thing, so anticipating a large new market opening up, Bob went ahead and learned Objective-C and Java to develop mobile apps that run on iOS and Android.

Over the years, Bob has created many successful applications that all have to be maintained and extended. Unfortunately, jumping daily between all these different programming languages and application frameworks has really started to wear down poor Bob.

Now let's talk about Ann. Two years ago, Ann graduated with a degree in software development, specializing in web- and cloud-based applications. She has created a few medium-sized web applications based on modern Model–view–controller (MVC) frameworks, along with accompanying mobile applications that run on iOS and Android. She has created a desktop application that runs on Linux, Windows, and OS X, and has even started building a serverless version of that application entirely based in the cloud. And *everything she has done has been written in JavaScript.*

That's extraordinary! What took Bob 10 years and 5 languages to do, Ann has achieved in 2 years and in *just one language.* Throughout the history of computing, it has been rare for a particular knowledge set to be so easily transferable and useful across so many different domains.

What started as a humble 10-day project back in 1995 is now one of the most widely used programming languages in the world. JavaScript is quite literally *everywhere*, thanks to more-powerful JavaScript engines and the introduction of frameworks such as Node, Apache Cordova, Ionic, and Electron, which have taken the language beyond the humble web page. And, like HTML, the language itself is now getting long overdue upgrades intended to make JavaScript even more suitable for modern application development.

In this book, we're going to make sure you know all you need to know about JavaScript so that, whether you're like Ann or like Bob, you can develop all sorts of applications on a green field or a brown field.

. . . .

Do you know?   **What are Babel and Traceur, and why are they important to today's JavaScript developers?**

What are the core parts of any web browser's JavaScript API used by web applications?

. . . .

1.1   *Understanding the JavaScript language*

As they advance through their careers, many JavaScript coders like Bob and Ann reach the point where they're actively using the vast number of elements that form the language. In many cases, however, those skills may not be taken beyond fundamental levels. Our guess is that this is often because JavaScript, using a C-like syntax, bears a surface resemblance to other widespread C-like languages (such as C# and Java), and thus leaves the impression of familiarity.

People often feel that if they know C# or Java, they already have a pretty solid understanding of how JavaScript works. But it's a trap! When compared to other mainstream languages, JavaScript is much more *functionally* oriented. Some JavaScript concepts differ fundamentally from those of most other languages.

These differences include the following:

- *Functions are first-class objects*—In JavaScript, functions coexist with, and can be treated like, any other JavaScript object. They can be created through literals, referenced by variables, passed around as function arguments, and even returned as function return values. We devote much of chapter 3 to exploring some of the wonderful benefits that functions as first-class objects bring to our JavaScript code.

- *Function closures*—The concept of function closures is generally poorly understood, but at the same time it fundamentally and irrevocably exemplifies the importance of functions to JavaScript. For now, it's enough to know that a function is *a closure when it actively maintains ("closes over") the external variables used in its body.* Don't worry for now if you don't see the many benefits of closures; we'll make sure all is crystal clear in chapter 5. In addition to closures, we'll thoroughly explore the many aspects of functions themselves in chapters 3 and 4, as well as identifier scopes in chapter 5.

- *Scopes*—Until recently, JavaScript didn't have block-level variables (as in other C-like languages); instead, we had to rely only on global variables and function-level variables.

- *Prototype-based object orientation*—Unlike other mainstream programming languages (such as C#, Java, and Ruby), which use class-based object orientation, JavaScript uses prototypes. Often, when developers come to JavaScript from class-based languages (such as Java), they try to use JavaScript as if it were Java, essentially writing Java's class-based code using the syntax of JavaScript. Then, for some reason, they're surprised when the results differ from what they expect. This is why we'll go deep into prototypes, how prototype-based object-orientation works, and how it's implemented in JavaScript.

JavaScript consists of a close relationship between objects and prototypes, and functions and closures. Understanding the strong relationships between these concepts can vastly improve your JavaScript programming ability, giving you a strong foundation for any type of application development, regardless of whether your JavaScript code will be executed in a web page, in a desktop app, in a mobile app, or on the server.

In addition to these fundamental concepts, other JavaScript features can help you write more elegant and more efficient code. Some of these are features that seasoned developers like Bob will recognize from other languages, such as Java and C++. In particular, we focus on the following:

- *Generators*, which are functions that can generate multiple values on a per-request basis and can suspend their execution between requests

- *Promises*, which give us better control over asynchronous code

- *Proxies,* which allow us to control access to certain objects

- *Advanced array methods,* which make array-handling code much more elegant

- *Maps*, which we can use to create dictionary collections; and *sets*, which allow us to deal with collections of unique items
- *Regular expressions*, which let us simplify what would otherwise be complicated pieces of code
- *Modules*, which we can use to break code into smaller, relatively self-contained pieces that make projects more manageable

Having a deep understanding of the fundamentals and learning how to use advanced language features to their best advantage can elevate your code to higher levels. Honing your skills to tie these concepts and features together will give you a level of understanding that puts the creation of any type of JavaScript application within your reach.

1.1.1 How will JavaScript evolve?

The ECMAScript committee, in charge of standardizing the language, has just finished the ES7/ES2016 version of JavaScript. The ES7 version is a relatively small upgrade to JavaScript (at least, when compared to ES6), because the committee's goal going forward is to focus on smaller, yearly incremental changes to the language.

In this book we thoroughly explore ES6 but also focus on ES7 features, such as the new async function, which will help you deal with asynchronous code (discussed in chapter 6).

> **NOTE** When we cover features of JavaScript defined in ES6/ES2015 or ES7/ES2016, you'll see these icons alongside a link to information about whether they're supported by your browser.

Yearly incremental updates to the language specification are excellent news, but this doesn't mean that web developers will instantly have access to the new features after the specification has been released. JavaScript code has to be executed by a JavaScript engine, so we're often left waiting impatiently for updates to our favorite JavaScript engines that incorporate these new and exciting features.

Unfortunately, although the JavaScript engine developers are trying to keep up and are doing better all the time, there's always a chance that you'll run into features that you are dying to use but that are yet to be supported.

Luckily, you can keep up with the state of feature support in the various browsers via the lists at https://kangax.github.io/compat-table/es6/, http://kangax.github.io/compat-table/es2016plus/, and https://kangax.github.io/compat-table/esnext/.

1.1.2 Transpilers give us access to tomorrow's JavaScript today

Because of the rapid release cycles of browsers, we usually don't have to wait long for a JavaScript feature to be supported. But what happens if we want to take advantage of

all the benefits of the newest JavaScript features but are taken hostage by a harsh reality: The users of our web applications may still be using older browsers?

One answer to this problem is to use *transpilers* ("transformation + compiling"), tools that take cutting-edge JavaScript code and transform it into equivalent (or, if that's not possible, similar) code that works properly in most current browsers.

Today's most popular transpilers are Traceur (https://github.com/google/traceur-compiler) and Babel (https://babeljs.io/). Setting them up is easy; just follow one of the tutorials, such as https://github.com/google/traceur-compiler/wiki/Getting-Started or http://babeljs.io/docs/setup/.

In this book, we put a special focus on running JavaScript code in the browser. To effectively use the browser platform, you have to get your hands dirty and study the inner workings of browsers. Let's get started!

1.2 Understanding the browser

These days, JavaScript applications can be executed in many environments. But the environment from which it all began, the environment from which all other environments have taken ideas, and the environment on which we'll focus, is the *browser*. The browser provides various concepts and APIs to thoroughly explore; see figure 1.1.
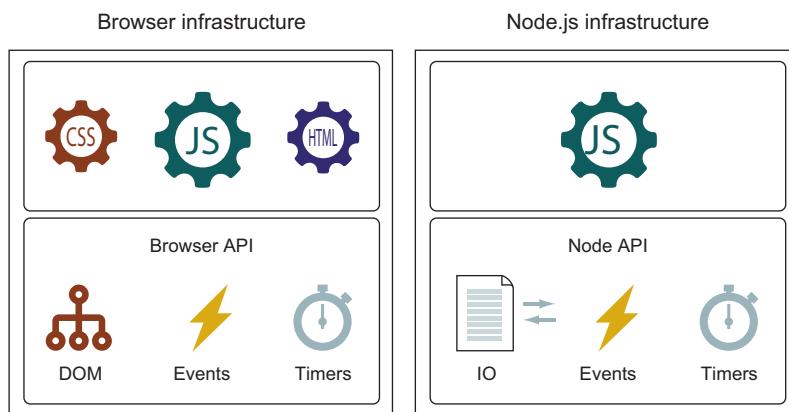
Figure 1.1   **Client-side web applications rely on the infrastructure provided by the browser. We'll particularly focus on the DOM, events, timers, and browser APIs.**

We'll concentrate on the following:

- *The Document Object Model* (DOM)—The DOM is a structured representation of the UI of a client-side web application that is, at least initially, built from the HTML code of a web application. To develop great applications, you need to not only have a deep understanding of the core JavaScript mechanics, but also study how the DOM is constructed (chapter 2) and how to write effective code that manipulates the DOM (chapter 12). This will put the creation of advanced, highly dynamic UIs at your fingertips.

- *Events*—A huge majority of JavaScript applications are *event-driven* applications, meaning that most of the code is executed in the context of a response to a particular event. Examples of events include network events, timers, and user-generated events such as clicks, mouse moves, keyboard presses, and so on. For this reason, we'll thoroughly explore the mechanisms behind events in chapter 13. We'll pay special attention to *timers,* which are frequently a mystery but let us tackle complex coding tasks such as long-running computations and smooth animations.
- *Browser API*—To help us interact with the world, the browser offers an API that allows us to access information about devices, store data locally, or communicate with remote servers. We'll explore some of these APIs throughout the book.

Perfecting your JavaScript programming skills and achieving deep understanding of APIs offered by the browser will take you far. But sooner, rather than later, you'll run face first into *the browsers* and their various issues and inconsistencies. In a perfect world, all browsers would be bug-free and would support web standards in a consistent fashion; unfortunately, we don't live in that world.

The quality of browsers has improved greatly as of late, but they all still have some bugs, missing APIs, and browser-specific quirks that we need to deal with. Developing a comprehensive strategy for tackling these browser issues, and becoming intimately familiar with their differences and quirks, can be almost as important as proficiency in JavaScript itself.

When we're writing browser applications or JavaScript libraries to be used in them, choosing which browsers to support is an important consideration. We'd like to support them all, but limitations on development and testing resources dictate otherwise. For this reason, we'll thoroughly explore strategies for cross-browser development in chapter 14.

Developing effective, cross-browser code can depend significantly on the skill and experience of the developers. This book is intended to boost that skill level, so let's get to it by looking at current best practices.

1.3   Using current best practices

Mastery of the JavaScript language and a grasp of cross-browser coding issues are important parts of becoming an expert web application developer, but they're not the complete picture. To enter the big leagues, you also need to exhibit the traits that scores of previous developers have proven are beneficial to the development of quality code. These traits are known as *best practices,* and in addition to mastery of the language, they include such elements as

- Debugging skills
- Testing
- Performance analysis

It's vitally important to adhere to these practices when coding, and we'll use them throughout the book. Let's examine some of them next.

1.3.1 Debugging

Debugging JavaScript used to mean using `alert` to verify the value of variables. Fortunately, the ability to debug JavaScript code has dramatically improved, in no small part because of the popularity of the Firebug developer extension for Firefox. Similar tools have been developed for all major browsers:

- *Firebug*—The popular developer extension for Firefox that got the ball rolling (http://getfirebug.com/)
- *Chrome DevTools*—Developed by the Chrome team and used in Chrome and Opera
- *Firefox Developer Tools*—A tool included in Firefox, built by the Firefox team
- *F12 Developer Tools*—Included in Internet Explorer and Microsoft Edge
- *WebKit Inspector*—Used by Safari

As you can see, every major browser offers developer tools that we can use to debug our web applications. The days of using JavaScript alerts for debugging are long gone!

All of these tools are based on similar ideas, which were mostly introduced by Firebug, so they offer similar functionality: exploring the DOM, debugging JavaScript, editing CSS styles, tracking network events, and so on. Any of them will do a fine job; use the one offered by your browser of choice, or in the browser in which you're investigating bugs.

In addition, you can use some of them, such as Chrome Dev Tools, to debug other kinds of applications, like Node.js apps. (We'll introduce you to some debugging techniques in appendix B.)

1.3.2 Testing

Throughout this book, we'll apply testing techniques to ensure that the example code operates as intended and to serve as examples of how to test code in general. The primary tool we'll use for testing is an `assert` function, whose purpose is to assert that a premise is either true or false. By specifying assertions, we can check whether the code is behaving as expected.

The general form of this function is as follows:

```
assert(condition, message);
```

The first parameter is a condition that should be true, and the second is a message that will be displayed if it's not.

Consider this, for example:

```
assert(a === 1, "Disaster! a is not 1!");
```

If the value of variable a isn't equal to 1, the assertion fails, and the somewhat overly dramatic message is displayed.

> **NOTE** The assert function isn't a standard feature of the language, so we'll implement it ourselves in appendix B.

1.3.3 Performance analysis

Another important practice is performance analysis. The JavaScript engines have made astounding strides in the performance of JavaScript, but that's no excuse for writing sloppy and inefficient code.

We'll use code such as the following later in this book to collect performance information:

```
console.time("My operation");            ⟵——— Starts the timer

for(var n = 0; n < maxCount; n++){       Performs
  /*perform the operation to be measured*/   the operation
}                                            multiple times

console.timeEnd("My operation");         ⟵——— Stops the timer
```

Here, we bracket the execution of the code to be measured with two calls to the time and timeEnd methods of the built-in console object.

Before the operation begins executing, the call to console.time starts a timer with a name (in this case, My operation). Then we run the code in the for loop a certain number of times (in this case, maxCount times). Because a single operation of the code happens much too quickly to measure reliably, we need to perform the code many times to get a measurable value. Frequently, this count can be in the tens of thousands, or even millions, depending on the nature of the code being measured. A little trial and error lets us choose a reasonable value.

When the operation ends, we call the console.timeEnd method with the same name. This causes the browser to output the time that elapsed since the timer was started.

These best-practice techniques, along with others you'll learn along the way, will greatly enhance your JavaScript development. Developing applications with the restricted resources that a browser provides, coupled with the increasingly complex world of browser capability and compatibility, requires a robust and complete set of skills.

1.4 Boosting skill transferability

When Bob was first learning web development, each browser had its own way of interpreting script and UI styles, preaching that their way was the best way and making every developer grind their teeth in frustration. Fortunately, the browser wars ended with HTML, CSS, the DOM API, and JavaScript all being standardized, and developer focus turning toward effective cross-browser JavaScript applications. Indeed, this focus on

treating websites as applications led to many ideas, tools, and techniques crossing over from desktop applications to web applications. And now, that knowledge and tools transfer has happened again as ideas, tools, and techniques that originated in client-side web development have also permeated other application domains.

Achieving a deep understanding of fundamental JavaScript principles with the knowledge of core APIs can therefore make you a more versatile developer. By using the browsers and Node.js (an environment derived from the browser), you can develop almost any type of application imaginable:

- *Desktop applications*, by using, for example, NW.js (http://nwjs.io/) or Electron (http://electron.atom.io/). These technologies usually wrap the browser so that we can build desktop UIs with standard HTML, CSS, and JavaScript (that way, we can rely on our core JavaScript and browser knowledge), with additional support that makes it possible to interact with the filesystem. We can build truly platform-independent desktop applications that have the same look and feel on Windows, Mac, and Linux.
- *Mobile apps with frameworks*, such as Apache Cordova (https://cordova .apache.org/). Similar to desktop apps built with web technologies, frameworks for mobile apps use a wrapped browser but with additional platform-specific APIs that let us interact with the mobile platform.
- *Server-side applications and applications for embedded devices with Node.js*, an environment derived from the browser that uses many of the same underlying principles as the browser. For example, Node.js executes JavaScript code and relies on events.

Ann doesn't know how lucky she is (although Bob has a pretty good idea). It doesn't matter whether she needs to build a standard desktop application, a mobile application, a server-side application, or even an embedded application—all these types of applications share some of the same underlying principles of standard client-side web applications. By understanding how the core mechanics of JavaScript work, and by understanding the core APIs provided by browsers (such as events, which also have a lot in common with mechanisms provided by Node.js), she can boost her development skills across the board. As can you. In the process, you'll become a more versatile developer and gain the knowledge and understanding to tackle a wide variety of problems. You'll even be able to build your own serverless applications based in the cloud by using JavaScript APIs for services such as AWS Lambda to deploy, maintain, and control your application's cloud components.

1.5 Summary

- Client-side web applications are among the most popular today, and the concepts, tools, and techniques once used only for their development have permeated other application domains. Understanding the foundations of client-side web applications will help you develop applications for a wide variety of domains.

- To improve your development skills, you have to gain a deep understanding of the core mechanics of JavaScript, as well as the infrastructure provided by the browsers.

- This book focuses on core JavaScript mechanisms such as functions, function closures, and prototypes, as well as on new JavaScript features such as generators, promises, proxies, maps, sets, and modules.

- JavaScript can be executed in a large number of environments, but the environment where it all began, and the environment we'll concentrate on, is the browser.

- In addition to JavaScript, we'll explore browser internals such as the DOM (a structured representation of the web page UI) and events, because client-side web applications are event-driven applications.

- We'll do this exploration with best practices in mind: debugging, testing, and performance analysis.

Building
the page at runtime

2

This chapter covers

- Steps in the lifecycle of a web application
- Processing HTML code to produce a web page
- Order of executing JavaScript code
- Achieving interactivity with events
- The event loop

Our exploration of JavaScript is performed in the context of client-side web applications and the browser as the engine that executes JavaScript code. In order to have a strong base from which to continue exploring JavaScript as a language and the browser as a platform, first we have to understand the complete web application lifecycle, and especially how our JavaScript code fits into this lifecycle.

In this chapter, we'll thoroughly explore the lifecycle of client-side web applications from the moment the page is requested, through various interactions performed by the user, all the way until the page is closed down. First we'll explore how the page is built by processing the HTML code. Then we'll focus on the execution of JavaScript code, which adds much-needed dynamicity to our pages. And finally

we'll look into how events are handled in order to develop interactive applications that respond to users' actions.

During this process, we'll explore some fundamental web application concepts such as the DOM (a structured representation of a web page) and the event loop (determines how events are handled by applications). Let's dive in!

. . . .

Do you know?

> **Does the browser always build the page exactly according to the given HTML?**
>
> **How many events can a web application handle at once?**
>
> **Why must browsers use an event queue to process events?**

. . . .

2.1   *The lifecycle overview*

The lifecycle of a typical client-side web application begins with the user typing a URL into the browser's address bar or clicking a link. Let's say we want to look up a term and go to Google's homepage. We type in the URL www.google.com, as shown at upper left in figure 2.1.
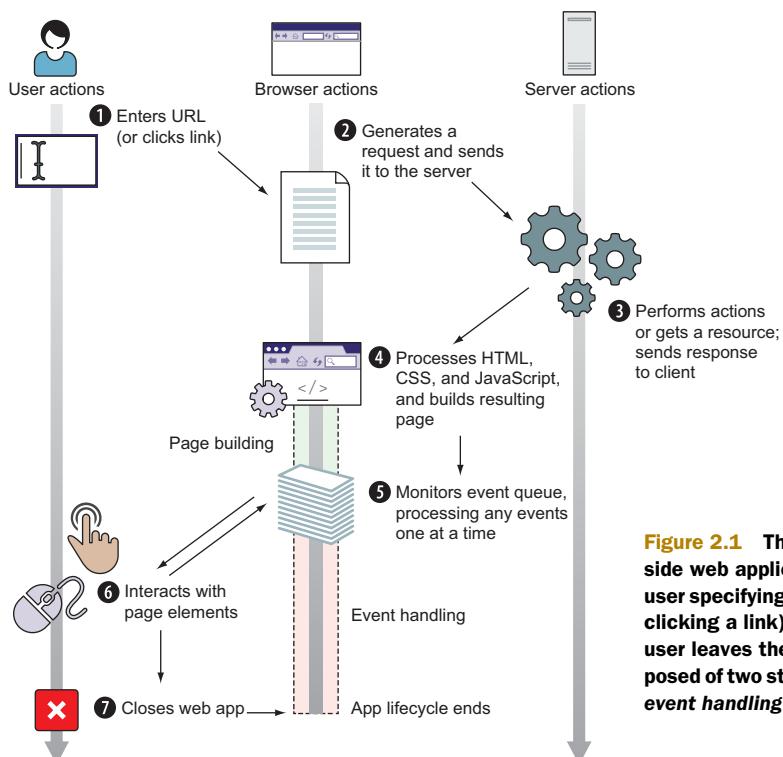
Figure 2.1   **The lifecycle of a client-side web application starts with the user specifying a website address (or clicking a link) and ends when the user leaves the web page. It's composed of two steps: *page building* and *event handling*.**

On behalf of the user, the browser formulates a request that is sent to a server ❷, which processes the request ❸ and formulates a response that is usually composed of HTML, CSS, and JavaScript code. The moment the browser receives this response ❹ is when our client-side web application truly starts coming to life.

Because client-side web applications are Graphical User Interface (GUI) applications, their lifecycle follows similar phases as other GUI applications (think standard desktop applications or mobile applications) and is carried out in the following two steps:

1. *Page building*—Set up the user interface.
2. *Event handling*—Enter a loop ❺ waiting for events to occur ❻, and start invoking event handlers.

The lifecycle of the application ends when the user closes or leaves the web page ❼.

Now let's look at an example web application with a simple UI that reacts to user actions: Every time a user moves a mouse or clicks the page, a message is displayed. We'll use this application throughout the chapter.

Listing 2.1   Small web application with a GUI reacting to events

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Web app lifecycle</title>
    <style>
      #first { color: green;}
      #second { color: red;}
    </style>
  </head>
  <body>
    <ul id="first"></ul>

    <script>
      function addMessage(element, message){
        var messageElement = document.createElement("li");
        messageElement.textContent = message;
        element.appendChild(messageElement);
      }

      var first = document.getElementById("first");
      addMessage(first, "Page loading");
    </script>

    <ul id="second"></ul>

    <script>
      document.body.addEventListener("mousemove", function() {
        var second = document.getElementById("second");
        addMessage(second, "Event: mousemove");
      });

      document.body.addEventListener("click", function(){
```

Defines a function that adds a message to an element

Attaches mousemove event handler to body

Attaches click event handler to body

```
        var second = document.getElementById("second");
        addMessage(second, "Event: click");
      });
    </script>
  </body>
</html>
```

Listing 2.1 first defines two CSS rules, #first and #second, that specify the text color for the elements with the IDs first and second (so that we can easily distinguish between them). We continue by defining a list element with the id first:

```
<ul id="first"></ul>
```

Then we define an addMessage function that, when invoked, creates a new list item element, sets its text content, and appends it to an existing element:

```
function addMessage(element, message){
  var messageElement = document.createElement("li");
  messageElement.textContent = message;
  element.appendChild(messageElement);
}
```

We follow this by using the built-in getElementById method to fetch an element with the ID first from the document, and adding a message to it that notifies us that the page is loading:

```
var first = document.getElementById("first");
addMessage(first, "Page loading");
```

Next we define another list element, now with the attribute ID second:

```
<ul id="second"></ul>
```

Finally we attach two event handlers to the body of the web page. We start with the mousemove event handler, which is executed every time the user moves the mouse, and adds a message "Event: mousemove" to the second list element by calling the addMessage function:

```
document.body.addEventListener("mousemove", function() {
  var second = document.getElementById("second");
  addMessage(second, "Event: mousemove");
});
```

We also register a click event handler, which, whenever the user clicks the page, logs a message "Event: click", also to the second list element:

```
document.body.addEventListener("click", function(){
  var second = document.getElementById("second");
  addMessage(second, "Event: click");
});
```

The result of running and interacting with this application is shown in figure 2.2.

We'll use this example application to explore and illustrate the differences between different phases of the web application lifecycle. Let's start with the page-building phase.

Figure 2.2 **When the code from listing 2.1 runs, messages are logged depending on user actions.**

2.2   The page-building phase

Before a web application can be interacted with or even displayed, the page must be built from the information in the response received from the server (usually HTML, CSS, and JavaScript code). The goal of this page-building phase is to set up the UI of a web application, and this is done in two distinct steps:

1 Parsing the HTML and building the Document Object Model (DOM)
2 Executing JavaScript code

Step 1 is performed when the browser is processing HTML nodes, and step 2 is performed whenever a special type of HTML element—the script element (that contains or refers to JavaScript code)—is encountered. During the page-building phase, the browser can switch between these two steps as many times as necessary, as shown in figure 2.3.

Figure 2.3   **The page-building phase starts when the browser receives the code of the page. It's performed in two steps: parsing the HTML and building the DOM, and executing JavaScript code.**

2.2.1   *Parsing the HTML and building the DOM*

The page-building phase starts with the browser receiving the HTML code, which is used as a base on top of which the browser builds the page's UI. The browser does this by parsing the HTML code, one HTML element at a time, and building a DOM, a structured representation of the HTML page in which every HTML element is represented as a node. For example, figure 2.4 shows the DOM of the example page that's built until the first `script` element is reached.

Notice how the nodes in figure 2.4 are organized such that each node except the first one (the root `html` node ❶) has exactly one parent. For example, the `head` node ❷ has the `html` node ❶ as its parent. At the same time, a node can have any number of children. For example, the `html` node ❶ has two children: the `head` node ❷ and the `body` node ❼. Children of the same element are called *siblings*. (The `head` node ❷ and the `body` node ❼ are siblings.)

It's important to emphasize that, although the HTML and the DOM are closely linked, with the DOM being constructed from HTML, they aren't one and the same. You should think of the HTML code as a *blueprint* the browser follows when constructing the initial DOM—the UI—of the page. The browser can even fix problems that it

Figure 2.4   By the time the browser encounters the first `script` element, it has already created a DOM with multiple HTML elements (the nodes on the right).

Invalid HTML

```
<html>
  <head>
    <p>
      Hello
    </p>
  </head>

  <body>
  </body>
</html>
```

Error: Content elements such as paragraphs (p) should be descendents of the body element, not of the head element!

The DOM as specified by the HTML code

The produced DOM, fixed by the browser
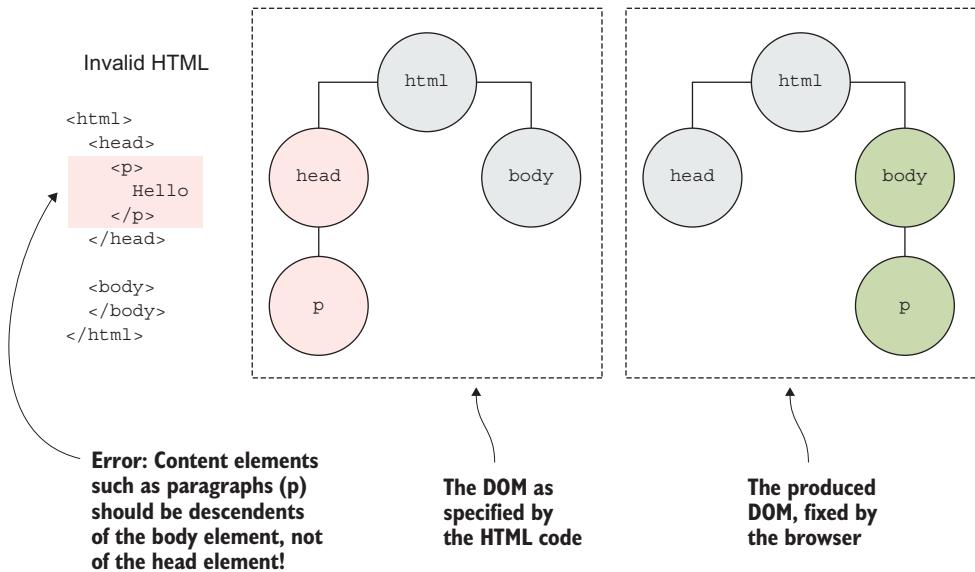
Figure 2.5   An example of invalid HTML that is fixed by the browser

finds with this blueprint in order to create a valid DOM. Let's consider the example shown in figure 2.5.

Figure 2.5 gives a simple example of erroneous HTML code in which a paragraph element is placed in the head element. The intention of the head element is that it is used for providing general page information: for example, the page title, character encodings, and external styles and scripts. It isn't intended for defining page content, as in this example. Because this is an error, the browser silently fixes it by constructing the correct DOM (at right in figure 2.5), in which the paragraph element is placed in the body element, where the page content ought to be.

HTML specification and DOM specification

The current version of HTML is HTML5, whose specification is available at https://html.spec.whatwg.org/. If you need something more readable, we recommend Mozilla's HTML5 guide, available at https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5.

The DOM, on the other hand, is evolving a bit more slowly. The current version is DOM3, whose specification is available at https://dom.spec.whatwg.org/. Again, Mozilla has prepared a report that can be found at https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

During page construction, the browser can encounter a special type of HTML element, the `script` element, which is used for including JavaScript code. When this happens, the browser pauses the DOM construction from HTML code and starts executing JavaScript code.

2.2.2 *Executing JavaScript code*

All JavaScript code contained in the `script` element is executed by the browser's JavaScript engine; for example, Firefox's Spidermonkey, Chrome and Opera's V8, or Edge's (IE's) Chakra. Because the primary purpose of JavaScript code is to provide dynamicity to the page, the browser provides an API through a global object that can be used by the JavaScript engine to interact with and modify the page.

GLOBAL OBJECTS IN JAVASCRIPT

The primary global object that the browser exposes to the JavaScript engine is the `window` object, which represents the window in which a page is contained. The `window` object is *the* global object through which all other global objects, global variables (even user-defined ones), and browser APIs are accessible. One of the most important properties of the global `window` object is the `document`, which represents the DOM of the current page. By using this object, the JavaScript code can alter the DOM of the page to any degree, by modifying or removing existing elements, and even by creating and inserting new ones.

Let's look at a snippet of code from listing 2.1:

```
var first = document.getElementById("first");
```

This example uses the global `document` object to select an element with the ID `first` from the DOM and assign it to a variable `first`. We can then use JavaScript code to make all sorts of modifications to that element, such as changing its textual content, modifying its attributes, dynamically creating and adding new children to it, and even removing the element from the DOM.

> **Browser APIs**
>
> Throughout the book, we use a number of browser built-in objects and functions (for example, `window` and `document`). Unfortunately, covering everything supported by the browser lies beyond the scope of a JavaScript book. Luckily, Mozilla again has our backs with https://developer.mozilla.org/en-US/docs/Web/API, where you can find the current status of the Web API Interfaces.

With this basic understanding of the global objects provided by the browser, let's look at two different types of JavaScript code that define exactly when that code is executed.

DIFFERENT TYPES OF JAVASCRIPT CODE

We broadly differentiate between two different types of JavaScript code: *global code* and *function code.* The following listing will help you understand the differences between these two types of code.

Listing 2.2   Global and function JavaScript code

```
<script>
  function addMessage(element, message){
    var messageElement = document.createElement("li");
    messageElement.textContent = message;
    element.appendChild(messageElement);
  }

  var first = document.getElementById("first");
  addMessage(first, "Page loading");
</script>
```

> **Function code is the code contained in a function.**

> **Global code is the code outside functions.**

The main difference between these two types of code is their placement: the code contained in a function is called *function code,* whereas the code placed outside all functions is called *global code.*

These two code types also differ in their execution (you'll see some additional differences later on, especially in chapter 5). Global code is executed automatically by the JavaScript engine (more on that soon) in a straightforward fashion, line by line, as it's encountered. For example, in listing 2.2, the pieces of global code that define the addMessage function use the built-in getElementById method to fetch the element with ID first and call the addMessage function; they are executed one after another as they're encountered, as shown in figure 2.6.

On the other hand, function code, in order to be executed, has to be called by something else: either by global code (for example, the addMessage function call in the global code causes the execution of the addMessage function code), by some other function, or by the browser (more on this soon).

EXECUTING JAVASCRIPT CODE IN THE PAGE-BUILDING PHASE

When the browser reaches the script node in the page-building phase, it pauses the DOM construction based on HTML code and starts executing JavaScript code instead.

```
<script>
function addMessage(element, message){
  var messageElement = document.createElement("li");
  messageElement.textContent = message;
  element.appendChild(messageElement);
}

var first = document.getElementById("first");
addMessage(first, "Page loading");
</script>
```

Figure 2.6   Program execution flow when executing JavaScript code

This means executing the global JavaScript code contained in the `script` element (and functions called by the global code are also executed). Let's go back to the example from listing 2.1.

Figure 2.7 shows the state of the DOM after the global JavaScript code has been executed. Let's walk slowly through its execution. First a function `addMessage` is defined:

```
function addMessage(element, message){
  var messageElement = document.createElement("li");
  messageElement.textContent = message;
  element.appendChild(messageElement);
 }
```

Then an existing element is fetched from the DOM by using the global `document` object and its `getElementById` method:

```
var first = document.getElementById("first");
```

```
<!DOCTYPE html>
<html>
  <head>
    <title> Web app lifecycle </title>
    <style>
      #first { color: green;}
      #second { color: red;}
    </style>
  </head>
  <body>
    <ul id="first"></ul>
    <script>
      function addMessage(element, message){
        var messageElement = document.createElement("li");
        messageElement.textContent = message;
        element.appendChild(messageElement);
      }

      var first = document.getElementById("first");
      addMessage(first, "Page loading");
    </script>
    <ul id="second"></ul>
    …
```


**addMessage(first, "Page loading");
creates the following node and
adds it to the DOM.**

**document.getElementById("first")
returns this node from the DOM.**
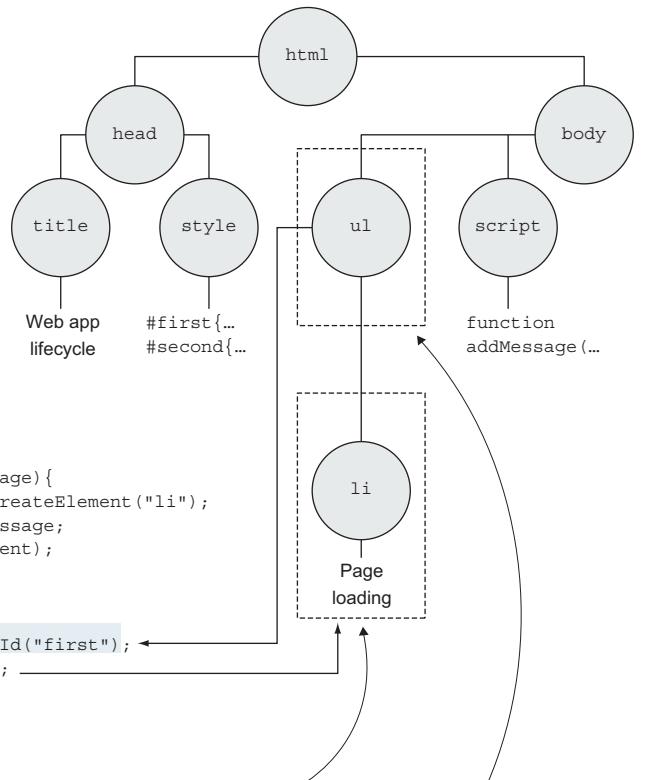
Figure 2.7   The DOM of the page after executing the JavaScript code contained in the `script` element

This is followed by a call to the `addMessage` function

```
addMessage(first, "Page loading");
```

which causes the creation of a new `li` element, the modification of its text content, and finally its insertion into the DOM.

In this example, the JavaScript code modifies the current DOM by creating a new element and inserting it into the DOM. But in general, JavaScript code can modify the DOM to any degree: It can create new nodes and modify or remove existing DOM nodes. But there are also some things it can't do, such as select and modify elements that haven't yet been created. For example, we can't select and modify the `ul` element with the ID `second`, because that element is found after the current `script` node and hasn't yet been reached and created. That's one of the reasons people tend to put their `script` elements at the bottom of the page. That way, we don't have to worry about whether a particular HTML element has been reached.

Once the JavaScript engine executes the last line of JavaScript code in the `script` element (in figure 2.5, this means returning from the `addMessage` function), the browser exits the JavaScript execution mode and continues building DOM nodes by processing the remaining HTML code. If, during that processing, the browser again encounters a `script` element, the DOM creation from HTML code is again paused, and the JavaScript runtime starts executing the contained JavaScript code. It's important to note that the global state of the JavaScript application persists in the meantime. All user-defined global variables created during the execution of JavaScript code in one `script` element are normally accessible to JavaScript code in other `script` elements. This happens because the global `window` object, which stores all global JavaScript variables, is alive and accessible during the entire lifecycle of the page.

These two steps

1. Building the DOM from HTML
2. Executing JavaScript code

are repeated as long as there are HTML elements to process and JavaScript code to execute.

Finally, when the browser runs out of HTML elements to process, the page-building phase is complete. The browser then moves on to the second part of the web application lifecycle: *event handling*.

2.3 Event handling

Client-side web applications are GUI applications, which means they react to different kinds of events: mouse moves, clicks, keyboard presses, and so on. For this reason, the JavaScript code executed during the page-building phase, in addition to influencing the global application state and modifying the DOM, can also register event listeners (or handlers): functions that are executed by the browser when an event occurs. With these event handlers, we provide interactivity to our applications. But before taking a closer look at registering event handlers, let's go through the general ideas behind event handling.

2.3.1   Event-handling overview

The browser execution environment is, at its core, based on the idea that only a single piece of code can be executed at once: the so-called *single-threaded* execution model. Think of a line at the bank. Everyone gets into a single line and has to wait their turn to be "processed" by the tellers. But with JavaScript, only *one* teller window is open! Customers (events) are processed only one at a time, as their turn comes. All it takes is one person who thinks it's appropriate to do their financial planning for the entire fiscal year while they're at the teller's window (we've all run into them!) to gum up the works.

Whenever an event occurs, the browser should execute the associated event-handler function. But there's no guarantee that we have extremely patient users who will always wait an appropriate amount of time before triggering another event. For this reason, the browser needs a way to keep track of the events that have occurred but have yet to be processed. To do this, the browser uses an *event queue,* as shown in figure 2.8.
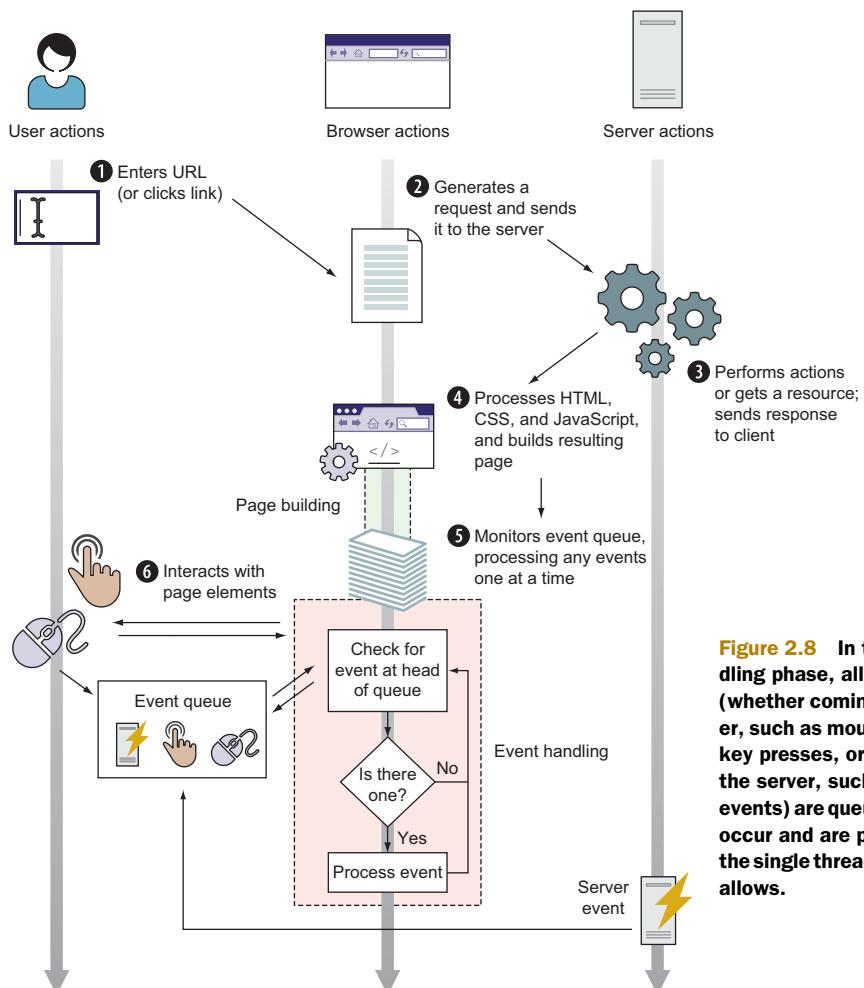
Figure 2.8 In the event-handling phase, all events (whether coming from the user, such as mouse clicks and key presses, or coming from the server, such as Ajax events) are queued up as they occur and are processed as the single thread of execution allows.

All generated events (it doesn't matter if they're user-generated, like mouse moves or key presses, or server-generated, such as Ajax events) are placed in the same event queue, in the order in which they're detected by the browser. As shown in the middle of figure 2.8, the event-handling process can then be described with a simple flowchart:

- The browser checks the head of the event queue.
- If there are no events, the browser keeps checking.
- If there's an event at the head of the event queue, the browser takes it and executes the associated handler (if one exists). During this execution, the rest of the events patiently wait in the event queue for their turn to be processed.

Because only one event is handled at a time, we have to be extra careful about the amount of time needed for handling events; writing event handlers that take a lot of time to execute leads to unresponsive web applications! (Don't worry if this sounds a bit vague; we'll come back to the event loop in chapter 13 and see exactly how it impacts the perceived performance of web applications).

It's important to note that the browser mechanism that puts events *onto* the queue is external to the page-building and event-handling phases. The processing that's necessary to determine when events have occurred and that pushes them onto the event queue doesn't participate in the thread that's *handling* the events.

EVENTS ARE ASYNCHRONOUS

Events, when they happen, can occur at unpredictable times and in an unpredictable order (it's tricky to force users to press keys or click in some particular order). We say that the handling of events, and therefore the invocation of their handling functions, is *asynchronous.*

The following types of events can occur, among others:

- Browser events, such as when a page is finished loading or when it's to be unloaded
- Network events, such as responses coming from the server (Ajax events, server-side events)
- User events, such as mouse clicks, mouse moves, and key presses
- Timer events, such as when a timeout expires or an interval fires

The vast majority of code executes as a result of such events!

The concept of event handling is central to web applications, and it's something you'll see again and again throughout the examples in this book: Code is set up in advance in order to be executed at a later time. Except for global code, the vast majority of the code we place on a page will execute as the result of some event.

Before events can be handled, our code has to notify the browser that we're interested in handling particular events. Let's look at how to register event handlers.

2.3.2 *Registering event handlers*

As we've already mentioned, event handlers are functions that we want executed whenever a particular event occurs. In order for this to happen, we have to notify the browser

that we're interested in an event. This is called *event-handler registration.* In client-side web applications, there are two ways to register events:

- By assigning functions to special properties
- By using the built-in `addEventListener` method

For example, writing the following code assigns a function to the special `onload` property of the `window` object:

```
window.onload = function(){};
```

An event handler for the `load` event (when the DOM is ready and fully built) is registered. (Don't worry if the notation on the right side of the assignment operator looks a bit funky; we'll talk at great length about functions in later chapters.) Similarly, if we want to register a handler for the `click` event on the document's body, we can write something along these lines:

```
document.body.onclick = function(){};
```

Assigning functions to special properties is an easy and straightforward way of registering event handlers, and you've probably run into it already. But we don't recommend that you register event handlers this way, because doing so comes with a drawback: It's only possible to register one function handler for a particular event. This means it's easy to overwrite previous event-handler functions, which can be a little frustrating. Luckily, there's an alternative: The `addEventListener` method enables us to register as many event-handler functions as we need. To show you an example, the following listing goes back to an excerpt of the example from listing 2.1.

Listing 2.3   Registering event handlers

```
<script>
  document.body.addEventListener("mousemove", function() {     ◁── Registers a handler
    var second = document.getElementById("second");                 for the mousemove
    addMessage(second, "Event: mousemove");                         event
  });

  document.body.addEventListener("click", function(){     ◁── Registers a handler
    var second = document.getElementById("second");             for the click event
    addmessage(second, "Event: click");
  });
</script>
```

This example uses the built-in `addEventListener` method on an HTML element to specify the type of event (`mousemove` or `click`) and the event-handler function. This means whenever the mouse is moved over the page, the browser calls a function that adds a message, `"Event: mousemove"`, to the list element with the ID `second` (a similar message, `"Event: click"`, is added to the same element whenever the body is clicked).

Now that you know how to set up event handlers, let's recall the simple flowchart you saw earlier and take a closer look at how events are handled.

2.3.3 Handling events

The main idea behind event handling is that when an event occurs, the browser calls the associated event handler. As we've already mentioned, due to the single-threaded execution model, only a single event handler can be executed at once. Any following events are processed only after the execution of the current event handler is fully complete!

Let's go back to the application from listing 2.1. Figure 2.9 shows an example execution in which a quick user has moved and clicked a mouse.

User actions

Browser actions

Page building

1 After the page was fully built, the user has, very quickly, moved and clicked the mouse. Two events are added to the event queue.

Event queue

Check for event at head of queue

Is there one?   No

Event handling

2 The event loop takes the event at the top of the queue, the mouse move event, and executes the associated event handler.

Yes

Process event

Event queue

3 After the mouse move event handler is fully processed, the mouse move event is taken off the queue, and the next event at the top of the queue is the mouse click event.

Figure 2.9   Example of an event-handling phase in which two events—`mousemove` and `click`—are handled

Let's examine what's going on here. As a response to these user actions, the browser puts the `mousemove` and `click` events onto the event queue in the order in which they have occurred: first the `mousemove` event and then the `click` event ❶.

In the event-handling phase, the event loop then checks the queue, sees that there's a `mousemove` event at the front of the queue, and executes the associated event handler ❷. While the `mousemove` handler is being processed, the `click` event waits in the queue for its turn. When the last line of the `mousemove` handler function has finished executing and the JavaScript engine exits the handler function, the `mousemove` event is fully processed ❸, and the event loop again checks the queue. This time, at the front of the queue, the event loop finds the `click` event and processes it. Once the `click` handler has finished executing, there are no new events in the queue, and the event loop keeps looping, waiting for new events to handle. This loop will continue executing until the user closes the web application.

Now that we have a sense of the overall steps that happen in the event-handling phase, let's see how this execution influences the DOM (figure 2.10). The execution of

```
<ul id="first"></ul>

<script>
  function addMessage(element, message){
    var messageElement = document.createElement('li');
    messageElement.textContent = message;
    element.appendChild(messageElement);
  }
</script>
```

```
<ul id="second"></ul>

<script>
  document.body.addEventListener('mousemove',function(){
    var second = document.getElementById('second');
    addMessage(second, 'Event: mousemove');
  });

  document.body.addEventListener('click',function(){
    var second = document.getElementById('second');
    addMessage(second, 'Event: click');
  });

</script>
```
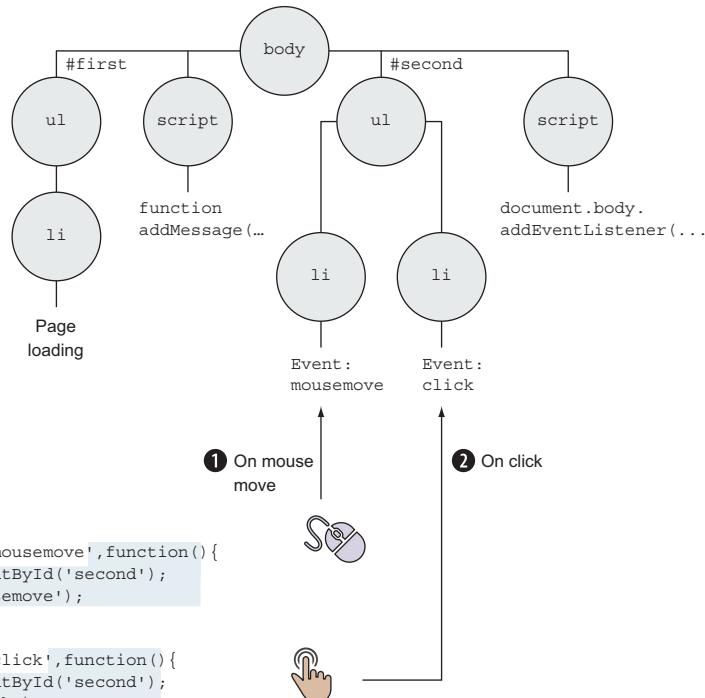
Figure 2.10   The DOM of the example application after handling the `mousemove` and `click` events

the `mousemove` handler selects the second list element with ID `second` and, by using the `addMessage` function, adds a new list item element ❶ with the text `"Event: mousemove"`. Once the execution of the `mousemove` handler is finished, the event loop executes the `click` handler, which leads to the creation of another list item element ❷, which is also appended to the second list element with the ID `second`.

Armed with this solid understanding of the lifecycle of client-side web applications, in the next part of the book, we'll start focusing on JavaScript the language, by learning the ins and outs of functions.

2.4   Summary

- The HTML code received by the browser is used as a blueprint for creating the DOM, an internal representation of the structure of a client-side web application.
- We use JavaScript code to dynamically modify the DOM to bring dynamic behavior to web applications.
- The execution of client-side web applications is performed in two phases:
  - *Page building*—HTML code is processed to create the DOM, and global JavaScript code is executed when script nodes are encountered. During this execution, the JavaScript code can modify the current DOM to any degree and can even register event handlers—functions that are executed when a particular event occurs (for example, a mouse click or a keyboard press). Registering event handlers is easy: Use the built-in `addEventListener` method.
  - *Event handling*—Various events are processed one at a time, in the order in which they were generated. The event-handling phase relies heavily on the event queue, in which all events are stored in the order in which they occurred. The event loop always checks the top of the queue for events, and if an event is found, the matching event-handler function is invoked.

2.5   Exercises

1. What are the two phases in the lifecycle of a client-side web application?
2. What is the main advantage of using the `addEventListener` method to register an event handler versus assigning a handler to a specific element property?
3. How many events can be processed at once?
4. In what order are events from the event queue processed?

Part 2

Understanding functions

Now that you're mentally prepared and you understand the environment in which JavaScript code is executed, you're ready to learn the fundamentals of the JavaScript features available to you.

In chapter 3, you'll learn all about the most important basic concept of JavaScript: no, not the object, but the function. This chapter will teach you why understanding JavaScript functions is the key to unlocking the secrets of the language.

Chapter 4 continues our in-depth exploration of functions by studying how functions are invoked, alongside all the ins and outs of implicit parameters that are accessible when executing function code.

Chapter 5 takes functions to the next level with closures—probably one of the most misunderstood (and even unknown) aspects of the JavaScript language. As you'll soon see, closures are closely tied to scopes. In this chapter, in addition to closures, we put a special focus on the scoping mechanisms in JavaScript.

Our exploration of functions will be completed in chapter 6, where we discuss a completely new type of function—the generator function—that has some special properties and is especially useful when dealing with asynchronous code.

3

First-class functions for the novice: definitions and arguments

This chapter covers

- Why understanding functions is so crucial
- How functions are first-class objects
- The ways to define a function
- The secrets of how parameters are assigned

You might be surprised, upon turning to this part of the book dedicated to JavaScript fundamentals, to see that the first topic of discussion is *functions* rather than *objects*. We'll certainly be paying plenty of attention to objects in part 3 of the book, but when it comes down to brass tacks, the main difference between writing JavaScript code like the average Jill (or Joe) and writing it like a JavaScript ninja is understanding JavaScript as a *functional language*. The level of sophistication of all the code you'll ever write in JavaScript hinges on this realization.

If you're reading this book, you're not a rank beginner. We're assuming that you know enough object fundamentals to get by (and we'll be taking a look at

more-advanced object concepts in chapter 7), but *really* understanding functions in JavaScript is the single most important weapon you can wield. So important, in fact, that this and the following three chapters are devoted to thoroughly understanding functions in JavaScript.

Most important, in JavaScript, functions are *first-class objects,* or *first-class citizens* as they're often called. They coexist with, and can be treated like, any other JavaScript object. Just like the more mundane JavaScript data types, they can be referenced by variables, declared with literals, and even passed as function parameters. In this chapter, we'll first take a look at the difference that this orientation to functions brings, and you'll see how this can help us write more compact and easy-to-understand code, by allowing us to define functions right where we need them. We'll also explore how to take advantage of functions as first-class objects in order to write better-performing functions. You'll see various ways of defining functions, even including some new types, such as *arrow* functions, which will help you write more elegant code. Finally, we'll look at the difference between function parameters and function arguments, with a special focus on ES6 additions, such as the rest and default parameters.

Let's start by going through some of the benefits of functional programming.

. . . .

Do you know?   In what situations might callback functions be used synchronously? Asynchronously?

What's the difference between an arrow function and a function expression?

Why might you need to use default parameter values in a function?

. . . .

3.1   *What's with the functional difference?*

One of the reasons that functions and functional concepts are so important in JavaScript is that functions are primary modular units of execution. Except for the global JavaScript code executed in the page-building phase, all of the script code that we'll write for our pages will be within a function.

Because most of our code will run as the result of a function invocation, you'll see that having functions that are versatile and powerful constructs gives us a great deal of flexibility and sway when writing code. Significant chunks of this book explain just how the nature of functions as first-class objects can be exploited to our great benefit. But first, let's take a look at some of the actions we can take with objects. In JavaScript, objects enjoy certain capabilities:

- They can be created via literals: { }
- They can be assigned to variables, array entries, and properties of other objects:

**Assigns a new object
to a variable**

**Adds a new object
to an array**

```
var ninja = {};
ninjaArray.push({});
ninja.data = {};
```

**Assigns a new object as a
property of another object**

- They can be passed as arguments to functions:

```
function hide(ninja){
    ninja.visibility = false;
}
hide({});
```

**A newly created object
passed as an argument
to a function**

- They can be returned as values from functions:

```
function returnNewNinja() {
  return {};
}
```

**Returns a new object
from a function**

- They can possess properties that can be dynamically created and assigned:

```
var ninja = {};
ninja.name = "Hanzo";
```

**Creates a new property
on an object**

It turns out that, unlike in many other programming languages, in JavaScript we can do almost the exact same things with functions also.

3.1.1 *Functions as first-class objects*

Functions in JavaScript possess all the capabilities of objects and are thus treated like any other object in the language. We say that functions are *first-class* objects, which can also be

- Created via literals

```
function ninjaFunction() {}
```

- Assigned to variables, array entries, and properties of other objects

**Assigns a new
function to a variable**

**Adds a new function
to an array**

```
var ninjaFunction = function() {};
ninjaArray.push(function(){});
ninja.data = function(){};
```

**Assigns a new function as a
property of another object**

- Passed as arguments to other functions

```
function call(ninjaFunction){
  ninjaFunction();
}
call(function(){});
```
A newly created function passed as an argument to a function

- Returned as values from functions

```
function returnNewNinjaFunction() {
  return function(){};
}
```
Returns a new function

- They can possess properties that can be dynamically created and assigned:

```
var ninjaFunction = function(){};
ninjaFunction.name = "Hanzo";
```
Adds a new property to a function

Whatever we can do with objects, we can do with functions as well. Functions *are* objects, just with an additional, special capability of being *invokable:* Functions can be called or invoked in order to perform an action.

Functional programming in JavaScript

Having functions as first-class objects is the first step toward *functional programming*, a style of programming that's focused on solving problems by composing functions (instead of specifying sequences of steps, as in more mainstream, imperative programming). Functional programming can help us write code that's easier to test, extend, and modularize. But it's a big topic, and in this book we only give it a nod (for example, in chapter 9). If you're interested in learning how to take advantage of functional programming concepts and apply them to your JavaScript programs, we recommend *Functional Programming in JavaScript* by Luis Atencio (Manning, 2016), available at www.manning.com/books/functional-programming-in-javascript.

One of the characteristics of first-class objects is that they can be passed to functions as arguments. In the case of functions, this means that we pass a function as an argument to another function that might, at a later point in application execution, call the passed-in function. This is an example of a more general concept known as a *callback function*. Let's explore this important concept.

3.1.2   *Callback functions*

Whenever we set up a function to be called at a later time, whether by the browser in the event-handling phase or by other code, we're setting up a *callback*. The term stems from the fact that we're establishing a function that other code will later "call back" at an appropriate point of execution.

Callbacks are an essential part of using JavaScript effectively, and we're willing to bet that you already use them in your code a lot—whether executing code on a button click, receiving data from a server, or animating parts of your UI.

In this section, we're about to look at how to use callbacks to handle events or to easily sort collections—typical real-world examples of how callbacks are used. But it's a tad complex, so before diving in, let's strip the callback concept completely naked and examine it in its simplest form. We'll start with an illuminating example of a completely useless function that accepts a reference to another function as a parameter and calls that function as a callback:

```
function useless(ninjaCallback) {
  return ninjaCallback();
}
```

As useless as this function is, it demonstrates the ability to pass a function as an argument to another function, and to subsequently invoke that function through the passed parameter.

We can test this useless function with the code in the following listing.

Listing 3.1   A simple callback example

```
var text = "Domo arigato!";
report("Before defining functions");

function useless(ninjaCallback) {          Defines a function that takes
  report("In useless function");           a callback function and
  return ninjaCallback();                   immediately invokes it
}

function getText() {                Defines a simple
  report("In getText function");     function that returns
  return text;                       a global variable
}

report("Before making all the calls");

assert(useless(getText) === text,          Calls our useless function with
       "The useless function works! " + text);   the getText function as a callback

report("After the calls have been made");
```

In this listing, we use a custom `report` function to output several messages as our code is being executed, so that we can track the execution of our program. We also use the `assert` testing function that we mentioned in chapter 1. The `assert` function usually takes two arguments. The first argument is an expression whose premise is asserted. In this case, we want to establish whether the result of invoking our `useless` function with the argument `getText` returns a value that's equal to the value of the variable `text` (`useless(getText) === text`). If the first argument evaluates to `true`, the assertion passes; otherwise, it's considered a failure. The second argument is the associated message, which is usually logged with an appropriate pass/fail indicator. (Appendix C

discusses testing in general, as well as our own little implementation of the `assert` and `report` functions).

When we run this code, we end up with the result shown in figure 3.1. As you can see, calling the `useless` function with our `getText` callback function as an argument returns the expected value.

We can also take a look at how exactly this simple callback example is executed. As figure 3.2 shows, we pass in the `getText` function to the `useless` function as an argument. This means that within the body of the `useless` function, the `getText` function can be referenced through the `callback` parameter. Then, by making the `call-`

Figure 3.1   **The result of running the code from listing 3.1**

back() call, we cause the execution of the `getText` function; the `getText` function, which we passed in as an argument, is called back by the `useless` function.
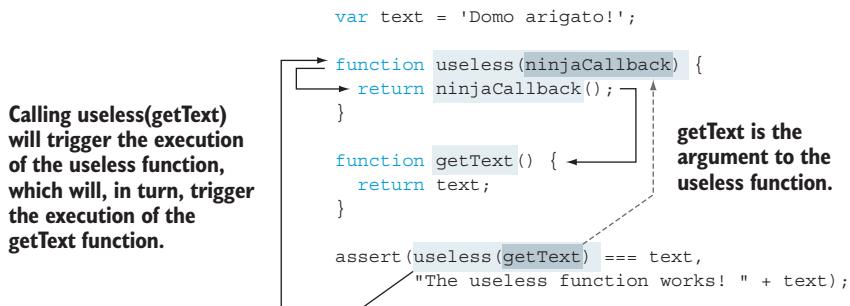
Figure 3.2   **The flow of execution when making the `useless(getText)` call. The `useless` function is called with `getText` as an argument. In the body of the `useless` function is a call to the passed-in function, which in this case triggers the execution of the `getText` function (we've "called back" to the `getText` function).**

This is easy, because JavaScript's functional nature lets us deal with functions as first-class objects. We can even take the whole thing a step further, by rewriting our code in the following manner:

```
var text = 'Domo arigato!';

function useless(ninjaCallback) {
  return ninjaCallback();
}

assert(useless(function () { return text;}) === text,
       "The useless function works! " + text);
```

Defines a callback function directly as an argument

One of the most important features of JavaScript is the ability to create functions in the code anywhere an expression can appear. In addition to making the code more compact and easy to understand (by putting function definitions near where they're used), this feature can also eliminate the need to pollute the global namespace with unnecessary names when a function isn't going to be referenced from multiple places within the code.

In the preceding example of a callback, we called our own callback. But callbacks can also be called by the browser. Think back to chapter 2, which has an example with the following snippet:

```javascript
document.body.addEventListener("mousemove", function() {
  var second = document.getElementById("second");
  addMessage(second, "Event: mousemove");
});
```

That's also a callback function, one that's defined as an event handler to the `mousemove` event, and that will be called by the browser when that event occurs.

> **NOTE** This section introduces callbacks as functions that other code will later "call back" at an appropriate point of execution. You've seen an example in which our own code immediately calls the provided callback (the `useless` function example), as well as an example in which the browser makes the call (the `mousemove` example) whenever a particular event happens. It's important to note that, unlike us, some people believe that a callback has to be called asynchronously, and therefore that the first example isn't really a callback. We mention this just in case you stumble upon some heated discussion.

Now let's consider a use of callbacks that will greatly simplify how we sort collections.

SORTING WITH A COMPARATOR

Almost as soon as we *have* a collection of data, odds are we're going to need to sort it. Let's say that we have an array of numbers in a random order: 0, 3, 2, 5, 7, 4, 8, 1. That order might be just fine, but chances are that, sooner or later, we'll want to rearrange it.

Usually, implementing sorting algorithms isn't the most trivial of programming tasks; we have to select the best algorithm for the job at hand, implement it, adapt it to our current need (so that the items are sorted in a particular order), and be careful not to introduce bugs. Out of these tasks, the only one that's application specific is the sorting order. Luckily, all JavaScript arrays have access to the `sort` method that requires us only to define a comparison algorithm that tells the sort algorithm how the values should be ordered.

This is where callbacks jump in! Instead of letting the sort algorithm decide what values go before other values, *we'll* provide a function that performs the comparison. We'll give the sort algorithm access to this function as a callback, and the algorithm will call the callback whenever it needs to make a comparison. The callback is expected to return a positive number if the order of the passed values should be

reversed, a negative number if not, and zero if the values are equal; subtracting the compared values produces the desired return value to sort the array:

```
var values = [0, 3, 2, 5, 7, 4, 8, 1];

values.sort(function(value1, value2){
  return value1 - value2;
});
```

There's no need to think about the low-level details of a sorting algorithm (or even which sorting algorithm to choose). We provide a callback that the JavaScript engine will call every time it needs to compare two items.

The *functional* approach allows us to create a function as a standalone entity, just as we can any other object type, and to pass it as an argument to a method, just like any other object type, which can accept it as a parameter, just like any other object type. It's that first-class status coming into play.

3.2   *Fun with functions as objects*

In this section, we'll examine ways to exploit the similarities that functions share with other object types. One capability that might be surprising is that there's nothing stopping us from attaching properties to functions:

```
var ninja = {};                       Creates an object and assigns
ninja.name = "hitsuke";               a new property to it

var wieldSword = function(){};        Creates a function and
wieldSword.swordType = "katana";      assigns a new property to it
```

Let's look at a couple of the more interesting things that can be done with this capability:

- *Storing functions in a collection* allows us to easily manage related functions—for example, callbacks that have to be invoked when something of interest occurs.
- *Memoization* allows the function to remember previously computed values, thereby improving the performance of subsequent invocations.

Let's get cracking.

3.2.1   *Storing functions*

In certain cases (for example, when we need to manage collections of callbacks that should be invoked when a certain event occurs), we want to store collections of unique functions. When adding functions to such a collection, a challenge we can face is determining which functions are new to the collection and should be added, and which are already resident and shouldn't be added. In general, when managing callback collections, we don't want any duplicates, because a single event would result in multiple calls to the same callback.

An obvious, but naïve, technique is to store all the functions in an array and loop through the array, checking for duplicate functions. Unfortunately, this performs poorly, and as a ninja, we want to make things work *well*, not merely work. We can use function properties to achieve this with an appropriate level of sophistication, as shown in the next listing.

Listing 3.2   Storing a collection of unique functions

```
var store = {
  nextId: 1,            ◁────
  cache: {},            ◁────
  add: function(fn) {
    if (!fn.id) {
      fn.id = this.nextId++;
      this.cache[fn.id] = fn;
      return true;
    }
  }
};
function ninja(){}
assert(store.add(ninja),
       "Function was safely added.");
assert(!store.add(ninja),
       "But it was only added once.");
```

Keeps track of the next available ID to be assigned

Creates an object to serve as a cache in which we'll store functions

Adds functions to the cache, but only if they're unique

Tests that all works as planned

In this listing, we create an object assigned to the variable `store`, in which we'll store a unique set of functions. This object has two data properties: one that stores a next available `id` value, and one within which we'll `cache` the stored functions. Functions are added to this cache via the `add()` method:

```
add: function(fn) {
  if (!fn.id) {
    fn.id = this.nextId++;
    this.cache[fn.id] = fn;
    return true;
  }
 ...
```

Within `add`, we first check to see whether the function has already been added to the collection by looking for the existence of the `id` property. If the current function has an `id` property, we assume that the function has already been processed and we ignore it. Otherwise, we assign an `id` property to the function (incrementing the `nextId` property along the way) and add the function as a property of the `cache`, using the `id` value as the property name. We then return the value `true`, so that we can tell when the function was added after a call to `add()`.

Running the page in the browser shows that when our tests try to add the ninja() function twice, the function is added only once, as shown in figure 3.3. Chapter 9 shows an even better technique for working with collections of unique items that utilize sets, a new type of object available in ES6.

Another useful trick to pull out of our sleeves when using function properties is giving a function the ability to modify itself. This technique can be used to remember previously computed values, saving time during future computations.

Figure 3.3 By tacking a property onto a function, we can keep track of it. In that way, we can be sure that our function has been added only once.

3.2.2 Self-memoizing functions

As noted earlier, *memoization* (no, that's not a typo) is the process of building a function that's capable of remembering its previously computed values. In a nutshell, whenever a function computes its result, we store that result alongside the function arguments. In this way, when another invocation occurs for the same set of arguments, we can return the previously stored result, instead of calculating it anew. This can markedly increase performance by avoiding needless complex computations that have already been performed. Memoization is particularly useful when performing calculations for animations, searching data that doesn't change that often, or any time-consuming math.

As an example, let's look at a simplistic (and certainly not particularly efficient) algorithm for computing prime numbers. Although this is a simple example of a complex calculation, this technique is readily applicable to other expensive computations (such as deriving the MD5 hash for a string) that are too complex to present here.

From the outside, the function appears to be just like any normal function, but we'll surreptitiously build in an answer cache in which the function will save the answers to the computations it performs. Look over the following code.

Listing 3.3   Memoizing previously computed values

```
function isPrime(value){
  if (!isPrime.answers){
    isPrime.answers = {};          Creates the cache
  }

  if (isPrime.answers[value] !== undefined) {
    return isPrime.answers[value];     Checks for cached values
  }

  var prime = value !== 1; // 1 is not a prime
```

```
  for (var i = 2; i < value; i++) {
    if (value % i === 0) {
      prime = false;
      break;
    }
  }
  return isPrime.answers[value] = prime;          ⊲——— Stores the computed value
}

assert(isPrime(5), "5 is prime!" );                   Tests that it
assert(isPrime.answers[5], "The answer was cached!" );   all works
```

Within the `isPrime` function, we start by checking whether the `answers` property that we'll use as a cache has been created, and if not, we create it:

```
if (!isPrime.answers) {
    isPrime.answers = {};
}
```

The creation of this initially empty object will occur only on the first call to the function; after that, the cache will exist.

    Then we check whether the result for the passed value has already been cached in `answers`:

```
  if (isPrime.answers[value] !== undefined) {
    return isPrime.answers[value];
  }
```

Within this cache, we'll store the computed answer (`true` or `false`) using the argument `value` as the property key. If we find a cached answer, we return it.

    If no cached value is found, we go ahead and perform the calculations needed to determine whether the value is `prime` (which can be an expensive operation for larger values) and store the result in the cache as we return it:

```
return isPrime.answers[value] = prime;
```

Our cache is a property of the function itself, so it's kept alive for as long as the function itself is alive.

    Finally, we test that the memoization is working!

```
assert(isPrime(5), "5 is prime!" );
assert(isPrime.answers[5], "The answer was cached!" );
```

This approach has two major advantages:

- The end user enjoys performance benefits for function calls asking for a previously computed value.
- It happens seamlessly and behind the scenes; neither the end user nor the page author needs to perform any special requests or do any extra initialization in order to make it all work.

But it's not all roses and violins; its disadvantages may need to be weighed against its advantages:

- Any sort of caching will certainly sacrifice memory in favor of performance.
- Purists may consider that caching is a concern that shouldn't be mixed with the business logic; a function or a method should do one thing and do it well. But don't worry; in chapter 8, you'll see how to tackle this complaint.
- It's difficult to load-test or measure the performance of an algorithm such as this one, because our results depend on the previous inputs to the function.

Now that you've seen some of the practical use cases of first-class functions, let's explore the various ways of defining functions.

3.3 *Defining functions*

JavaScript functions are usually defined by using a *function literal* that creates a function value in the same way that, for example, a numeric literal creates a numeric value. Remember that as first-class objects, functions are values that can be used in the language just like other values, such as strings and numbers. And whether you realize it or not, you've been doing that all along.

JavaScript provides a couple of ways to define functions, which can be divided into four groups:

- *Function declarations* and *function expressions*—The two most common, yet subtly different ways of defining functions. Often people don't even consider them as separate, but as you'll see, being aware of their differences can help us understand when our functions are available for invocation:

  ```
  function myFun(){ return 1;}
  ```

- *Arrow functions* (often called *lambda functions*)—A recent, ES6 addition to the JavaScript standard that enables us to define functions with far less syntactic clutter. They even solve one common problem with callback functions, but more on that later:

  ```
  myArg => myArg*2
  ```

- *Function constructors*—A not-so-often used way of defining functions that enables us to dynamically construct a new function from a string that can also be dynamically generated. This example dynamically creates a function with two parameters, a and b, that returns the sum of those two parameters:

  ```
  new Function('a', 'b', 'return a + b')
  ```

- *Generator functions*—This ES6 addition to JavaScript enable us to create functions that, unlike normal functions, can be exited and reentered later in the application execution, while keeping the values of their variables across these re-entrances. We can define generator versions of *function declarations*, *function expressions*, and *function constructors*:

  ```
  function* myGen(){ yield 1; }
  ```

It's important that you understand these differences, because the way in which a function is defined significantly influences when the function is available to be invoked and how it behaves, as well as on which object the function can be invoked.

In this chapter, we'll explore function declarations, function expressions, and arrow functions. You'll learn their syntax and how they work, and we'll come back to them multiple times throughout the book to explore their specifics. Generator functions, on the other hand, are rather peculiar and are significantly different from the standard functions. We'll revisit them in detail in chapter 6.

That leaves us with function constructors, a JavaScript feature that we'll skip entirely. Although it has some interesting applications, especially when dynamically creating and evaluating code, we consider it a corner feature of the JavaScript language. If you want to know more about function constructors, visit http://mng.bz/ZN8e.

Let's start with the simplest, most traditional ways of defining functions: *function declarations* and *function expressions.*

3.3.1 Function declarations and function expressions

The two most common ways of defining functions in JavaScript are by using function declarations and function expressions. These two techniques are so similar that often we don't even make a distinction between them, but as you'll see in the following chapters, subtle differences exist.

FUNCTION DECLARATIONS

The most basic way of defining a function in JavaScript is by using function declarations (see figure 3.4). As you can see, every function declaration starts with a mandatory `function` keyword, followed by a mandatory function name and a list of optional comma-separated parameter names enclosed within mandatory parentheses. The function body, which is a potentially empty list of statements, must be enclosed within an opening and a closing brace. In addition to this form, which every function declaration must satisfy, there's one more condition: A function declaration must be placed on its own, as a separate JavaScript statement (but can be
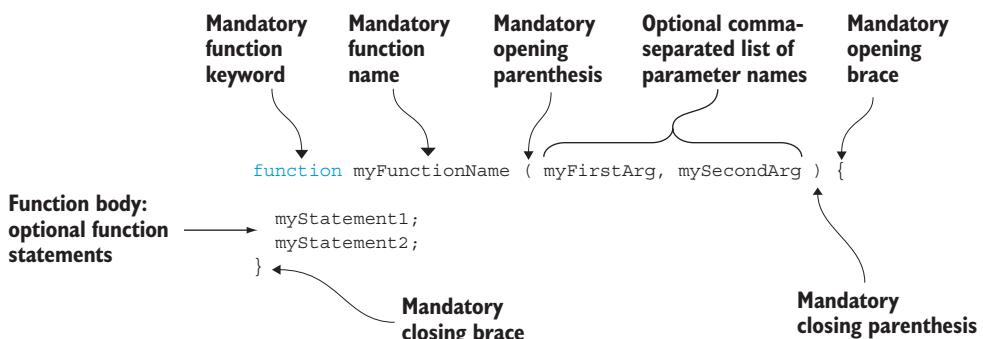
Figure 3.4 The function declaration stands on its own, as a separate block of JavaScript code! (It can be contained within other functions.)

contained within another function or a block of code; you'll see exactly what we mean by that in the next section).

A couple of function declaration examples are shown in the following listing.

Listing 3.4   Examples of function declarations

```
function samurai() {                  Defines function samurai
  return "samurai here";              in the global code
}
                                      Defines function ninja
                                      in the global code
function ninja() {            ◁─────┘

  function hiddenNinja() {            Defines function
    return "ninja here";             hiddenNinja within
  }                                   the ninja function

  return hiddenNinja();
}
```

If you take a closer look, you'll see something that you might not be accustomed to, if you haven't had much exposure to functional languages: a function defined within another function!

```
function ninja() {
  function hiddenNinja() {
    return "ninja here";
  }
  return hiddenNinja();
}
```

In JavaScript, this is perfectly normal, and we've used it here to again emphasize the importance of functions in JavaScript.

> **NOTE**   Having functions contained in other functions might raise some tricky questions regarding scope and identifier resolution, but save them for now, because we'll revisit this case in detail in chapter 5.

FUNCTION EXPRESSIONS

As we've already mentioned multiple times, functions in JavaScript are first-class objects, which, among other things, means that they can be created via literals, assigned to variables and properties, and used as arguments and return values to and from other functions. Because functions are such fundamental constructs, JavaScript enables us to treat them as any other expressions. So, just as we can use number literals, for example

```
var a = 3;
myFunction(4);
```
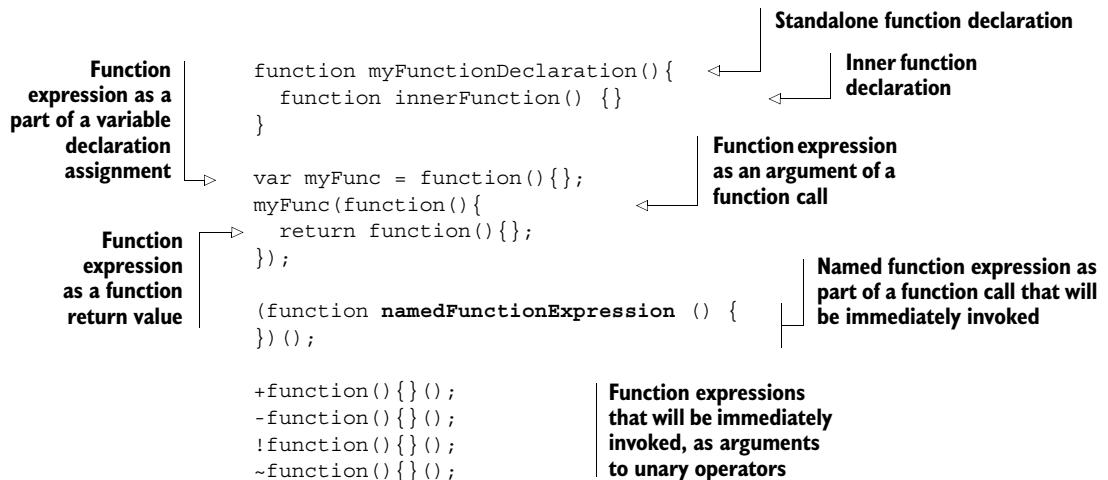
so too we can use function literals, in the same locations

```
var a = function() {};
myFunction(function(){});
```

Such functions that are always a part of another statement (for example, as the right side of an assignment expression, or as an argument to another function) are called *function expressions.* Function expressions are great because they allow us to define functions exactly where we need them, in the process making our code easier to understand.

The following listing shows the differences between function declarations and function expressions.

Listing 3.5   Function declarations and function expressions

Standalone function declaration

Inner function declaration

Function expression as a part of a variable declaration assignment

```
function myFunctionDeclaration(){
  function innerFunction() {}
}

var myFunc = function(){};
myFunc(function(){
  return function(){};
});

(function namedFunctionExpression () {
})();

+function(){}();
-function(){}();
!function(){}();
~function(){}();
```

Function expression as an argument of a function call

Function expression as a function return value

Named function expression as part of a function call that will be immediately invoked

Function expressions that will be immediately invoked, as arguments to unary operators

This example code begins with a standard function declaration that contains another inner function declaration:

```
function myFunctionDeclaration(){
  function innerFunction() {}
}
```

Here you can see how function declarations are separate statements of JavaScript code, but can be contained within the body of other functions.

In contrast are function expressions, which are always a part of another statement. They're placed on the expression level, as the right side of a variable declaration (or an assignment):

```
var myFunc = function(){};
```

Or as an argument to another function call, or as a function return value:

```
myFunc(function() {
  return function(){};
});
```

Besides the position in code where they're placed, there's one more difference between function declarations and function expressions: For function declarations, the function name is *mandatory*, whereas for function expressions it's completely *optional*.

Function declarations must have a name defined because they stand on their own. Because one of the basic requirements for a function is that it has to be invokable, we have to have a way to reference it, and the only way to do this is through its name.

Function expressions, on the other hand, are parts of other JavaScript expressions, so we have alternative ways to invoke them. For example, if a function expression is assigned to a variable, we can use that variable to invoke the function:

```
var doNothing = function(){};
doNothing();
```

Or, if it's an argument to another function, we can invoke it within that function through the matching parameter name:

```
function doSomething(action) {
  action();
}
```

IMMEDIATE FUNCTIONS

Function expressions can even be placed in positions where they look a bit weird at first, such as at a location where we'd normally expect a function identifier. Let's stop and take a closer look at that construct (see figure 3.5).
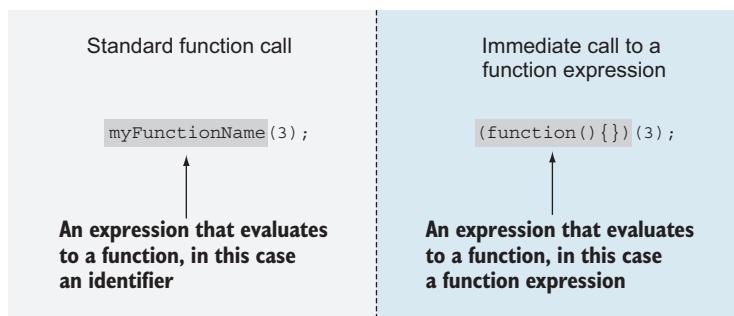
Figure 3.5   **A comparison of a standard function call and an immediate call to a function expression**

When we want to make a function call, we use an expression that evaluates to a function, followed by a pair of function call parentheses, which might contain arguments. In the most basic function call, we put an identifier that evaluates to a function, as on the left side of figure 3.5. But the expression to the left of the calling parenthesis doesn't have to be a simple identifier; it can be *any* expression that evaluates to a function. For example, a simple way to specify an expression that evaluates to a function is to use a function expression. So on the right side of figure 3.5, we first create a function, and then we immediately invoke that newly created function. This, by the way, is called an *immediately invoked function expression* (IIFE), or *immediate function* for short, and is an important concept in JavaScript development because it allows us to mimic modules in JavaScript. We'll focus on this application of IIFEs in chapter 11.

Parentheses around function expressions

One more thing might be nagging you about the way we've immediately called our function expression: the parentheses around the function expression itself. Why do we even need those? The reason is purely syntactical. The JavaScript parser has to be able to easily differentiate between function declarations and function expressions. If we leave out the parentheses around the function expression, and put our immediate call as a separate statement `function(){}(3)`, the JavaScript parser will start processing it, and will conclude, because it's a separate statement starting with the keyword `function`, that it's dealing with a function declaration. Because every function declaration has to have a name (and here we didn't specify one), an error will be thrown. To avoid this, we place the function expression within parentheses, signaling to the JavaScript parser that it's dealing with an expression, and not a statement.

There's also an alternative, even simpler way (yet, strangely, a little less often used) of achieving the same goal: `(function(){}(3))`. By wrapping the immediate function definition and call within parentheses, you can also notify the JavaScript parser that it's dealing with an expression.

The last four expressions in listing 3.5 are variations of the same theme of immediately invoked function expressions often found in various JavaScript libraries:

```
+function(){}();
-function(){}();
!function(){}();
~function(){}();
```

This time, instead of using parentheses around the function expressions to differentiate them from function declarations, we can use unary operators: +, -, !, and ~. We do this to signal to the JavaScript engine that it's dealing with expressions and not statements. Notice how the results of applying these unary operators aren't stored anywhere; from a computational perspective, they don't really matter; only the calls to our IIFEs matter.

Now that we've studied the ins and outs of the two most basic ways of defining functions in JavaScript (function declarations and function expressions), let's explore a new addition to the JavaScript standard: *arrow functions.*

3.3.2 Arrow functions

NOTE   Arrow functions are an ES6 addition to the JavaScript standard (for browser compatibility, see http://mng.bz/8bnH).

Because in our JavaScript we use *a lot* of functions, it makes sense to add some syntactic sugar that enables us to create functions in a shorter, more succinct way, thus making our lives as developers more pleasant.

In a lot of ways, arrow functions are a simplification of function expressions. Let's revisit our sorting example from the first section of this chapter:

```
var values = [0, 3, 2, 5, 7, 4, 8, 1];
values.sort(function(value1,value2){
  return value1 - value2;
});
```

This example uses a callback function expression sent to the sort method of the array object; this callback will be invoked by the JavaScript engine to sort the values of the array in descending order.

Now let's see how to do the exact same thing with arrow functions:

```
var values = [0, 3, 2, 5, 7, 4, 8, 1];
values.sort((value1,value2) => value1 - value2);
```

See how much more succinct this is?

There's no clutter caused by the function keyword, the braces, or the return statement. In a much simpler way than a function expression can, the arrow function states: here's a function that takes two arguments and returns their difference. Notice the introduction of a new operator, =>, the so-called *fat-arrow* operator (an equals sign immediately followed by a greater-than sign), that's at the core of defining an arrow function.

Now let's deconstruct the syntax of an arrow function, starting with the simplest possible way:

```
param => expression
```

This arrow function takes a parameter and returns the value of an expression. We can use this syntax as shown in the following listing.

Listing 3.6 Comparing an arrow function and a function expression

```
var greet = name => "Greetings " + name;          ⊲——— Defines an arrow function
assert(greet("Oishi") === "Greetings Oishi", "Oishi is properly greeted");

var anotherGreet = function(name){
  return "Greetings " + name;                      Defines a function
};                                                  expression
assert(anotherGreet("Oishi") === "Greetings Oishi",
       "Again, Oishi is properly greeted");
```

Take a while to appreciate how arrow functions make code more succinct, without sacrificing clarity. That's the simplest version of the arrow function syntax, but in general, the arrow function can be defined in two ways, as shown in figure 3.6.

As you can see, the arrow function definition starts with an optional comma-separated list of parameter names. If there are no parameters, or more than one parameter, this list must be enclosed within parentheses. But if we have only a single parameter, the parentheses are optional. This list of parameters is followed by a mandatory fat-arrow operator, which tells us and the JavaScript engine that we're dealing with an arrow function.

After the fat-arrow operator, we have two options. If it's a simple function, we put an expression there (a mathematical operation, another function invocation, whatever), and the result of the function invocation will be the value of that expression. For instance, our first arrow function example has the following arrow function:

```
var greet = name => "Greetings " + name;
```

Mandatory parentheses for 0 or more than I parameter; optional for I parameter.

Mandatory fat-arrow operator (equals and greater than).

If the body of the arrow function is an expression, the return value of the function will be the value of that expression.

```
(param1, param2) => expression
                    {
                       myStatement1;
                       myStatement2;
                    }
```

Optional comma-separated list of parameter names.

If the body of the arrow function is a block of code, the return value is as it would be in a standard function (undefined if there is no return statement, and the value of the return expression if there is).

Figure 3.6 The syntax of an arrow function

The return value of the function is a concatenation of the string `"Greetings "` with the value of the `name` parameter.

In other cases, when our arrow functions aren't that simple and require more code, we can include a block of code after the arrow operator. For example:

```
var greet = name => {
  var helloString = 'Greetings ';
  return helloString + name;
};
```

In this case, the return value of the arrow function behaves as in a standard function. If there's no return statement, the result of the function invocation will be `undefined`, and if there is, the result will be the value of the return expression.

We'll revisit arrow functions multiple times throughout this book. Among other things, we'll present additional features of arrow functions that will help us evade subtle bugs that can occur with more standard functions.

Arrow functions, like all other functions, can receive arguments in order to use them to perform their task. Let's see what happens with the values that we pass to a function.

3.4   *Arguments and function parameters*

When discussing functions, we often use the terms *argument* and *parameter* almost interchangeably, as if they were more or less the same thing. But now, let's be more formal:

- A *parameter* is a variable that we list as part of a function definition.
- An *argument* is a value that we pass to the function when we invoke it.

Figure 3.7 illustrates the difference.
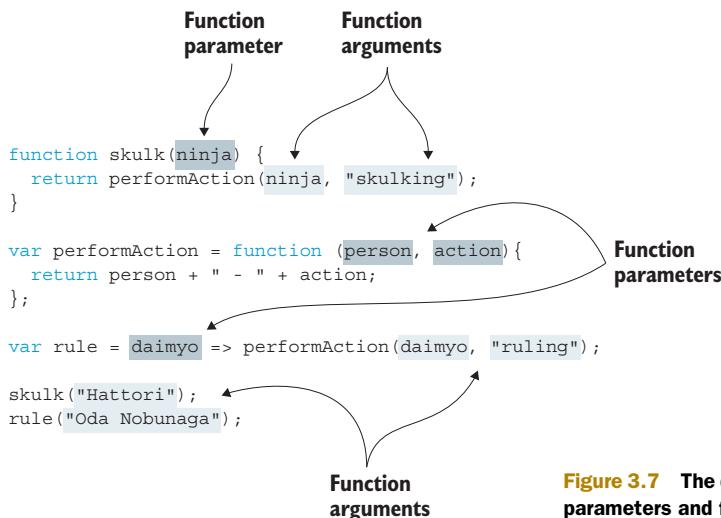
Figure 3.7   **The difference between function parameters and function arguments**

As you can see, a function parameter is specified with the definition of the function, and all types of functions can have parameters:

- Function declarations (the `ninja` parameter to the `skulk` function)
- Function expressions (the `person` and `action` parameters to the `performAction` function)
- Arrow functions (the `daimyo` parameter)

Arguments, on the other hand, are linked with the invocation of the function; they're values passed to a function at the time of its invocation:

- The string `Hattori` is passed as an argument to the `skulk` function.
- The string `Oda Nobunaga` is passed as an argument to the `rule` function.
- The parameter `ninja` of the `skulk` function is passed as an argument to the `performAction` function.

When a list of arguments is supplied as a part of a function invocation, these arguments are assigned to the parameters in the function definition in the order specified. The first argument gets assigned to the first parameter, the second argument to the second parameter, and so on.

If we have a different number of arguments than parameters, no error is raised. JavaScript is perfectly fine with this situation and deals with it in the following way. If more arguments are supplied than there are parameters, the "excess" arguments aren't assigned to parameter names. For example, see figure 3.8.

Figure 3.8 shows that if we were to call the `practice` function with `practice("Yoshi", "sword", "shadow sword", "katana")`, the arguments `Yoshi`, `sword`, and `shadow sword` would be assigned to the parameters `ninja`, `weapon`, and `technique`,

Argument "Yoshi" is assigned to parameter ninja.
Argument "sword" is assigned to parameter weapon.
Argument "shadow sword" is assigned to parameter technique.

Excess arguments aren't assigned to parameters.

```
practice ("Yoshi", "sword", "shadow sword", "katana");

function practice (ninja, weapon, technique) { … }

practice ("Yoshi");
```

Argument "Yoshi" is assigned to parameter ninja.
undefined is assigned to parameter weapon.
undefined is assigned to parameter technique.

Figure 3.8 Arguments are assigned to function parameters in the order specified. Excess arguments aren't assigned to any parameters.

respectively. The argument katana is an excess argument, and wouldn't be assigned to any parameter. In the next chapter, you'll see that even though some arguments aren't assigned to parameter names, we still have a way to access them.

On the other hand, if we have more parameters than arguments, the parameters that have no corresponding argument are set to undefined. For example, if we were to make the call practice("Yoshi"), the parameter ninja would be assigned the value Yoshi, while the parameters weapon and technique would be set to undefined.

Dealing with function arguments and parameters is as old as JavaScript itself, but now let's explore two new features of JavaScript bestowed by ES6: rest and default parameters.

3.4.1   Rest parameters

> **ES6** **NOTE**   Rest parameters are added by the ES6 standard (for browser compatibility, see http://mng.bz/3go1).

For our next example, we'll build a function that multiplies the first argument with the largest of the remaining arguments. This probably isn't something that's particularly applicable in our applications, but it's an example of yet more techniques for dealing with arguments within a function.

This might seem simple enough: We'll grab the first argument and multiply it by the biggest of the remaining argument values. In the old versions of JavaScript, this would require some workarounds (which we'll look at in the next chapter). Luckily, in ES6, we don't need to jump through any hoops. We can use *rest parameters,* as shown in the following listing.

Listing 3.7   Using rest parameters

```
function multiMax(first, ...remainingNumbers){          Rest parameters are
  var sorted = remainingNumbers.sort(function(a, b){    prefixed with ...
     return b – a;
  });                              Sort the remaining
  return first * sorted[0];        numbers, descending.
}
assert(multiMax(3, 1, 2, 3) == 9,           The function is called just
      "3*3=9 (First arg, by largest.)");    like any other function.
```

By prefixing the last-named argument of a function with an ellipsis (...), we turn it into an array called *the rest parameters*, which contains the remaining passed-in arguments.

```
function multiMax(first, ...remainingNumbers){
  ...
}
```

For example, in this case, the `multiMax` function is called with four arguments: `multiMax(3, 1, 2, 3)`. In the body of the `multiMax` function, the value of the first argument, 3, is assigned to the first `multiMax` function parameter, `first`. Because the second parameter of the function is the rest parameter, all remaining arguments (1, 2, 3) are placed in a new array: `remainingNumbers`. We then obtain the biggest number by sorting the array in descending order (notice how it's simple to change the sorting order) and picking the largest number, which is in the first place of our sorted array. (This is far from the most efficient way of determining the largest number, but why not take advantage of the skills we gained earlier in the chapter?)

> **NOTE** Only the last function parameter can be a rest parameter. Trying to put the ellipsis in front of any parameter that isn't last will bring us only sorrow, in the form of `SyntaxError: parameter after rest parameter`.

In the next section, we'll continue adding to our JavaScript tool belt with additional ES6 functionality: default parameters.

3.4.2 *Default parameters*

> **NOTE** Default parameters are added by the ES6 standard (for browser compatibility, see http://mng.bz/wI8w).

Many web UI components (especially jQuery plugins) can be configured. For example, if we're developing a slider component, we might want to give our users an option to specify a timer interval after which one item is replaced with another, as well as an animation that will be used as the change occurs. At the same time, maybe some users don't care and are happy to use whatever settings we offer. Default parameters are ideal for such situations!

Our little example with slider component settings is just a specific case of a situation in which *almost* all function calls use the same value for a particular parameter (notice the emphasis on *almost*). Consider a simpler case in which most of our ninjas are used to skulking around, but not Yagyu, who cares only about simple sneaking:

```
function  performAction(ninja, action) {
  return ninja + " " + action;
}
performAction("Fuma", "skulking");
performAction("Yoshi", "skulking");
performAction("Hattori", "skulking");
performAction("Yagyu", "sneaking");
```

Doesn't it seem tedious to always have to repeat the same argument, `skulking`, simply because Yagyu is obstinate and refuses to act like a proper ninja?

In other programming languages, this problem is most often solved with function overloading (specifying additional functions with the same name but a different set of parameters). Unfortunately, JavaScript doesn't support function overloading, so when faced with this situation in the past, developers often resorted to something like the following listing.

Listing 3.8 Tackling default parameters before ES6

> **If the action parameter is undefined, we use a default value, skulking, and if it's defined, we keep the passed-in value.**

```
function performAction(ninja, action){
    action = typeof action === "undefined" ? "skulking" : action;    ◁
    return ninja + " " + action;
}

assert(performAction("Fuma") === "Fuma skulking",
       "The default value is used for Fuma");

assert(performAction("Yoshi") === "Yoshi skulking",
       "The default value is used for Yoshi");
assert(performAction("Hattori") === "Hattori skulking",
       "The default value is used for Hattori");

assert(performAction("Yagyu", "sneaking") === "Yagyu sneaking",
       "Yagyu can do whatever he pleases, even sneak!");
```

> **We haven't passed in a second argument, the value of the action parameter; after executing the first function, the body statement will default to skulking.**

> **Pass a string as the value of the action parameter; that value will be used throughout the function body.**

Here we define a `performAction` function, which checks whether the value of the `action` parameter is `undefined` (by using the `typeof` operator), and if it is, the function sets the value of the `action` variable to `skulking`. If the `action` parameter is sent through a function call (it's not `undefined`), we keep the value.

> **NOTE** The `typeof` operator returns a string indicating the type of the operand. If the operand isn't defined (for example, if we haven't supplied a matching argument for a function parameter), the return value is the string `undefined`.

This is a commonly occurring pattern that's tedious to write, so the ES6 standard has added support for *default parameters*, as shown in the following listing.

Listing 3.9   Tackling default parameters in ES6

```
function performAction(ninja, action = "skulking"){
    return ninja + " " + action;
}

assert(performAction("Fuma") === "Fuma skulking",
      "The default value is used for Fuma");

assert(performAction("Yoshi") === "Yoshi skulking",
      "The default value is used for Yoshi");

assert(performAction("Hattori") === "Hattori skulking",
      "The default value is used for Hattori");

assert(performAction("Yagyu", "sneaking") === "Yagyu sneaking",
      "Yagyu can do whatever he pleases, even sneak!");
```

> In ES6, it's possible to assign a value to a function parameter.

> If the value isn't passed in, the default value is used.

> The passed value is used.

Here you can see the syntax of default function parameters in JavaScript. To create a default parameter, we assign a value to a function parameter:

```
function performAction(ninja, action = "skulking"){
    return ninja + " " + action;
}
```

Then, when we make a function call and the matching argument value is left out, as with `Fuma`, `Yoshi`, and `Hattori`, the default value (in this case, `skulking`), is used:

```
assert(performAction("Fuma") === "Fuma skulking",
      "The default value is used for Fuma");

assert(performAction("Yoshi") === "Yoshi skulking",
      "The default value is used for Yoshi");

assert(performAction("Hattori") === "Hattori skulking",
      "The default value is used for Hattori");
```

If, on the other hand, we specify the value, the default value is overridden:

```
assert(performAction("Yagyu", "sneaking") === "Yagyu sneaking",
      "Yagyu can do whatever he pleases, even sneak!");
```

We can assign any values to default parameters: simple, primitive values such as numbers or strings, but also complex types such as objects, arrays, and even functions. The values are evaluated on each function call, from left to right, and when assigning values to later default parameters, we can reference previous parameters, as in the following listing.

Listing 3.10   **Referencing previous default parameters**

We can place arbitrary expressions as default parameter values, in the process even referencing previous function parameters.

```
function performAction(ninja, action = "skulking",
                       message = ninja + " " + action) {   ◁
  return message;
}

assert(performAction("Yoshi") === "Yoshi skulking", "Yoshi is skulking");
```

Even though JavaScript allows you to do something like this, we urge caution. In our opinion, this doesn't enhance code readability and should be avoided, whenever possible. But moderate use of default parameters—as a means of avoiding null values, or as relatively simple flags that configure the behaviors of our functions—can lead to much simpler and more elegant code.

3.5 *Summary*

- Writing sophisticated code hinges upon learning JavaScript as a functional language.
- Functions are first-class objects that are treated just like any other objects within JavaScript. Similar to any other object type, they can be
  - Created via literals
  - Assigned to variables or properties
  - Passed as parameters
  - Returned as function results
  - Assigned properties and methods
- Callback functions are functions that other code will later "call back," and are often used, especially with event handling.
- We can take advantage of the fact that functions can have properties and that those properties can be used to store any information; for example
  - We can store functions in function properties for later reference and invocation.
  - We can use function properties to create a cache (memoization), thereby avoiding unnecessary computations.
- There are different types of functions: function declarations, function expressions, arrow functions, and function generators.
- Function declarations and function expressions are the two most common types of functions. Function declarations must have a name, and must be placed as separate statements in our code. Function expressions don't have to be named, but do have to be a part of another code statement.
- Arrow functions are a new addition to JavaScript, enabling us to define functions in a much more succinct way than with standard functions.
- A parameter is a variable that we list as a part of a function definition, whereas an argument is a value that we pass to the function when we invoke it.

- A function's parameter list and its argument list can be different lengths:
  - Unassigned parameters evaluate as undefined.
  - Extra arguments aren't bound to parameter names.
- Rest parameters and default parameters are new additions to JavaScript:
  - Rest parameters enable us to reference the remaining arguments that don't have matching parameter names.
  - Default parameters enable us to specify default parameter values that will be used if no value is supplied during function invocation.

3.6   *Exercises*

1 In the following code snippet, which functions are callback functions?

```
numbers.sort(function sortAsc(a,b){
  return a - b;
});

function ninja(){}
ninja();

var myButton = document.getElementById("myButton");
myButton.addEventListener("click", function handleClick(){
  alert("Clicked");
});
```

2 In the following snippet, categorize functions according to their type (function declaration, function expression, or arrow function).

```
numbers.sort(function sortAsc(a,b){
  return a - b;
});

numbers.sort((a,b) => b - a);

(function(){})();

function outer(){
  function inner(){}
  return inner;
}

(function(){}());

(()=>"Yoshi")();
```

3 After executing the following code snippet, what are the values of variables samurai and ninja?

```
var samurai = (() => "Tomoe")();
var ninja = (() => {"Yoshi"})();
```

4 Within the body of the `test` function, what are the values of parameters `a`, `b`, and `c` for the two function calls?

```
function test(a, b, ...c){ /*a, b, c*/}

test(1, 2, 3, 4, 5);
test();
```

5 After executing the following code snippet, what are the values of the `message1` and `message2` variables?

```
function getNinjaWieldingWeapon(ninja, weapon = "katana"){
  return ninja + " " + katana;
}

var message1 = getNinjaWieldingWeapon("Yoshi");
var message2 = getNinjaWieldingWeapon("Yoshi", "wakizashi");
```

Functions for the journeyman: understanding function invocation

This chapter covers

- Two implicit function parameters: arguments and this
- Ways of invoking functions
- Dealing with problems of function contexts

In the previous chapter, you saw that JavaScript is a programming language with significant functionally oriented characteristics. We explored the differences between function call arguments and function parameters, and how the values are transferred from call arguments to function parameters.

This chapter continues in a similar vein, by first discussing something that we kept from you in the previous chapter: the implicit function parameters `this` and `arguments`. These are silently passed to functions and can be accessed just like any other explicitly named function parameter within the function's body.

The `this` parameter represents the function context, the object on which our function is invoked, whereas the `arguments` parameter represents all arguments that are passed in through a function call. Both parameters are vital in JavaScript code. The `this` parameter is one of the fundamental ingredients of object-oriented JavaScript, and the `arguments` parameter allows us to be creative with the arguments that are accepted by our functions. For this reason, we'll explore some of the common pitfalls related to these implicit arguments.

We'll then continue by exploring ways of invoking functions in JavaScript. The way in which we invoke a function has a great influence on how the implicit function parameters are determined.

Finally, we'll conclude the chapter by learning about common gotchas related to the function context, the `this` parameter. Without further ado, let's start exploring!

. . . .

Do you know?	**Why is the `this` parameter known as the function *context*?** **What's the difference between a function and a method?** **What would happen if a constructor function explicitly returned an object?**

. . . .

4.1   Using implicit function parameters

In the preceding chapter, we explored the differences between function *parameters* (variables listed as part of a function definition) and function *arguments* (values passed to the function when we invoke it). But we didn't mention that in addition to the parameters that we've explicitly stated in the function definition, function invocations are usually passed two implicit parameters: `arguments` and `this`.

By *implicit*, we mean that these parameters aren't explicitly listed in the function signature, but are silently passed to the function and accessible within the function. They can be referenced within the function just like any other explicitly named parameter. Let's take a look at each of these implicit parameters in turn.

4.1.1   The arguments parameter

The `arguments` parameter is a collection of all arguments passed to a function. It's useful because it allows us to access all function arguments, regardless of whether the matching parameter is explicitly defined. This allows us to implement function over-loading, a feature that JavaScript doesn't natively support, and variadic functions that accept a variable number of arguments. To be honest, with `rest` parameters, intro-duced in the preceding chapter, the need for the `arguments` parameter has been greatly reduced. Still, it's important to understand how the `arguments` parameter works, because you're bound to run into it when dealing with legacy code.

The `arguments` object has a property named `length` that indicates the exact num-
ber of arguments. The individual argument values can be obtained by using array
indexing notation; for example, `arguments[2]` would fetch the third parameter. Take
a look at the following listing.

Listing 4.1   Using the `arguments` parameter

```
function whatever(a, b, c){                        ◁──────   Declares a function with three
                                                            parameters: a, b, and c

  assert(a === 1, 'The value of a is 1');
  assert(b === 2, 'The value of b is 2');          Tests for correct values
  assert(c === 3, 'The value of c is 3');

  assert(arguments.length === 5,                   In all, the function is
    'We've passed in 5 parameters');               passed five arguments.

  assert(arguments[0] === a,
    'The first argument is assigned to a');        Checks that the first
  assert(arguments[1] === b,                       three arguments match
    'The second argument is assigned to b');       the function parameters
  assert(arguments[2] === c,
    'The third argument is assigned to c');

  assert(arguments[3] === 4,                       Checks that the excess arguments
    'We can access the fourth argument');          can be accessed through the
  assert(arguments[4] === 5,                       arguments parameter
    'We can access the fifth argument');
}

whatever(1,2,3,4,5);            ◁────────   Calls a function with five arguments
```

Here we have a `whatever` function that gets called with five arguments, `whatever`
`(1,2,3,4,5)`, even though it has only three declared parameters, `a`, `b`, `c`:

```
function whatever(a, b, c){
  ...
}
```

We can access the first three arguments through their respective function parameters,
`a`, `b`, and `c`:

```
assert(a === 1, 'The value of a is 1');
assert(b === 2, 'The value of b is 2');
assert(c === 3, 'The value of c is 3');
```

We can also check how many arguments in total were passed to the function by using
the `arguments.length` property.

The `arguments` parameter can also be used to access each individual argument
through array notation. It's important to note that this also includes the excess argu-
ments that aren't associated with any function parameters:

```
assert(arguments[0] === a, 'The first argument is assigned to a');
assert(arguments[1] === b, 'The second argument is assigned to b');
assert(arguments[2] === c, 'The third argument is assigned to c');
assert(arguments[3] === 4, 'We can access the fourth argument');
assert(arguments[4] === 5, 'We can access the fifth argument');
```

Throughout this section, we go out of our way to avoid calling the `arguments` parameter an *array*. You may be fooled into thinking that it's an array; after all, it has a `length` parameter and its entries can be fetched using array notation. But it's *not* a JavaScript array, and if you try to use array methods on `arguments` (for example, the `sort` method used in the previous chapter), you'll find nothing but heartbreak and disappointment. Just think of `arguments` as an *array-like* construct, and exhibit restraint in its use.

As we've already mentioned, the main point of the `arguments` object is to allow us to access all arguments that were passed to the function, regardless of whether a particular argument is associated with a function parameter. Let's see how to do this by implementing a function that can calculate the sum of an arbitrary number of arguments.

Listing 4.2   Using the `arguments` object to perform operations on all function arguments

```
function sum() {        ◁——— A function without any explicitly defined parameters
  var sum = 0;
  for(var i = 0; i < arguments.length; i++){      Iterates through all arguments
    sum += arguments[i];                          passed, and accesses individual
  }                                               items through index notation
  return sum;
}

assert(sum(1, 2) === 3, "We can add two numbers");        Calls the function
assert(sum(1, 2, 3) === 6, "We can add three numbers");   with any number
assert(sum(1, 2, 3, 4) === 10, "We can add four numbers"); of arguments
```

Here we first define a `sum` function that doesn't explicitly list any parameters. Regardless of this, we can still access all function arguments through the `arguments` object. We iterate through all the arguments and calculate their sum.

Now comes the payoff. We can call the function with any number of arguments, so we test a couple of cases to see if everything works. This is the true power of the `arguments` object. It allows us to write more versatile and flexible functions that can easily deal with different situations.

> **NOTE**   We mentioned earlier that in a lot of cases we can use the `rest` parameter instead of the `arguments` parameter. The `rest` parameter is a real array, which means that we can use all our favorite array methods on it. This gives it a certain advantage over the `arguments` object. As an exercise, rewrite listing 4.2 to use the `rest` parameter instead of the `arguments` parameter.

Now that we understand how the `arguments` object works, let's explore some of its gotchas.

ARGUMENTS OBJECT AS AN ALIAS TO FUNCTION PARAMETERS

The `arguments` parameter has one curious feature: It aliases function parameters. If we set a new value to, for example, `arguments[0]`, the value of the first parameter will also be changed. Take a look at the following listing.

Listing 4.3   The `arguments` object aliases function parameters

```
function infiltrate(person) {
  assert(person === 'gardener',
    'The person is a gardener');       The person parameter has
  assert(arguments[0] === 'gardener',  the value "gardener" sent
    'The first argument is a gardener'); as a first argument.

  arguments[0] = 'ninja';

                                       Changing the arguments
  assert(person === 'ninja',           object will also change
    'The person is a ninja now');      the matching parameter.
  assert(arguments[0] === 'ninja',
    'The first argument is a ninja');

  person = 'gardener';

                                       The alias works
  assert(person === 'gardener',        both ways.
    'The person is a gardener once more');
  assert(arguments[0] === 'gardener',
    'The first argument is a gardener again');
}

infiltrate("gardener");
```

You can see how the `arguments` object is an alias for the function parameters. We define a function, `infiltrate`, that takes a single parameter, `person`, and we invoke it with the argument `gardener`. We can access the value `gardener` through the function parameter `person` and through the `arguments` object:

```
assert(person === 'gardener', 'The person is a gardener');
assert(arguments[0] === 'gardener', 'The first argument is a gardener');
```

Because the `arguments` object is an alias for the function parameters, if we change the `arguments` object, the change is also reflected in the matching function parameter:

```
arguments[0] = 'ninja';

assert(person === 'ninja', 'The person is a ninja now');
assert(arguments[0] === 'ninja', 'The first argument is a ninja');
```

The same holds true in the other direction. If we change a parameter, the change can be observed in both the parameter and the `arguments` object:

```
person = 'gardener';

assert(person === 'gardener',
    'The person is a gardener once more');
assert(arguments[0] === 'gardener',
    'The first argument is a gardener again');
```

AVOIDING ALIASES

The concept of aliasing function parameters through the `arguments` object can be confusing, so JavaScript provides a way to opt out of it by using *strict mode.*

> ### Strict mode
> Strict mode is an ES5 addition to JavaScript that changes the behavior of JavaScript engines so that errors are thrown instead of silently picked up. The behavior of some language features is changed, and some unsafe language features are even completely banned (more on this later). One of the things that strict mode changes is that it disables `arguments` aliasing.

As always, let's take a look at a simple example.

Listing 4.4   Using strict mode to avoid `arguments` aliasing

```
"use strict";                    ◁——— Enables strict mode

function infiltrate(person){
  assert(person === 'gardener',
    'The person is a gardener');        The person argument and
  assert(arguments[0] === 'gardener',   the first argument start
    'The first argument is a gardener'); with the same value.

  arguments[0] = 'ninja';        ◁——— Changes the first argument

  assert(arguments[0] === 'ninja',      The first argument
    'The first argument is now a ninja'); is changed.

  assert(person === 'gardener',         The value of the person
    'The person is still a gardener');  parameter hasn't changed.
}

infiltrate("gardener");
```

Here we start by placing the simple string `use strict` as the first line of code. This tells the JavaScript engine that we want to execute the following code in strict mode. In this example, strict mode changes the semantics of our program in a way that the `person` parameter and the first `argument` start with the same value:

```
assert(person === 'gardener', 'The person is a gardener');
assert(arguments[0] === 'gardener', 'The first argument is a gardener');
```

But, unlike in nonstrict mode, this time around the `arguments` object doesn't alias the parameters. If we change the value of the first argument, `arguments[0] = 'ninja'`, the first argument is changed, but the `person` parameter isn't:

```
assert(arguments[0] === 'ninja', 'The first argument is now a ninja');
assert(person === 'gardener', 'The person is still a gardener');
```

We'll revisit the `arguments` object later in this book, but for now, let's focus on another implicit parameter: `this`, which is in some ways even more interesting.

4.1.2 The this parameter: introducing the function context

When a function is invoked, in addition to the parameters that represent the explicit arguments provided in the function call, an implicit parameter named `this` is passed to the function. The `this` parameter, a vital ingredient in object-oriented JavaScript, refers to an object that's associated with the function invocation. For this reason, it's often termed the *function context.*

The function context is a notion that those coming from object-oriented languages such as Java might think that they understand. In such languages, `this` usually points to an instance of the class within which the method is defined.

But beware! As we'll soon see, in JavaScript, invoking a function as a *method* is only one way that a function can be invoked. And as it turns out, what the `this` parameter points to isn't (as in Java or C#) defined only by how and where the function is defined; it can also be heavily influenced by how the function is *invoked.* Because understanding the exact nature of the `this` parameter is one of the most important pillars of object-oriented JavaScript, we're about to look at various ways of invoking functions. You'll see that one of the primary differences between them is how the value of `this` is determined. And then we'll take a long and hard look at function contexts again in several following chapters, so don't worry if things don't gel right away.

Now let's see, in great detail, how functions can be invoked.

4.2 Invoking functions

We've all called JavaScript functions, but have you ever stopped to wonder what really happens when a function is called? As it turns out, the manner in which a function is invoked has a huge impact on how the code within it operates, primarily in how the `this` parameter, the function context, is established. This difference is much more important than it might seem at first. We'll examine it within this section and exploit it throughout the rest of this book to help elevate our code to the ninja level.

We can invoke a function in four ways, each with its own nuances:

- *As a function*—`skulk()`, in which the function is invoked in a straightforward manner
- *As a method*—`ninja.skulk()`, which ties the invocation to an object, enabling object-oriented programming

- *As a constructor*—`new Ninja()`, in which a new object is brought into being
- *Via the function's* `apply` *or* `call` *methods*—`skulk.call(ninja)` or `skulk.apply(ninja)`

Here are examples:

```
function skulk(name) {}
function Ninja(name) {}

skulk('Hattori');
(function(who){ return who; })('Hattori');
```
Invoked as a function

```
var ninja = {
  skulk: function(){}
};

ninja.skulk('Hattori');
```
⟵ **Invoked as a method of ninja**

```
ninja = new Ninja('Hattori');
```
⟵ **Invoked as a constructor**

```
skulk.call(ninja, 'Hattori');
```
⟵ **Invoked via the call method**

```
skulk.apply(ninja, ['Hattori']);
```
⟵ **Invoked via the apply method**

For all but the `call` and `apply` approaches, the function invocation operator is a set of parentheses following any expression that evaluates to a function reference.

Let's start our exploration with the simplest form, invoking functions as functions.

4.2.1   *Invocation as a function*

Invocation as a function? Well, of course functions are invoked as *functions*. How silly to think otherwise. But in reality, we say that a function is invoked "as a function" to distinguish it from the other invocation mechanisms: methods, constructors, and `apply`/`call`. If a function isn't invoked as a method, as a constructor, or via `apply` or `call`, it's invoked as a function.

This type of invocation occurs when a function is invoked using the `()` operator, and the expression to which the `()` operator is applied doesn't reference the function as a property of an object. (In that case, we'd have a method invocation, but we discuss that next.) Here are some simple examples:

```
                     function ninja(){};        Function declaration
                     ninja();                   invoked as a function

Immediately invoked
function expression,   var samurai = function(){};     Function expression
invoked as a function  samurai();                      invoked as a function
                  ⟶  (function(){})()
```

When invoked in this manner, the function context (the value of the `this` keyword) can be two things: In nonstrict mode, it will be the global context (the `window` object), whereas in strict mode, it will be `undefined`.

The following listing illustrates the difference in behavior between strict and non-strict modes.

Listing 4.5   Invocation as a function

```
function ninja() {
  return this;
}
```
A function in nonstrict mode

```
function samurai() {
  "use strict";
  return this;
}
```
A function in strict mode

```
assert(ninja() === window,
  "In a 'nonstrict' ninja function, " +
  "the context is the global window object");
```
As expected, a nonstrict function has window as the function context.

```
assert(samurai() === undefined,
  "In a 'strict' samurai function, " +
  "the context is undefined");
```
The strict function, on the other hand, has an undefined context.

> **NOTE** As you can see, strict mode is, in most cases, much more straightforward than nonstrict mode. For example, when listing 4.5 invokes a function as a function (as opposed to as a method), it hasn't specified an object on which the function should be invoked. So, in our opinion, it makes more sense that the `this` keyword should be set to `undefined` (as in strict mode), as opposed to the global `window` object (as in nonstrict mode). In general, strict mode fixes a lot of these small JavaScript oddities. (Remember arguments aliasing from the beginning of the chapter?)

You've likely written code such as this many times without giving it much thought. Now let's step it up a notch by looking at how functions are invoked as *methods*.

4.2.2   *Invocation as a method*

When a function is assigned to a property of an object *and* the invocation occurs by referencing the function using that property, then the function is invoked as a *method* of that object. Here's an example:

```
var ninja = {};
ninja.skulk = function(){};
ninja.skulk();
```

Okay; so what? The function is called a *method* in this case, but what makes that interesting or useful? Well, if you come from an object-oriented background, you'll remember that the object to which a method belongs is available within the body of the method as `this`. The same thing happens here. When we invoke a function as a *method* of an object, that object becomes the function context and is available within

the function via the `this` parameter. This is one of the primary means by which JavaScript allows object-oriented code to be written. (Constructors are another, and we'll get to them in short order.)

Let's consider some test code in the next listing to illustrate the differences and similarities between invocation as a function and invocation as a method.

Listing 4.6   The differences between function and method invocations

getMyThis gets a reference to the whatsMyContext function.

```
function whatsMyContext() {
  return this;
}
```
Returns the function context that will allow us to examine the context from outside

```
assert(whatsMyContext() === window,
  "Function call on window");
```
Invoking as a function sets the context to the window object.

A ninja1 object is created with a getMyThis property that references the whatsMyContext function.

```
var getMyThis = whatsMyContext;

assert(getMyThis() === window,
  "Another function call in window");

var ninja1 = {
  getMyThis: whatsMyContext
};
```
Invokes the function using the getMyThis variable. Even though we use a variable, the function is still invoked as a function, and the function context is the window object.

```
assert(ninja1.getMyThis() === ninja1,
  "Working with 1st ninja");

var ninja2 = {
  getMyThis: whatsMyContext
};
```
Invoking the functions through getMyThis calls it as a method of ninja1. The function context is now ninja1. That's object orientation!

Another object, ninja2, also has a getMyThis property referencing whatsMyContext.

```
assert(ninja2.getMyThis() === ninja2,
  "Working with 2nd ninja");
```
Invoking the function as a method of ninja2 shows that the function context is now ninja2.

This test sets up a function named `whatsMyContext` that we'll use throughout the rest of the listing. The only thing that this function does is to return its function context so that we can see, from outside the function, what the function context for the invocation is. (Otherwise, we'd have a hard time knowing.)

```
function whatsMyContext() {
  return this;
}
```

When we call the function directly by name, this is a case of invoking the function as a function, so we expect that the function context will be the global context (`window`), because we're in nonstrict mode. We assert that this is so:

```
assert(whatsMyContext() === window, ...)
```

Then we create a reference to the function `whatsMyContext` in a variable named `get-MyThis: var getMyThis = whatsMyContext`. This doesn't create a second instance of the function; it merely creates a reference to the same function (you know, first-class object and all).

When we invoke the function via the variable—something we can do because the function invocation operator can be applied to any expression that evaluates to a function—we're once again invoking the function as a function. As such, we again expect that the function context is `window`, and it is:

```
assert(getMyThis() === window,
       "Another function call in window");
```

Now, we get a bit trickier and define an object in variable `ninja1` with a property named `getMyThis` that receives a reference to the `whatsMyContext` function. By doing so, we say that we've created a *method* named `getMyThis` on the object. We don't say that `whats-MyContext` has *become* a method of `ninja1`; it hasn't. You've already seen that `whatsMy-Context` is its own independent function that can be invoked in numerous ways:

```
var ninja1 = {
  getMyThis: whatsMyContext
};
```

According to what we stated earlier, when we invoke the function via a method reference, we expect the function context to be the method's object (in this case, `ninja1`) and we assert as much:

```
assert(ninja1.getMyThis() === ninja1,
       "Working with 1st ninja");
```

> **NOTE** Invoking functions as methods is crucial to writing JavaScript in an object-oriented manner. Doing so enables you to use `this` within any method to reference the method's "owning" object—a fundamental concept in object-oriented programming.

To drive that point home, we continue testing by creating yet another object, `ninja2`, also with a property named `getMyThis` that references the `whatsMyContext` function. Upon invoking this function through the `ninja2` object, we correctly assert that its function context is `ninja2`:

```
var ninja2 = {
  getMyThis: whatsMyContext
};

assert(ninja2.getMyThis() === ninja2,
  "Working with 2nd ninja");
```

Even though the *same* function—`whatsMyContext`—is used throughout the example, the function context returned by `this` changes depending on how `whatsMyContext` is invoked. For example, the exact same function is shared by both `ninja1` and `ninja2`,

yet when it's executed, the function has access to, and can perform operations on, the object through which the method was invoked. We don't need to create separate copies of a function to perform the exact same processing on different objects. This is a tenet of object-oriented programming.

Though a powerful capability, the manner in which it's used in this example has limitations. Foremost, when we create the two ninja objects, we're able to share the same function to be used as a method in each, but we have to use a bit of repeated code to set up the separate objects and their methods.

But that's nothing to despair over—JavaScript provides mechanisms to make creating objects from a single pattern much easier than in this example. We'll explore those capabilities in depth in chapter 7. But for now, let's consider a part of that mechanism that relates to function invocations: the *constructor.*

4.2.3   *Invocation as a constructor*

There's nothing special about a function that's going to be used as a constructor. *Constructor functions* are declared just like any other functions, and we can easily use function declarations and function expressions for constructing new objects. The only exception is the arrow function, which as you'll see later in the chapter, works a bit differently. But, in any case, the main difference is in how the function is invoked.

To invoke the function as a constructor, we precede the function invocation with the keyword `new`. For example, recall the `whatsMyContext` function from the previous section:

```
function whatsMyContext(){ return this; }
```

If we want to invoke the `whatsMyContext` function as a constructor, we write this:

```
new whatsMyContext();
```

But even though we can invoke `whatsMyContext` as a constructor, that function isn't a particularly useful constructor. Let's find out why by discussing what makes constructors special.

> **NOTE**   Remember in chapter 3, when we discussed ways of defining functions? Among function declarations, function expressions, arrow functions, and generator functions, we also mentioned *function constructors,* which enable us to construct new functions from strings. For example: `new Function('a', 'b', 'return a + b')` creates a new function with two parameters, a and b, that returns their sum. Be careful not to confuse these *function constructors* with *constructor functions*! The difference is subtle, yet significant. A function constructor enables us to create functions from dynamically created strings. On the other hand, *constructor functions,* the topic of this section, are functions that we use to create and initialize object instances.

THE SUPERPOWERS OF CONSTRUCTORS

Invoking a function as a constructor is a powerful feature of JavaScript that we'll explore in the following listing.

Listing 4.7   Using a constructor to set up common objects

Creates two objects by invoking the constructor with new. The newly created objects are referenced by ninja1 and ninja2.

```
function Ninja() {
  this.skulk = function() {
    return this;
  };
}

var ninja1 = new Ninja();
var ninja2 = new Ninja();

assert(ninja1.skulk() === ninja1,
  "The 1st ninja is skulking");
assert(ninja2.skulk() === ninja2,
  "The 2nd ninja is skulking");
```

A constructor that creates a skulk property on whatever object is the function context. The method once again returns the function context so that we can test it externally.

Tests the skulk method of the constructed objects. Each should return its own constructed object.

In this example, we create a function named `Ninja` that we'll use to construct, well, ninjas. When invoked with the keyword `new`, an empty object instance is created and passed to the function as its function context, the `this` parameter. The constructor creates a property named `skulk` on this object, which is assigned a function, making that function a method of the newly created object.

In general, when a constructor is invoked, a couple of special actions take place, as shown in figure 4.1. Calling a function with the keyword `new` triggers the following steps:

1 A new empty object is created.

2 This object is passed to the constructor as the `this` parameter, and thus becomes the constructor's function context.

3 The newly constructed object is returned as the `new` operator's value (with an exception that we'll get to in short order).
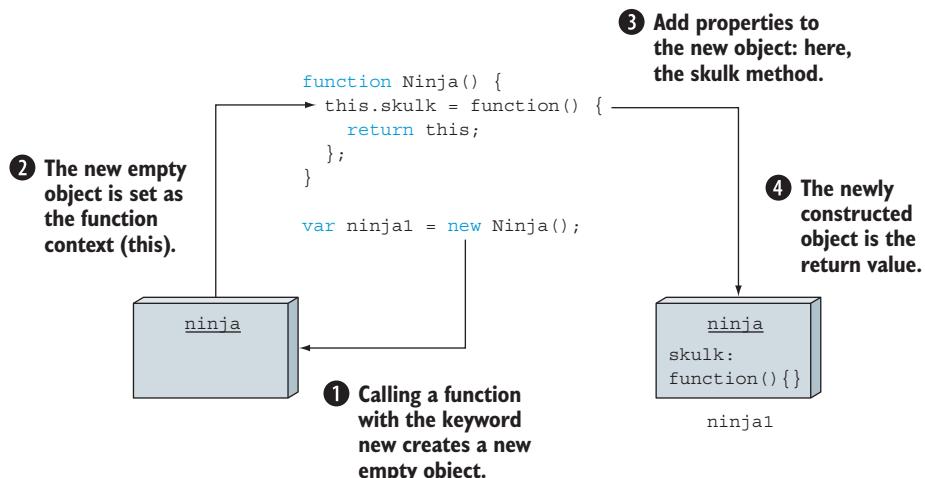
Figure 4.1   When calling a function with a keyword `new`, a new empty object is created and set as the context of the constructor function, the `this` parameter.

The last two points touch on why whatsMyContext in new whatsMyContext() makes for a lousy constructor. The purpose of a constructor is to cause a new object to be created, to set it up, and to return it as the constructor value. Anything that interferes with that intent isn't appropriate for constructors.

Let's consider a more appropriate constructor, Ninja, that sets up skulking ninjas, as shown in listing 4.7:

```
function Ninja() {
  this.skulk = function() {
    return this;
  };
}
```

The skulk method performs the same operation as whatsMyContext in the previous sections, returning the function context so that we can test it externally.

With the constructor defined, we create two new Ninja objects by invoking the constructor twice. Note that the returned values from the invocations are stored in variables that become references to the newly created Ninjas:

```
var ninja1 = new Ninja();
var ninja2 = new Ninja();
```

Then we run the tests that ensure that each invocation of the method operates on the expected object:

```
assert(ninja1.skulk() === ninja1,
  "The 1st ninja is skulking");
assert(ninja2.skulk() === ninja2,
  "The 2nd ninja is skulking");
```

That's it! Now you know how to create and initialize new objects with constructor functions. Calling a function with the keyword new returns the newly created object. But let's check whether that's always exactly true.

CONSTRUCTOR RETURN VALUES

We mentioned earlier that constructors are intended to initialize newly created objects, and that the newly constructed object is returned as a result of a constructor invocation (via the new operator). But what happens when the constructor returns a value of its own? Let's explore that situation in the following listing.

Listing 4.8   Constructors returning primitive values

```
function Ninja() {                    ⟵——— Defines a constructor function named Ninja
  this.skulk = function () {
    return true;
  };

  return 1;                           ⟵——— The constructor returns a specific
                                           primitive value, the number 1.
```

```
          }
          assert(Ninja() === 1,
            "Return value honored when not called as a constructor");

          var ninja = new Ninja();

          assert(typeof ninja === "object",
            "Object returned when called as a constructor");
          assert(typeof ninja.skulk === "function",
            "ninja object has a skulk method");
```

The function is called as a function and its return value is l, as expected.

The function is called as a constructor via the new operator.

Tests verify that the return value of l is ignored, and that a new, initialized object has been returned from new.

If we run this listing, we'll see that all is fine and well. The fact that this `Ninja` function returns a simple number 1 has no significant influence on how the code behaves. If we call the `Ninja` function as a function, it returns 1 (just as we'd expect); and if we call it as a constructor, with the keyword `new`, a new `ninja` object is constructed and returned. So far, so good.

But now let's try something different, a constructor function that returns another object, as shown in the following listing.

Listing 4.9   Constructors explicitly returning object values

```
var puppet = {
  rules: false
};

function Emperor() {
  this.rules = true;
  return puppet;
}

var emperor = new Emperor();

assert(emperor === puppet,
  "The emperor is merely a puppet!");
assert(emperor.rules === false,
  "The puppet does not know how to rule!");
```

Creates our own object with a known property

Returns that object despite initializing the object passed as this

Invokes the function as a constructor

Tests show that the object returned by the constructor is assigned to the variable emperor (and not the object created by the new expression).

This listing takes a slightly different approach. We start by creating a `puppet` object with the property `rules` set to `false`:

```
var puppet = {
  rules: false
};
```

Then we define an `Emperor` function that adds a `rules` property to the newly constructed object and sets it to `true`. In addition, the `Emperor` function has one quirk; it returns the global `puppet` object:

```
function Emperor() {
  this.rules = true;
  return puppet;
}
```

Later, we call the `Emperor` function as a constructor, with the keyword `new`:

```
var emperor = new Emperor();
```

With this, we've set up an ambiguous situation: We get one object passed to the constructor as the function context in `this`, which we initialize, but then we return a completely different `puppet` object. Which object will reign supreme?

   Let's test it:

```
assert(emperor === puppet, "The emperor is merely a puppet!");
assert(emperor.rules === false,
       "The puppet does not know how to rule!");
```

It turns out that our tests indicate that the `puppet` object is returned as the value of constructor invocation, and that the initialization that we performed on the function context in the constructor was all for naught. The `puppet` has been exposed!

   Now that we've gone through some tests, let's summarize our findings:

- If the constructor returns an object, that object is returned as the value of the whole `new` expression, and the newly constructed object passed as `this` to the constructor is discarded.
- If, however, a nonobject is returned from the constructor, the returned value is ignored, and the newly created object is returned.

Because of these peculiarities, functions intended for use as constructors are generally coded differently from other functions. Let's explore that in greater detail.

CODING CONSIDERATIONS FOR CONSTRUCTORS

The intent of constructors is to initialize the new object that will be created by the function invocation to initial conditions. And although such functions *can* be called as "normal" functions, or even assigned to object properties in order to be invoked as methods, they're generally not useful as such. For example

```
function Ninja() {
    this.skulk = function() {
      return this;
    };
}
var whatever = Ninja();
```

We can call `Ninja` as a simple function, but the `skulk` property would be created on `window` in nonstrict mode—not a particularly useful operation. Things go even more awry in strict mode, as `this` would be undefined and our JavaScript application would crash. But this is a good thing; if we make this mistake in nonstrict mode, it might

escape our notice (unless we had good tests), but there's no missing the mistake in strict mode. This is a good example of why strict mode was introduced.

Because constructors are generally coded and used in a manner that's different from other functions, and aren't all that useful unless invoked as constructors, a naming convention has arisen to distinguish constructors from run-of-the-mill functions and methods. If you've been paying attention, you may have already noticed it.

Functions and methods are generally named starting with a verb that describes what they do (`skulk`, `creep`, `sneak`, `doSomethingWonderful`, and so on) and start with a lowercase letter. Constructors, on the other hand, are usually named as a noun that describes the object that's being constructed and start with an uppercase character: `Ninja`, `Samurai`, `Emperor`, `Ronin`, and so on.

It's easy to see how a constructor makes it more elegant to create multiple objects that conform to the same pattern without having to repeat the same code over and over. The common code is written just once, as the body of the constructor. In chapter 7, you'll see more about using constructors and about the other object-oriented mechanisms that JavaScript provides to make it even easier to set up object patterns.

But we're not finished with function invocations yet. There's still another way that JavaScript lets us invoke functions that provides a great deal of control over the invocation details.

4.2.4 Invocation with the apply and call methods

So far, you've seen that one of the major differences between the types of function invocation is what object ends up as the function context referenced by the implicit `this` parameter that's passed to the executing function. For methods, it's the method's owning object; for top-level functions, it's either `window` or `undefined` (depending on the current strictness); for constructors, it's a newly created object instance.

But what if we want to make the function context whatever we want? What if we want to set it explicitly? What if...well, why would we want to do such a thing?

To get a glimpse of why we'd care about this ability, we'll look at a practical example that illustrates a surprisingly common bug related to event handling. For now, consider that when an event handler is called, the function context is set to the object to which the event was bound. (Don't worry if this seems vague; you'll learn about event handling in great detail in chapter 13.) Take a look at the following listing.

Listing 4.10   Binding a specific context to a function

A constructor function that creates objects that retain state regarding our button. With it, we'll track whether the button has been clicked.

A button element to which we'll assign an event handler

Declares the method that we'll use as the click handler. Because it's a method of the object, we use this within the function to get a reference to the object.

```
<button id="test">Click Me!</button>
<script>
  function Button(){
    this.clicked = false;
    this.click = function(){
```

Creates an instance that will track whether the button was clicked

Establishes the click handler on the button

```
          this.clicked = true;
          assert(button.clicked, "The button has been clicked");     ◁
        };
      }
→   var button = new Button();
      var elem = document.getElementById("test");
      elem.addEventListener("click", button.click);
    </script>
```

Within the method, we test that the button state has been correctly changed after a click.

In this example, we have a button, `<button id="test">Click Me!</button>`, and we want to know whether it has ever been clicked. To retain that state information, we use a constructor function to create a backing object named `button`, in which we'll store the clicked state:

```
function Button(){
  this.clicked = false;
  this.click = function(){
    this.clicked = true;
    assert(button.clicked, "The button has been clicked");
  };
}
var button = new Button();
```

In that object, we also define a `click` method that will serve as an event handler that fires when the button is clicked. The method sets the clicked property to `true` and then tests that the state was properly recorded in the backing object (we've intentionally used the `button` identifier instead of the `this` keyword—after all, they should refer to the same thing, or should they?). Finally, we establish the `button.click` method as a click handler for the button:

```
var elem = document.getElementById("test");
elem.addEventListener("click", button.click);
```

When we load the example into a browser and click the button, we see by the display of figure 4.2 that something is amiss; the stricken text indicates that the test has failed. The code in listing 4.10 fails because the context of the `click` function *isn't* referring to the `button` object as we intended.

Recalling the lessons of earlier in the chapter, if we had called the function via `button.click()`, the context *would* have been the button, because the function would be invoked as a method on the
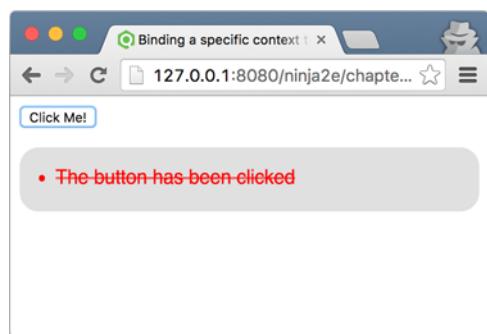
Figure 4.2   Why did our test fail? Where did the change of state go? Usually, the event callback's context is the object raising the event (in this case, the HTML button, and not the button object).

button object. But in this example, the event-handling system of the browser defines the context of the invocation to be the target element of the event, which causes the context to be the <button> element, not the button object. So we set our click state on the wrong object!

This is a surprisingly common problem, and later in the chapter, you'll see techniques for completely evading it. For now, let's explore how to tackle it by examining how to explicitly set the function context by using the apply and call methods.

USING THE APPLY AND CALL METHODS

JavaScript provides a means for us to invoke a function and to explicitly specify any object we want as the function context. We do this through the use of one of two methods that exist for every function: apply and call.

Yes, we said methods of functions. As first-class objects (created, by the way, by the built-in Function constructor), functions can have properties just like any other object type, including methods.

To invoke a function by using its apply method, we pass two parameters to apply: the object to be used as the function context, and an array of values to be used as the invocation arguments. The call method is used in a similar manner, except that the arguments are passed directly in the argument list rather than as an array.

The following listing shows both of these methods in action.

Listing 4.11 Using the `apply` and `call` methods to supply the function context

```
function juggle() {
  var result = 0;
  for (var n = 0; n < arguments.length; n++) {
    result += arguments[n];
  }
  this.result = result;
}

var ninja1 = {};
var ninja2 = {};

juggle.apply(ninja1,[1,2,3,4]);
juggle.call(ninja2, 5,6,7,8);

assert(ninja1.result === 10, "juggled via apply");
assert(ninja2.result === 26, "juggled via call");
```

The function "juggles" the arguments and puts the result onto whatever object is the function context.

These objects are initially empty and will serve as our test subjects.

Uses the apply method, passing ninja1 and an array of arguments

Uses the call method, passing ninja2 and a list of arguments

The tests show how the juggle result is placed on the objects passed to the methods.

In this example, we set up a function named juggle, in which we define juggling as adding up all the arguments and storing them as a property named result on the function context (referenced by the this keyword). That may be a rather lame definition of juggling, but it *will* allow us to determine whether arguments were passed to the function correctly, and which object ended up as the function context.

We then set up two objects, `ninja1` and `ninja2`, that we'll use as function contexts, passing the first to the function's `apply` method, along with an *array* of arguments, and passing the second to the function's `call` method, along with a *list* of other arguments:

```
juggle.apply(ninja1,[1,2,3,4]);
juggle.call(ninja2, 5,6,7,8);
```

Notice that the only difference between `apply` and `call` is how the arguments are supplied. In the case of `apply`, we use an array of arguments, and in the case of `call`, we list them as call arguments, after the function context. See figure 4.3.

The object that will serve as the function context

The second argument to apply is an array of items that will be used as function arguments.

```
juggle.apply(ninja1, [1,2,3,4]);
```

```
juggle.call(ninja2, 5,6,7,8);
```

A sequence of arguments that will be used as function arguments
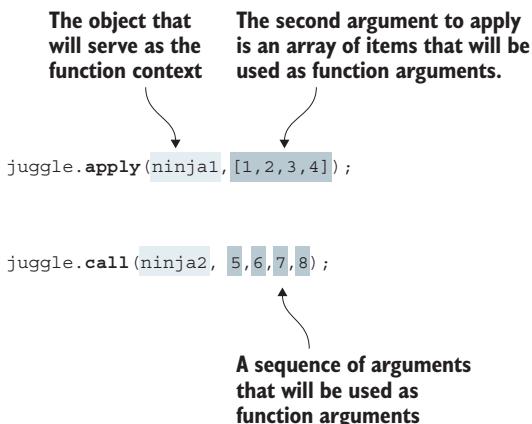
Figure 4.3   **As the first argument, both the `call` and `apply` methods take the object that will be used as the function context. The difference is in the following arguments. `apply` takes only one additional argument, an array of argument values; `call` takes any number of arguments, which will be used as function arguments.**

After we've supplied our function contexts and arguments, we continue by testing! First, we check that `ninja1`, which was called via `apply`, received a `result` property that's the result of adding up all the argument values (`1`, `2`, `3`, `4`) in the passed array. Then we do the same for `ninja2`, which was called via `call`, where we check the result for arguments 5, 6, 7, and 8:

```
assert(ninja1.result === 10, "juggled via apply");
assert(ninja2.result === 26, "juggled via call");
```

Figure 4.4 provides a closer look at what's going on in listing 4.11.

These two methods, `call` and `apply`, can come in handy whenever it's expedient to usurp what would normally be the function context with an object of our own choosing—something that can be particularly useful when invoking callback functions.

FORCING THE FUNCTION CONTEXT IN CALLBACKS

Let's consider a concrete example of forcing the function context to be an object of our own choosing. We'll use a simple function to perform an operation on every entry of an array.

```
function juggle() {
  var result = 0;
  for (var n = 0; n < arguments.length; n++){
    result += arguments[n];
  }
  this.result = result;
}

var ninja1 = {};
var ninja2 = {};
```
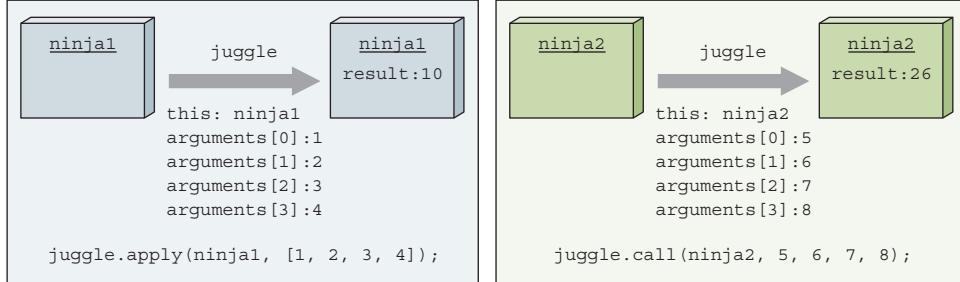

Figure 4.4 **Manually setting a function context by using built-in `call` and `apply` from listing 4.11 results in these combinations of function contexts (the `this` parameter) and `arguments`.**

In imperative programming, it's common to pass the array to a method and use a `for` loop to iterate over every entry, performing the operation on each entry:

```
function(collection) {
  for (var n = 0; n < collection.length; n++) {
    /* do something to collection[n] */
  }
}
```

In contrast, the functional approach is to create a function that operates on a single element and passes each entry to that function:

```
function(item){
  /* do something to item */
}
```

The difference lies in thinking at a level where functions are the main building blocks of the program. You might think that it's moot, and that all you're doing is moving the `for` loop out one level, but we're not done massaging this example yet.

To facilitate a more functional style, all array objects have access to a `forEach` function that invokes a callback on each element within an array. This is often more succinct, and this style is preferred over the traditional `for` statement by those familiar with functional programming. Its organizational benefits will become even more evident (*cough*, code reuse, *cough*) after covering closures in chapter 5. Such an iteration

function *could* pass the current element to the callback as a parameter, but most make the current element the function context of the callback.

Even though all modern JavaScript engines now support a forEach method on arrays, we'll build our own (simplified) version of such a function in the next listing.

Listing 4.12   Building a `forEach` function to demonstrate setting a function context

The callback is invoked such that the current iteration item is the function context.

```
function forEach(list, callback) {
  for (var n = 0; n < list.length; n++) {
    callback.call(list[n], n);
  }
}

var weapons = [ { type: 'shuriken' },
                { type: 'katana' },
                { type:'nunchucks' }];

forEach(weapons, function(index){
    assert(this === weapons[index],
          "Got the expected value of " + weapons[index].type);
});
```

Our iteration function accepts the collection to be iterated over and a callback function.

Calls the iteration function and ensures that the function context is correct for each invocation of the callback

Our test subject

The iteration function sports a simple signature that expects the array of objects to be iterated over as the first argument, and a callback function as the second. The function iterates over the array entries, invoking the callback function for each entry:

```
function forEach(list,callback) {
  for (var n = 0; n < list.length; n++) {
    callback.call(list[n], n);
  }
}
```

We use the call method of the callback function, passing the current iteration entry as the first parameter and the loop index as the second. This *should* cause the current entry to become the function context, and the index to be passed as the single parameter to the callback.

Now to test that! We set up a simple weapons array. Then we call the forEach function, passing the test array and a callback within which we test that the expected entry is set as the function context for each invocation of the callback:

```
forEach(weapons, function(index){
  assert(this === weapons[index],
        "Got the expected value of " + weapons[index].type);
});
```

Figure 4.5 shows that our function works splendidly.

In a production-ready implementation of such a function, there'd be a lot more work to do. For example, what if the first argument isn't an array? What if the second
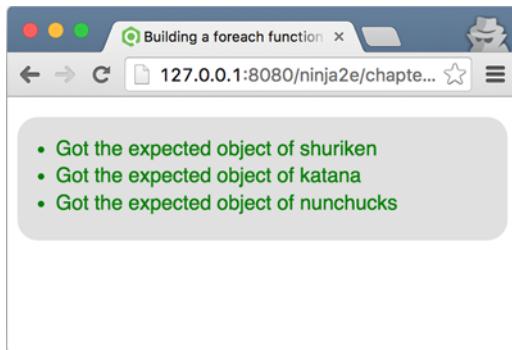
argument isn't a function? How would you allow the page author to terminate the loop at any point? As an exercise, you can augment the function to handle these situations. Another exercise you could task yourself with is to enhance the function so that the page author can pass an arbitrary number of arguments to the callback in addition to the iteration index.

Given that `apply` and `call` do pretty much the same thing, here's something you might be asking yourself at this point: How do we decide which to use? The high-level answer is the same as for many such questions: We use whichever one improves code clarity. A more practical answer is to use the one that best matches the arguments we have handy. If we have a bunch of unrelated values in variables or specified as literals, `call` lets us list them directly in its argument list. But if we already have the argument values in an array, or if it's convenient to collect them as such, `apply` could be the better choice.

4.3   *Fixing the problem of function contexts*

In the preceding section, you saw some of the problems that can happen when dealing with function context in JavaScript. In callback functions (such as event handlers), the function context might not be exactly what we expect, but we can use the `call` and `apply` methods to get around it. In this section, you'll see two other options: arrow functions and the `bind` method, which can, in certain cases, achieve the same effect, but in a much more elegant way.

4.3.1   *Using arrow functions to get around function contexts*

Besides allowing us to create functions in a more elegant way than standard function declarations and function expressions, the arrow functions introduced in the previous chapter have one feature that makes them particularly good as callback functions: Arrow functions don't have their own `this` value. Instead, they remember the value of the `this` parameter at the time of their definition. Let's revisit our problem with button-click callbacks in the following listing.

Listing 4.13   Using arrow functions to work around callback function contexts

A button element to which we'll assign an event handler ──▷

A constructor function that creates objects that retain state regarding our button. With it, we'll track whether the button has been clicked.

Declares the arrow function that we'll use as the click handler. Because it's a method of the object, we use this within the function to get a reference to the object.

```
<button id="test">Click Me!</button>
<script>
  function Button(){
    this.clicked = false;
    this.click = () => {
      this.clicked = true;
      assert(button.clicked,"The button has been clicked");//
    };
  }
  var button = new Button();
  var elem = document.getElementById("test");
  elem.addEventListener("click", button.click);
</script>
```

Within the method, we test that the button state has been correctly changed after a click.

Establishes the click handler on the button

The only change, when compared to listing 4.10, is that listing 4.13 uses an arrow function:

```
this.click = () = > {
  this.clicked = true;
  assert(button.clicked, "The button has been clicked");
};
```

Now, if we run the code, we'll get the output shown in figure 4.6.

As you can see, all is well now. The button object keeps track of the clicked state. What happened is that our click handler was created inside the Button constructor as an arrow function:

```
function Button(){
    this.clicked = false;
    this.click = () => {
      this.clicked = true;
      assert(button.clicked, "The button has been clicked");
    };
}
```
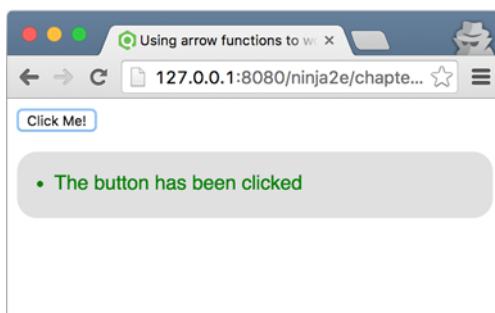

Figure 4.6   Arrow functions don't have their own context. Instead, the context is inherited from the function in which they're defined. The `this` parameter in our arrow function callback refers to the button object.

As we already mentioned, arrow functions don't get their own implicit `this` parameter when we call them; instead they remember the value of the `this` parameter at the time they were created. In our case, the `click` arrow function was created inside a constructor function, where the `this` parameter is the newly constructed object, so whenever we (or the browser) call the `click` function, the value of the `this` parameter will always be bound to the newly constructed button object.

CAVEAT: ARROW FUNCTIONS AND OBJECT LITERALS

Because the value of the `this` parameter is picked up at the moment that the arrow function is created, some seemingly strange behaviors can result. Let's go back to our button-click handler example. Let's say we've come to the conclusion that we don't need a constructor function, because we have only one button. We replace it with a simple object literal, in the following way.

Listing 4.14   Arrow functions and object literals

The button object is defined as an object literal.

The value of the this parameter in global code is the global window object.

```
<button id="test">Click Me!</button>
<script>
  assert(this === window, "this === window");
  var button = {
    clicked: false,
    click: () => {
      this.clicked = true;
      assert(button.clicked,"The button has been clicked");
      assert(this === window, "In arrow function this === window");
      assert(window.clicked, "clicked is stored in window");
    }
  }

  var elem = document.getElementById("test");
  elem.addEventListener("click", button.click);
</script>
```

Our arrow function is a property of an object literal.

Test whether the button was clicked.

clicked is stored on window.

The value of this in our arrow function is the global window object.

If we run listing 4.14, we'll again be disappointed, because the `button` object has once more failed to track the `clicked` state. See figure 4.7.
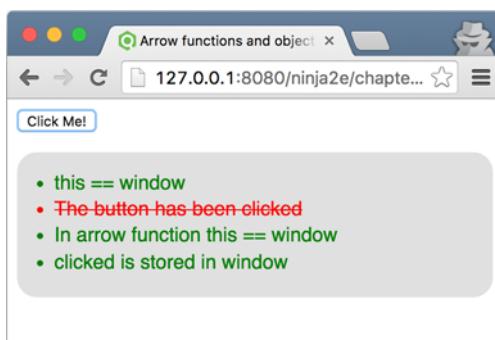
Figure 4.7   If an arrow function is defined within an object literal that's defined in global code, the value of the `this` parameter associated with the arrow function is the global `window` object.

Luckily, we've scattered a couple of assertions throughout our code that will help. For example, we've placed the following directly in global code, in order to check the value of the `this` parameter:

```
assert(this === window, "this === window");
```

Because the assertion passes, we can be sure that in global code `this` refers to the global `window` object.

We follow this by specifying that the `button` object literal has a `click` arrow function property:

```
var button = {
  clicked: false,
  click: () => {
    this.clicked = true;
    assert(button.clicked,"The button has been clicked");
    assert(this === window, "In arrow function this === window");
    assert(window.clicked, "Clicked is stored in window");
  };
}
```

Now, we'll again revisit our little rule: *Arrow functions pick up the value of the* `this` *parameter at the moment of their creation.* Because the `click` arrow function is created as a property value on an object literal, and the object literal is created in global code, the `this` value of the arrow function will be the `this` value of the global code. And, as we've seen from the first assertion placed in our global code

```
assert(this === window, "this === window");
```

the value of the `this` parameter in global code is the global `window` object. Therefore, our `clicked` property will be defined on the global `window` object, and not on our `button` object. Just to be sure, in the end, we check that the `window` object has been assigned a `clicked` property:

```
assert(window.clicked, "Clicked is stored in window");
```

As you can see, failing to keep in mind all the consequences of arrow functions can lead to some subtle bugs, so be careful!

Now that we've explored how arrow functions can be used to circumvent the problem of function contexts, let's continue with another method for fixing the same problem.

4.3.2   *Using the bind method*

In this chapter, you've already met two methods that every function has access to, `call` and `apply`, and you've seen how to use them for greater control over the context and arguments of our function invocations.

In addition to these methods, every function has access to the `bind` method that, in short, creates a new function. This function has the same body, but its context is *always* bound to a certain object, *regardless* of the way we invoke it.

Let's revisit our little problem with button-click handlers one last time.

Listing 4.15   Binding a specific context to an event handler

```
<button id="test">Click Me!</button>
<script>
  var button = {
    clicked: false,
    click: function(){
      this.clicked = true;
      assert(button.clicked,"The button has been clicked");
    }
  };
  var elem = document.getElementById("test");
  elem.addEventListener("click", button.click.bind(button));

  var boundFunction = button.click.bind(button);
  assert(boundFunction != button.click,
         "Calling bind creates a completly new function");
</script>
```

> Uses the bind function to create a new function bound to the button object

The secret sauce added here is the `bind()` method:

```
elem.addEventListener("click", button.click.bind(button));
```

The `bind` method is available to all functions, and is designed to *create* and return a *new* function that's bound to the passed-in object (in this case, the `button` object). The value of the `this` parameter is always set to that object, regardless of the way the bound function was invoked. Apart from that, the bound function behaves like the originating function, because it has the same code in its body.

Whenever the button is clicked, that bound function will be invoked with the `button` object as its context, because we've used that `button` object as an argument to `bind`.

Note that calling the `bind` method doesn't modify the original function. It creates a completely new function, a fact asserted at the end of the example:

```
var boundFunction = button.click.bind(button);
assert(boundFunction != button.click,
           "Calling bind creates a completly new function");
```

With this, we'll end our exploration of the function context. Rest for now, because in the next chapter, we'll be dealing with one of the most important concepts in JavaScript: closures.

4.4   Summary

- When invoking a function, in addition to the parameters explicitly stated in the function definition, function invocations are passed in two implicit parameters: `arguments` and `this`:
  - The `arguments` parameter is a collection of arguments passed to the function. It has a `length` property that indicates how many arguments were passed in, and it enables us to access the values of arguments that don't have matching parameters. In nonstrict mode, the `arguments` object aliases the function parameters (changing the argument changes the value of the parameter, and vice versa). This can be avoided by using strict mode.
  - The `this` parameter represents the function context, an object to which the function invocation is associated. How `this` is determined can depend on the way a function is defined as well as on how it's invoked.
- A function can be invoked in four ways:
  - As a function: `skulk()`
  - As a method: `ninja.skulk()`
  - As a constructor: `new Ninja()`
  - Via its `apply` and `call` methods: `skulk.call(ninja)` or `skulk.apply(ninja)`
- The way a function is invoked influences the value of the `this` parameter:
  - If a function is invoked as a function, the value of the `this` parameter is usually the global `window` object in nonstrict mode, and `undefined` in strict mode.
  - If a function is invoked as a method, the value of the `this` parameter is usually the object on which the function was invoked.
  - If a function is invoked as a constructor, the value of the `this` parameter is the newly constructed object.
  - If a function is invoked through `call` and `apply`, the value of the `this` parameter is the first argument supplied to `call` and `apply`.
- Arrow functions don't have their own value of the `this` parameter. Instead, they pick it up at the moment of their creation.
- Use the `bind` method, available to all functions, to create a new function that's always bound to the argument of the `bind` method. In all other aspects, the bound function behaves as the original function.

4.5   Exercises

1 The following function calculates the sum of the passed-in arguments by using the `arguments` object:

```
function sum(){
  var sum = 0;
  for(var i = 0; i < arguments.length; i++){
    sum += arguments[i];
  }
```

```
  return sum;
}

assert(sum(1, 2, 3) === 6, 'Sum of first three numbers is 6');
assert(sum(1, 2, 3, 4) === 10, 'Sum of first four numbers is 10');
```

By using the rest parameters introduced in the previous chapter, rewrite the sum function so that it doesn't use the arguments object.

2 After running the following code, what are the values of variables ninja and samurai?

```
function getSamurai(samurai){
  "use strict"

  arguments[0] = "Ishida";

  return samurai;
}

function getNinja(ninja){
  arguments[0] = "Fuma";
  return ninja;
}

var samurai = getSamurai("Toyotomi");
var ninja = getNinja("Yoshi");
```

3 When running the following code, which of the assertions will pass?

```
function whoAmI1(){
  "use strict";
  return this;
}

function whoAmI2(){
  return this;
}

assert(whoAmI1() === window, "Window?");
assert(whoAmI2() === window, "Window?");
```

4 When running the following code, which of the assertions will pass?

```
var ninja1 = {
  whoAmI: function(){
    return this;
  }
};

var ninja2 = {
  whoAmI: ninja1.whoAmI
};
```

```
var identify = ninja2.whoAmI;

assert(ninja1.whoAmI() === ninja1, "ninja1?");
assert(ninja2.whoAmI() === ninja1, " ninja1 again?");

assert(identify() === ninja1, "ninja1 again?");

assert(ninja1.whoAmI.call(ninja2) === ninja2, "ninja2 here?");
```

5 When running the following code, which of the assertions will pass?

```
function Ninja(){
  this.whoAmI = () => this;
}

var ninja1 = new Ninja();
var ninja2 = {
  whoAmI: ninja1.whoAmI
};


assert(ninja1.whoAmI() === ninja1, "ninja1 here?");
assert(ninja2.whoAmI() === ninja2, "ninja2 here?");
```

6 Which of the following assertions will pass?

```
function Ninja(){
  this.whoAmI = function(){
    return this;
  }.bind(this);
}

var ninja1 = new Ninja();
var ninja2 = {
  whoAmI: ninja1.whoAmI
};


assert(ninja1.whoAmI() === ninja1, "ninja1 here?");
assert(ninja2.whoAmI() === ninja2, "ninja2 here?");
```

Functions for the master: 5
closures and scopes

This chapter covers

- Using closures to simplify development
- Tracking the execution of JavaScript programs with execution contexts
- Tracking variable scopes with lexical environments
- Understanding types of variables
- Exploring how closures work

Closely tied to the functions we learned about in previous chapters, closures are a defining feature of JavaScript. Although scores of JavaScript developers can write code without understanding the benefits of closures, their use can not only help us reduce the amount and complexity of code needed to add advanced features, but also enable us to do things that otherwise wouldn't be possible, or would be too complex to be feasible. For example, any tasks involving callbacks, such as event handling or animations, would be significantly more complex without closures. Others, such as providing support for private object variables, would be outright

impossible. The landscape of the language and the way we write our code is forever shaped by the inclusion of closures.

Traditionally, closures have been a feature of purely functional programming languages. Seeing them cross over into mainstream development is encouraging. It's common to find closures permeating JavaScript libraries, along with other advanced code bases, because of their ability to drastically simplify complex operations.

Closures are a side effect of how scopes work in JavaScript. For this reason, we'll explore the scoping rules of JavaScript, with a special focus on recent additions. This will help you understand how closures work behind the scenes. Let's jump right in!

. . . .

Do you know?

How many different scopes can a variable or method have, and what are they?

How are identifiers and their values tracked?

What is a mutable variable, and how do you define one in JavaScript?

. . . .

5.1   Understanding closures

A *closure* allows a function to access and manipulate variables that are external to that function. Closures allow a function to access all the variables, as well as other functions, that are in scope when the function itself is defined.

> **NOTE**   You're probably familiar with the concept of scopes, but just in case, a *scope* refers to the visibility of identifiers in certain parts of a program. A scope is a part of the program in which a certain name is bound to a certain variable.

That may seem intuitive until you remember that a declared function can be called at any later time, even *after* the scope in which it was declared has gone away. This concept is probably best explained through code. But before we get into concrete examples that will help you develop more elegant animations in code or to define private object properties, let's start small, with the following listing.

Listing 5.1   A simple closure

```
var outerValue = "ninja";                                  Defines a value in global scope
function outerFunction(){
  assert(outerValue === "ninja","I can see the ninja.");    Declares a function
}                                                            in global scope
outerFunction();                Executes the function
```

In this code example, we declare a variable `outerValue` and a function `outerFunction` in the same scope—in this case, the global scope. Afterward, we call `outerFunction`.

As you can see in figure 5.1, the function is able to "see" and access the `outerValue` variable. You've likely written code such as this hundreds of times without realizing that you were creating a closure!

Not impressed? Guess that's not surprising. Because both `outerValue` and `outerFunction` are declared in global scope, that scope (which is a closure) never goes away (as long as our application is running). It's not surprising that the function can access the variable, because it's still in scope and viable.

Figure 5.1 **Our function has found the ninja, who was hiding in plain sight.**

Even though the closure exists, its benefits aren't yet clear. Let's spice it up in the next listing.

Listing 5.2   Another closure example

```
var outerValue = "samurai";
var later;

function outerFunction(){
  var innerValue = "ninja";

  function innerFunction(){
    assert(outerValue === "samurai", "I can see the samurai.");
    assert(innerValue === "ninja", "I can see the ninja.")
  }

  later = innerFunction;
}
outerFunction();

later();
```

Declares a value inside the function. This variable's scope is limited to the function and can't be accessed from outside the function.

An empty variable that we'll use later

Declares an inner function within the outerFunction. innerValue is in scope when we create this function.

Invokes outerFunction, which causes innerFunction to be created and its reference assigned to later

Stores a reference to innerFunction in the later variable. Because later is in the global scope, it'll allow us to call the function later.

Invokes innerFunction through later. We can't invoke it directly because its scope (along with innerValue) is limited to within outerFunction.

Let's overanalyze the code in `innerFunction` and see whether we can predict what might happen:

- The first `assert` is certain to pass; `outerValue` is in the global scope and is visible to everything. But what about the second `assert`?
- We're executing `innerFunction` *after* `outerFunction` has been executed via the trick of copying a reference to the function to the global variable `later`.

- When `innerFunction` executes, the scope inside the outer function is long gone and not visible at the point at which we're invoking the function through `later`.
- So we could very well expect `assert` to fail, as `innerValue` is sure to be undefined. Right?

But when we run the test, we see the display in figure 5.2.

How can that be? What magic allows the `innerValue` variable to still be "alive" when we execute the inner function, long after the scope in which it was created has gone away? The answer is closures.

When we declare `innerFunction` inside the outer function, not only is the function declaration defined, but a closure is created that encompasses the function definition as well as all variables in scope *at the point of function definition.* When `innerFunction` even-

Figure 5.2   Despite trying to hide inside a function, the ninja has been detected!

tually executes, even if it's executed *after* the scope in which it was declared goes away, it has access to the original scope in which it was declared through its closure, as shown in figure 5.3.

That's what closures are all about. They create a "safety bubble" of the function and the variables in scope at the point of the function's definition, so that the function has all it needs to execute. This bubble, containing the function and its variables, stays around as long as the function does.

Although all this structure isn't readily visible (there's no "closure" object holding all of this information that you can inspect), storing and referencing information in this way has a direct cost. It's important to remember that each function that accesses information via a closure has a "ball and chain" attached to it, carrying this information around. So although closures are incredibly useful, they aren't free of overhead. All that information needs to be held in memory until it's

Figure 5.3   Like a protective bubble, the closure for `innerFunction` keeps the variables in the function's scope alive for as long as the function exists.

absolutely clear to the JavaScript engine that it's no longer needed (and is safe to garbage-collect), or until the page unloads.

Don't worry; this isn't all that we have to say about how closures work. But before exploring the mechanisms that enable closures, let's look at their practical uses.

5.2   Putting closures to work

Now that we have a high-level understanding of closures, let's see how to put them to work in our JavaScript applications. For now, we'll focus on their practical aspects and benefits. Later in the chapter, we'll revisit the same examples to see exactly what's going on behind the scenes.

5.2.1   Mimicking private variables

Many programming languages use *private variables*—properties of an object that are hidden from outside parties. This is a useful feature, because we don't want to over-burden the users of our objects with unnecessary implementation details when access-ing those objects from other parts of the code. Unfortunately, JavaScript doesn't have native support for private variables. But by using a closure, we can achieve an accept-able approximation, as demonstrated by the following code.

Listing 5.3   Using closures to approximate private variables

Defines the constructor for a Ninja

Declares a variable inside the constructor function. Because the scope of the variable is limited to inside the constructor, it's a "private" variable. We'll use it to count how many times the ninja has feinted.

Creates an accessor method for the feints counter. Because the variable isn't accessible to code outside the constructor, this is a common way to give read-only access to the value.

Now for testing—first we construct an instance of Ninja.

```
function Ninja() {
  var feints = 0;
  this.getFeints = function(){
    return feints;
  };
  this.feint = function(){
    feints++;
  };
}

var ninja1 = new Ninja();
ninja1.feint();

assert(ninja1.feints === undefined,
       "And the private data is inaccessible to us.");
assert(ninja1.getFeints()=== 1,
       "We're able to access the internal feint count.");

var ninja2 = new Ninja();
assert(ninja2.getFeints()=== 0,
       "The second ninja object gets its own feints variable.");
```

Declares the increment method for the value. Because the value is private, no one can screw it up behind our backs; they're limited to the access that we give them via methods.

Verifies that we can't get at the variable directly

Calls the feint method, which increments the count of the number of times that our ninja has feinted

When we create a new ninja2 object with the Ninja constructor, the ninja2 object gets its own feints variable.

We were able to change the "private" variable, even though we had no direct access to it.

Here we create a function, `Ninja`, to serve as a constructor. We introduced using a function as a constructor in chapter 3 (and we'll take an in-depth look in chapter 7). For now, recall that when using the `new` keyword on a function, a new object instance is created, and the function is called with that new object as its context, to serve as a constructor to that object. So `this` within the function refers to a newly instantiated object.

Within the constructor, we define a variable to hold state, `feints`. The JavaScript scoping rules for this variable limit its accessibility to *within* the constructor. To give access to the value of the variable from code that's outside the scope, we define an *accessor* method: `getFeints`, which can be used to read the private variable. (Accessor methods are frequently called *getters*.)

```
function Ninja() {
  var feints = 0;
  this.getFeints = function(){
    return feints;
  };
  this.feint = function(){
    feints++;
  };
}
```

An implementation method, `feint`, is then created to give us control over the value of the variable. In a real-world application, this might be a business method, but in this example, it merely increments the value of `feints`.

After the constructor has done its duty, we can call the `feint` method on the newly created `ninja1` object:

```
var ninja1 = new Ninja();
ninja1.feint();
```

Our tests show that we can use the accessor method to obtain the value of the private variable but that we can't access it directly. This prevents us from being able to make uncontrolled changes to the value of the variable, just as if it were a true private variable. This situation is depicted in figure 5.4.

Using closures allows the state of the ninja to be maintained within a method, without letting it be directly accessed by a user of the method—because the variable is available to the inner methods via their closures, but not to code that lies outside the constructor.

This is a glimpse into the world of object-oriented JavaScript, which we'll explore in greater depth in chapter 7. For now, let's focus on another common use of closures.

5.2.2   *Using closures with callbacks*

Another common use of closures occurs when dealing with callbacks—when a function is called at an unspecified later time. Often, within such functions, we frequently

In the outer scope, we have no access to feints.

The object instance is visible via the ninja variable.

The feints method has access to the feints variable, as it's within the closure.

```
var ninja
```

```
var feints
```

```
Ninja instance
feint ─────────────→ function(){}
```
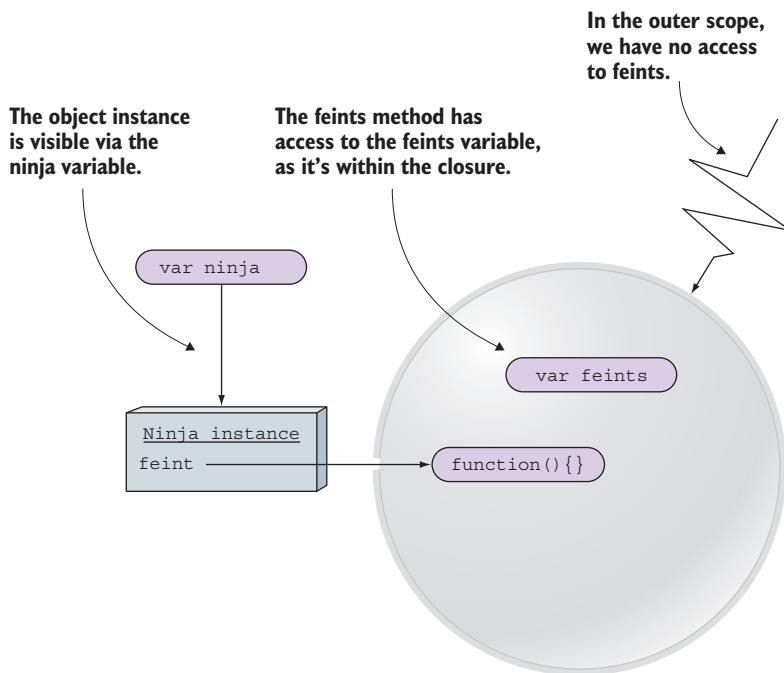
Figure 5.4 Hiding the variable inside the constructor keeps it invisible to the outer scope, but where it counts, the variable is alive and well, protected by the closure.

need to access outside data. The following listing shows an example that creates a simple animation with callback timers.

Listing 5.4 Using a closure in a timer interval callback

Creates the element that we're going to animate

Inside the animateIt function, we get a reference to that element.

Establishes a counter to keep track of animation ticks (steps)

```
<div id="box1">First Box</div>
<script>
  function animateIt(elementId) {
    var elem = document.getElementById(elementId);
    var tick = 0;
    var timer = setInterval(function(){
      if (tick < 100) {
        elem.style.left = elem.style.top = tick + "px";
        tick++;
      }
      else {
        clearInterval(timer);
```

The timer callback is invoked every 10 milliseconds. For 100 ticks, it adjusts the position of the element.

A built-in function that creates and starts an interval timer, given a callback

After 100 ticks, we stop the timer and perform tests to assert that we can see all relevant variables needed to perform the animation.

```
          assert(tick === 100,
                 "Tick accessed via a closure.");
          assert(elem,
                 "Element also accessed via a closure.");
          assert(timer,
                 "Timer reference also obtained via a closure." );
        }
      }, 10);
    }
    animateIt("box1");
  </script>
```

The setInterval duration— the callback will be called every 10ms.

Now that it's all set up, we set it in motion!

What's especially important about this code is that it uses a single anonymous function, placed as a `setInterval` argument, to accomplish the animation of the target `div` element. That function accesses three variables: `elem`, `tick`, and `timer`, via a closure, to control the animation process. The three variables (the reference to the DOM element, `elem`; the tick counter, `tick`; and the timer reference, `timer`) all must be maintained *across* the steps of the animation. And we need to keep them out of the global scope.

But the example will still work fine if we move the variables out of the `animateIt` function and into the global scope. So why all the arm flailing about not polluting the global scope?

Go ahead and move the variables into the global scope and verify that the example still works. Now modify the example to animate two elements: Add another element with a unique ID, and call the `animateIt` function with that ID right after the original call.

The problem immediately becomes obvious. If we keep the variables in the global scope, we need a set of three variables for *each* animation. Otherwise, they'll step all over each other, trying to use the same set of variables to keep track of multiple states.

By defining the variables *inside* the function, and by relying on closures to make them available to the timer callback invocations, each animation gets its own private "bubble" of variables, as shown in figure 5.5.

Without closures, doing multiple things at once, whether event handling, animations, or even server requests, would be incredibly difficult. If you've been waiting for a reason to care about closures, this is it!

This example is a particularly good one for demonstrating how closures are capable of producing some surprisingly intuitive and concise code. By including the variables in the `animateIt` function, we create an implied closure without needing any complex syntax.

There's another important concept that this example makes clear. Not only can we see the values that these variables had at the time the closure was created, but we can update them within the closure while the function within the closure executes. The closure isn't just a snapshot of the state of the scope at the time of creation, but an active encapsulation of that state that we can modify as long as the closure exists.

Each animation gets its own private little bubble of variables within the closures for the handler.

This instance has no access to variables in the other closures.

Neither does this one.
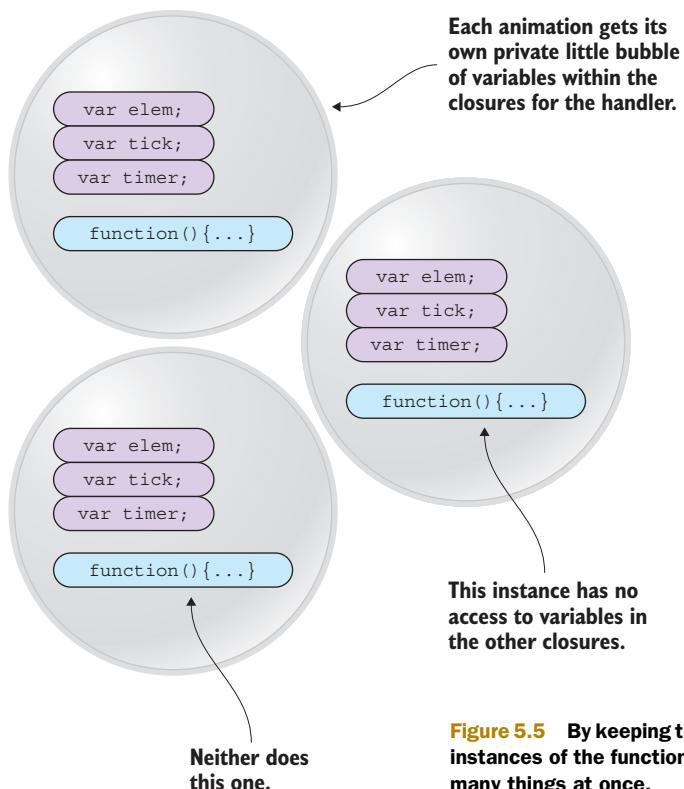
Figure 5.5 By keeping the variables for multiple instances of the function separate, we can do many things at once.

Closures are closely related to scopes, so we'll spend a good deal of this chapter exploring scoping rules in JavaScript. But first, we'll start with the details of how code execution is tracked in JavaScript.

5.3   *Tracking code execution with execution contexts*

In JavaScript, the fundamental unit of execution is a function. We use them all the time, to calculate something, to perform side effects such as changing the UI, to achieve code reuse, or to make our code easier to understand. To fulfill its purpose, a function can call another function, which in turn can call another function, and so on. And when a function does its thing, our program execution has to return to the position from which the function was called. But have you ever wondered how the JavaScript engine keeps track of all these executing functions and return positions?

As we mentioned in chapter 2, there are two main types of JavaScript code: *global code*, placed outside all functions, and *function code*, contained in functions. When our code is being executed by the JavaScript engine, each statement is executed in a certain *execution context*.

And just as we have two types of code, we have two types of execution contexts: a *global execution context* and a *function execution context*. Here's the significant difference:

There's only *one* global execution context, created when our JavaScript program starts executing, whereas a *new* function execution context is created on *each* function invocation.

> **NOTE** You may recall from chapter 4 that *function context* is the object on which our function is invoked, which can be accessed through the `this` keyword. An execution context, although it has a similar name, is a completely different thing. It's an internal JavaScript concept that the JavaScript engine uses to track the execution of our functions.

As we mentioned in chapter 2, JavaScript is based on a single-threaded execution model: Only one piece of code can be executed at a time. Every time a function is invoked, the current execution context has to be stopped, and a new function execution context, in which the function code will be evaluated, has to be created. After the function performs its task, its function execution context is usually discarded, and the caller execution context restored. So there's a need to keep track of all these execution contexts—both the one that's executing and the ones that are patiently waiting. The easiest way to do this is by using a *stack*, called the *execution context stack* (or often called a *call stack*).

> **NOTE** A stack is a fundamental data structure in which you can put new items only to the top and can take existing items only from the top. Think of a stack of trays in a cafeteria. When you want to take one, you pick one from the top. And a cafeteria worker who has a new clean one also puts it on the top.

This might seem vague, so let's look at the following code, which reports the activity of two skulking ninjas.

Listing 5.5   The creation of execution contexts

```
function skulk(ninja) {
  report(ninja + " skulking");     A function that calls
}                                   another function

function report(message) {          A function that reports a message
  console.log(message);             through the built-in console.log
}                                   function

skulk("Kuma");                      Two function calls
skulk("Yoshi");                     from global code
```

This code is straightforward; we define the `skulk` function, which calls the `report` function, which outputs a message. Then, from global code, we make two separate calls to the `skulk` function: `skulk("Kuma")` and `skulk("Yoshi")`. By using this code as a basis, we'll explore the creation of execution contexts, as shown in figure 5.6.

```
function skulk(ninja) {
    report(ninja + " skulking");
}

function report(message) {
    console.log(message);
}

skulk("Kuma");
skulk("Yoshi");
```

1 The program starts executing with the global execution context on the stack.

3 A new function context for the report function is pushed onto the stack when it is called. The skulk execution context is paused.

5 The skulk execution context is popped off of the stack when the skulk function finishes execution. The global execution context resumes execution.

| report ("Kuma skulking") execution context |
| skulk("Kuma") execution context | skulk("Kuma") execution context | skulk("Kuma") execution context |
| Global execution context | Global execution context | Global execution context | Global execution context | Global execution context |

2 When the skulk function is called, a new function context is pushed onto the stack, and the global execution context is paused.

4 When the report function finishes executing, its function context is popped off of the stack. The skulk execution context resumes execution.

Figure 5.6 **The behavior of the execution context stack**

When executing the example code, the execution context behaves as follows:

1 The execution context stack starts with the global execution context that's created only once per JavaScript program (once per page in the case of web pages). The global execution context is the active execution context when executing global code.

2 In global code, the program first defines two functions: `skulk` and `report`, and then calls the `skulk` function with `skulk("Kuma")`. Because only one piece of code can be executed at once, the JavaScript engine pauses the execution of the global code, and goes to execute the `skulk` function code with `Kuma` as an argument. This is done by creating a *new* function execution context and pushing it on top of the stack.

3 The `skulk` function, in turn, calls the `report` function with the argument `Kuma skulking`. Again, because only one piece of code can be executed at a time, the `skulk` execution context is paused, and a new function execution context for the `report` function, with the argument `Kuma skulking`, is created and pushed onto the stack.

4   After the `report` function logs the message by using the built-in `console.log` function (see appendix C) and finishes its execution, we have to go back to the `skulk` function. This is done by popping the `report` function execution context from the stack. The `skulk` function execution context is then reactivated, and the execution of the `skulk` function continues.

5   A similar thing happens when the `skulk` function finishes its execution: The function execution context of the `skulk` function is removed from the stack, and the global execution context, which has been patiently waiting this whole time, is restored as the active execution context. The execution of global JavaScript code is restored.

This whole process is repeated in a similar way for the second call to the `skulk` function, now with the argument `Yoshi`. Two new function execution contexts are created and pushed to the stack, `skulk("Yoshi")` and `report("Yoshi skulking")`, when the respective functions are called. These execution contexts are also popped off the stack when the program returns from the matching function.

Even though the execution context stack is an internal JavaScript concept, you can explore it in any JavaScript debugger, where it's referred to as a *call stack*. Figure 5.7 shows the call stack in Chrome DevTools.

> **NOTE**   Appendix C gives a closer look at the debugging tools available in various browsers.

Besides keeping track of the position in the application execution, the execution context is vital in *identifier resolution*, the process of figuring out which variable a certain identifier refers to. The execution context does this via the *lexical environment*.
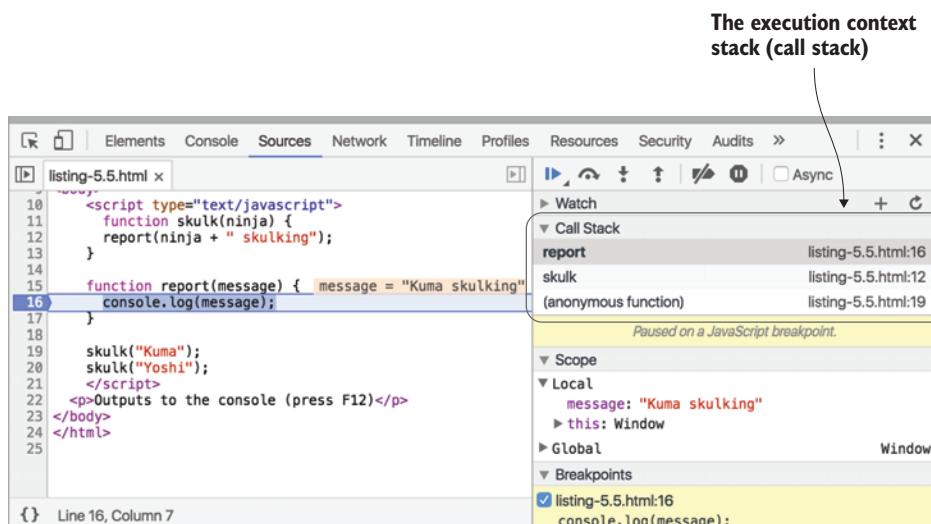

Figure 5.7   The current state of the execution context stack in Chrome DevTools

5.4 Keeping track of identifiers with lexical environments

A *lexical environment* is an internal JavaScript engine construct used to keep track of the mapping from identifiers to specific variables. For example, consider the following code:

```
var ninja = "Hattori";
console.log(ninja);
```

The lexical environment is consulted when the `ninja` variable is accessed in the `console.log` statement.

> **NOTE** Lexical environments are an internal implementation of the JavaScript scoping mechanism, and people often colloquially refer to them as *scopes*.

Usually, a lexical environment is associated with a specific structure of JavaScript code. It can be associated with a function, a block of code, or the `catch` part of a `try-catch` statement. Each of these structures (functions, blocks, and `catch` parts) can have its own separate identifier mappings.

> **NOTE** In pre-ES6 versions of JavaScript, a lexical environment could be associated with only a function. Variables could be only function scoped. This caused a lot of confusion. Because JavaScript is a C-like language, people coming from other C-like languages (such as C++, C#, or Java) naturally expected some low-level concepts, such as the existence of block scopes, to be the same. With ES6, this is finally fixed.

5.4.1 Code nesting

Lexical environments are heavily based on *code nesting*, which enables one code structure to be contained within another. Figure 5.8 shows various types of code nesting.

```
<script>
  var ninja = "Muneyoshi";
  function skulk() {
    var action = "skulking";
    function report() {
      var reportNum = 3;

      for(var i = 0; i < reportNum; i++) {
        console.log(ninja + " " + action + " " + i);
      }
    }
    report();
  }
  skulk();
</script>
```

The skulk function is contained within global code.

The report function is nested within the skulk function.

The for loop is nested within the report function.
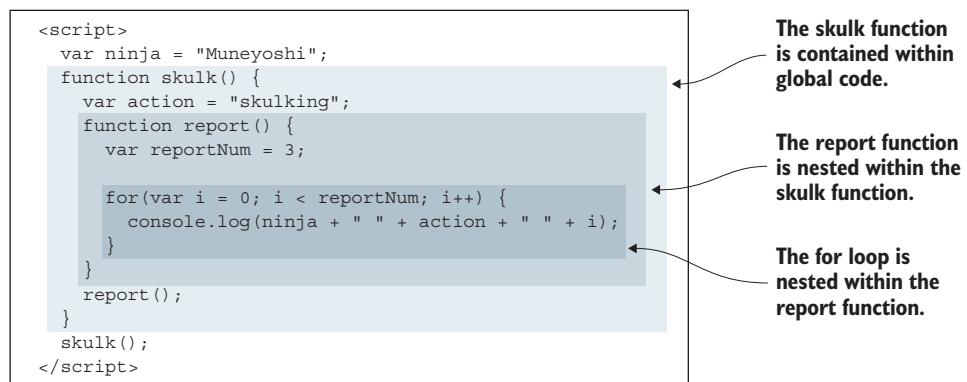
Figure 5.8 Types of code nesting

In this example, we can see the following:

- The `for` loop is nested within the `report` function.
- The `report` function is nested within the `skulk` function.
- The `skulk` function is nested within global code.

In terms of scopes, each of these code structures gets an associated lexical environment *every* time the code is evaluated. For example, on every invocation of the `skulk` function, a new function lexical environment is created.

In addition, it's important to emphasize that an inner code structure has access to the variables defined in outer code structures; for example, the `for` loop can access variables from the `report` function, the `skulk` function, and the global code; the `report` function can access variables from the `skulk` function and the global code; and the `skulk` function can access only additional variables from the global code.

There's nothing special about this way of accessing variables; all of us have probably done it many times. But how does the JavaScript engine keep track of all these variables, and what's accessible from where? This is where lexical environments jump in.

5.4.2   *Code nesting and lexical environments*

In addition to keeping track of local variables, function declarations, and function parameters, each lexical environment has to keep track of its *outer* (parent) lexical environment. This is necessary because we have to be able to access variables defined in outer code structures; if an identifier can't be found in the current environment, the outer environment is searched. This stops either when the matching variable is found, or with a reference error if we've reached the global environment and there's no sign of the searched-for identifier. Figure 5.9 shows an example; you can see how the identifiers `intro`, `action`, and `ninja` are resolved when executing the `report` function.

In this example, the `report` function is called by the `skulk` function, which in turn is called by global code. Each execution context has a lexical environment associated with it that contains the mapping for all identifiers defined directly in that context. For example, the global environment holds the mapping for identifiers `ninja` and `skulk`, the `skulk` environment holds the mapping for the identifiers `action` and `report`, and the `report` environment holds the mapping for the `intro` identifier (the right side of figure 5.9).

In a particular execution context, besides accessing identifiers defined directly in the matching lexical environment, our programs often access other variables defined in outer environments. For example, in the body of the `report` function, we access the variable `action` of the outer `skulk` function, as well as the global `ninja` variable. To do this, we have to somehow keep track of these outer environments. JavaScript does this by taking advantage of functions as first-class objects.

Whenever a function is created, a reference to the lexical environment in which the function was created is stored in an internal (meaning that you can't access or manipulate it directly) property named `[[Environment]]`; double brackets is the notation that we'll use to mark these internal properties. In our case, the `skulk` function will

```
<script>
  var ninja = "Muneyoshi";

  function skulk() {
    var action = "Skulking";

    function report() {
      var intro = "Aha!";
      assert(intro === "Aha!", "Local");
      assert(action === "Skulking", "Outer");
      assert(ninja === "Muneyoshi", "Global");
    }

    report();
  }

  skulk();
</script>
```

Execution stack

report environment

intro: "Aha!"

report execution context

Outer

skulk environment

[[Environment]]

skulk execution context

action: "Skulking"

report: ──────→ function report(){}

Global execution context

Outer

Global environment

[[Environment]]

ninja: "Muneyoshi"

skulk: ──────→ function skulk(){}

Find `intro`

1. Check `report` environment -> found

Find `action`

1. Check `report` environment -> not found
2. Check `report`'s outer environment: `skulk`
   Check `skulk` environment -> found

Find `ninja`

1. Check `report` environment -> not found
2. Check `report`'s outer environment: `skulk`
   Check `skulk` environment -> not found
3. Check `skulk`'s outer environment: `global`
   Check `global` environment -> found

Figure 5.9   How JavaScript engines resolve the values of variables

keep a reference to the global environment, and the `report` function will keep a refer-
ence to the `skulk` environment, because these were the environments in which the
functions were created.

> **NOTE**   This might seem odd at first. Why don't we just traverse the whole
> stack of execution contexts and search their matching environments for iden-
> tifier mappings? Technically, this would work in our example. But remember,
> a JavaScript function can be passed around as any other object, so the posi-
> tion of the function definition and the position from where the function is
> called generally aren't related (remember closures).

Whenever a function is called, a new function execution context is created and pushed onto the execution context stack. In addition, a new associated lexical environment is created. Now comes the crucial part: For the outer environment of the newly created lexical environment, the JavaScript engine puts the environment referenced by the called function's internal `[[Environment]]` property, the environment in which the now-called function was created!

In our case, when the `skulk` function is called, the outer environment of the newly created `skulk` environment becomes the global environment (because it's the environment in which the `skulk` function was created). Similarly, when calling the `report` function, the outer environment of the newly created `report` environment is set to the `skulk` environment.

Now let's take a look at the `report` function:

```
function report() {
  var intro = "Aha!";
  assert(intro === "Aha!", "Local");
  assert(action === "Skulking", "Outer");
  assert(action === "Muneyoshi", "Global");
}
```

When the first `assert` statement is being evaluated, we have to resolve the `intro` identifier. To do this, the JavaScript engine starts by checking the environment of the currently running execution context, the `report` environment. Because the `report` environment contains a reference to `intro`, the identifier is resolved.

Next, the second `assert` statement has to resolve the `action` identifier. Again, the environment of the currently running execution context is checked. But the `report` environment doesn't contain a reference to the `action` identifier, so the JavaScript engine has to check the outer environment of the `report` environment: the `skulk` environment. Luckily, the `skulk` environment contains a reference to the `action` identifier, and the identifier is resolved. A similar process is followed when trying to resolve the `ninja` identifier (a little hint: the identifier can be found in the global environment).

Now that you understand the fundamentals of identifier resolution, let's look at the various ways a variable can be declared.

5.5 Understanding types of JavaScript variables

In JavaScript, we can use three keywords for defining variables: `var`, `let`, and `const`. They differ in two aspects: *mutability* and their relationship toward the lexical environment.

> **NOTE**   The keyword `var` has been part of JavaScript since its beginning, whereas `let` and `const` are ES6 additions. You can check whether your browser supports `let` and `const` at the following links: http://mng.bz/CGJ6 and http://mng.bz/uUIT.

If we were to divide variable declaration keywords by mutability, we'd put `const` on one side and `var` and `let` on the other side. All variables defined with `const` are immutable, meaning that their value can be set only once. On the other hand, variables defined with keywords `var` and `let` are typical run-of-the-mill variables, whose value we can change as many times as necessary.

Now, let's delve into how `const` variables work and behave.

CONST VARIABLES

A `const` "variable" is similar to a normal variable, with the exception that we have to provide an initialization value when it's declared, and we can't assign a completely new value to it afterward. Hmmm, not very *variable*, is it?

Const variables are often used for two slightly different purposes:

- Specifying variables that shouldn't be reassigned (and in the rest of the book, we use them mostly in this regard).
- Referencing a fixed value, for example, the maximum number of ronin in a squad, `MAX_RONIN_COUNT`, by name, instead of using a literal number such as 234. This makes our programs easier to understand and maintain. Our code isn't filled with seemingly arbitrary literals (234), but with meaningful names (`MAX_RONIN_COUNT`) whose values are specified in only one place.

In either case, because `const` variables aren't meant to be reassigned during program execution, we've safeguarded our code against unwanted or accidental modifications and we've even made it possible for the JavaScript engine to do some performance optimizations.

The following listing illustrates the behavior of `const` variables.

Listing 5.6 The behavior of `const` variables

```
const firstConst = "samurai";
assert(firstConst === "samurai", "firstConst is a samurai");
```
Defines a const variable and verifies that the value was assigned

```
try{
  firstConst = "ninja";
  fail("Shouldn't be here");
} catch(e){
  pass("An exception has occurred");
}
```
Attempting to assign a new value to a const variable throws an exception.

```
assert(firstConst === "samurai",
      "firstConst is still a samurai!");

const secondConst = {};
```
Creates a new const variable and assigns a new object to it

```
secondConst.weapon = "wakizashi";
assert(secondConst.weapon === "wakizashi",
      "We can add new properties");
```
We can't assign a completely new object to the secondConst variable, but there's nothing stopping us from modifying the one we already have.

```
const thirdConst = [];
assert(thirdConst.length === 0, "No items in our array");

thirdConst.push("Yoshi");

assert(thirdConst.length === 1, "The array has changed");
```

The exact same thing holds for arrays.

Here we first define a `const` variable named `firstConst` with a value `samurai` and test that the variable has been initialized, as expected:

```
const firstConst = "samurai";
assert(firstConst === "samurai", "firstConst is a samurai");
```

We continue by trying to assign a completely new value, `ninja`, to our `firstConst` variable:

```
try{
  firstConst = "ninja";
  fail("Shouldn't be here");
} catch(e){
  pass("An exception has occurred");
}
```

Because the `firstConst` variable is, well, a constant, we can't assign a new value to it, so the JavaScript engine throws an exception without modifying the variable's value. Notice that we're using two functions that we haven't used so far: `fail` and `pass`. These two methods behave similarly to the `assert` method, except `fail` always fails and `pass` always passes. Here we use them to check whether an exception has occurred: If an exception occurs, the `catch` statement is activated and the `pass` method is executed. If there's no exception, the `fail` method is executed, and we'll be notified that something isn't as it should be. We can check to verify that the exception happens, as shown in figure 5.10.

Next, we define another `const` variable, this time initializing it to an empty object:

```
const secondConst = {};
```

Now we'll discuss an important feature of `const` variables. As you've already seen, we can't assign a completely new value to a `const` variable. But there's nothing stopping us from *modifying* the current one. For example, we can add new properites to the current object:
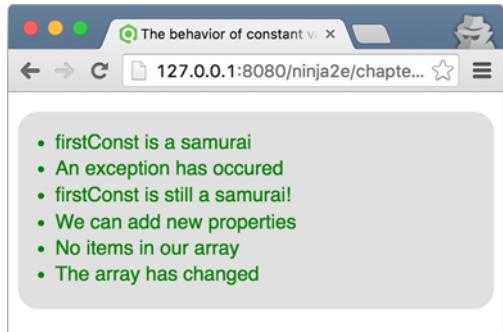

Figure 5.10   Checking the behavior of `const` variables. An exception occurs when we try to assign a completely new value to a `const` variable.

```
secondConst.weapon = "wakizashi";
assert(secondConst.weapon === "wakizashi",
       "We can add new properties");
```

Or, if our `const` variable refers to an array, we can modify that array to any degree:

```
const thirdConst = [];
assert(thirdConst.length === 0, "No items in our array");

thirdConst.push("Yoshi");

assert(thirdConst.length === 1, "The array has changed");
```

And that's about it. `const` variables aren't that complicated to begin with. You only have to remember that a value of a `const` variable can be set only on initialization and that we can't assign a completely new value later. We can still modify the existing value; we just can't completely override it.

Now that we've explored variable mutability, let's consider the details of the relationships between various types of variables and lexical environments.

5.5.2 *Variable definition keywords and lexical environments*

The three types of variable definitions—`var`, `let`, and `const`—can also be categorized by their relationship with the lexical environment (in other words, by their scope). In that case, we can put `var` on one side, and `let` and `const` on the other.

USING THE VAR KEYWORD

When we use the `var` keyword, the variable is defined in the closest function or global lexical environment. (Note that blocks are ignored!) This is a long-standing detail of JavaScript that has tripped up many developers coming from other languages.

Consider the following listing.

Listing 5.7   Using the `var` keyword

```
var globalNinja = "Yoshi";                          ◁── Defines a global
                                                        variable, using var
function reportActivity(){
  var functionActivity = "jumping";        ◁── Defines a function local
                                               variable, using var

  for(var i = 1; i < 3; i++) {                              Defines two
     var forMessage = globalNinja + " " + functionActivity;  variables in the for
     assert(forMessage === "Yoshi jumping",                  loop, using var
            "Yoshi is jumping within the for block");
     assert(i, "Current loop counter:" + i);        Within the for
  }                                                  loop, we can
                                                     access the block
  assert(i === 3 && forMessage === "Yoshi jumping",  variables, function
       "Loop variables accessible outside of the loop");  variables, and
  }                                                  global variables—
                                                     nothing surprising
reportActivity();        But the variables of the for loop are   there.
                         also accessible outside the for loop.
```

```
assert(typeof functionActivity === "undefined"
    && typeof i === "undefined" && typeof forMessage === "undefined",
    "We cannot see function variables outside of a function");
```

**Normally, none of the function variables
are accessible outside of the function.**

We start by defining a global variable, `globalNinja`, which is followed by defining a `reportActivity` function that loops two times and notifies us about the jumping activity of our `globalNinja`. As you can see, within the body of the `for` loop, we can normally access both the block variables (`i` and `forMessage`), the function variables (`functionActivity`), and the global variables (`globalNinja`).

But what's strange with JavaScript, and what confuses a lot of developers coming from other languages, is that we can access the variables defined with code blocks even outside those blocks:

```
assert(i === 3 && forMessage === "Yoshi jumping",
        "Loop variables accessible outside of the loop");
```

This stems from the fact that variables declared with the keyword `var` are always registered in the closest function or global lexical environment, without paying any attention to blocks. Figure 5.11 depicts this situation, by showing the state of lexical environments after the second iteration of the `for` loop in the `reportActivity` function.

```
var globalNinja = "Yoshi";

function reportActivity() {
 var functionActivity = "jumping";

 for(var i = 1; i < 3; i++) {
    var forMessage = globalNinja + " " + functionActivity;
    assert(forMessage = "Yoshi jumping",
           "Yoshi is jumping within the for block");
    assert(i, "Current loop counter:" + i);
 }
   assert(i == 3 && forMessage == "Yoshi jumping",
          "Loop variables accessible outside of the loop");
 }

reportActivity();
 ...
```

Global environment

Function environment

Block environment

The state of lexical environments when finishing the second iteration of the for loop

Figure 5.11 When defining variables with the keyword `var`, a variable is defined in the closest function or global environment (while ignoring block environments). In our case, the variables `forMessage` and `i` are registered in the `reportActivity` environment (the closest functional environment), even though they're contained within a `for` loop.

Here we have three lexical environments:

- The global environment in which the `globalNinja` variable is registered (because this is the closest function or global lexical environment)
- The `reportActivity` environment, created on the `reportActivity` function invocation, which contains the `functionActivity`, `i`, and `forMessage` variables, because they're defined with the keyword `var`, and this is their closest function environment
- The `for` block environment, which is empty, because `var`-defined variables ignore blocks (even when contained within them)

Because this behavior is a bit strange, the ES6 version of JavaScript offers two new variable declaration keywords: `let` and `const`.

USING LET AND CONST TO SPECIFY BLOCK-SCOPED VARIABLES

Unlike `var`, which defines the variable in the closest function or global lexical environment, the `let` and `const` keywords are more straightforward. They define variables in the closest lexical environment (which can be a block environment, a loop environment, a function environment, or even the global environment). We can use `let` and `const` to define block-scoped, function-scoped, and global-scoped variables.

Let's rewrite our previous example to use `const` and `let`.

Listing 5.8   Using `const` and `let` keywords

Defines a global variable, using const. Global const variables are usually written in uppercase.

Defines a function local variable, using const

Within the for loop, we can access the block variables, function variables, and global variables—nothing surprising there.

Defines two variables in the for loop, using let

Now, the variables of the for loop aren't accessible outside the for loop.

```javascript
const GLOBAL_NINJA = "Yoshi";

function reportActivity(){
  const functionActivity = "jumping";

  for(let i = 1; i < 3; i++) {
      let forMessage = GLOBAL_NINJA + " " + functionActivity;
      assert(forMessage === "Yoshi jumping",
              "Yoshi is jumping within the for block");
      assert(i, "Current loop counter:" + i);
  }

  assert(typeof i === "undefined" && typeof forMessage === "undefined",
        "Loop variables not accessible outside the loop");
}

reportActivity();
assert(typeof functionActivity === "undefined"
    && typeof i === "undefined" && typeof forMessage === "undefined",
    "We cannot see function variables outside of a function");
```

Normally, none of the function variables are accessible outside the function.

Figure 5.12 illustrates the current situation, when finishing the execution of the second iteration of the `for` loop in the `reportActivity` function. We again have three lexical environments: the global environment (for global code outside all functions and blocks), the `reportActivity` environment bound to the `reportActivity` function, and the block environment for the `for` loop body. But because we're using `let` and `const` keywords, the variables are defined in their closest lexical environment; the `GLOBAL_NINJA` variable is defined in the global environment, the `functionActivity` variable in the `reportActivity` environment, and the `i` and `forMessage` variables in the `for` block environment.

Now that `const` and `let` have been introduced, scores of new JavaScript developers who have recently come from other programming languages can be at peace. JavaScript

```
const GLOBAL_NINJA = "Yoshi";                                          Global
                                                                       environment
function reportActivity() {
 const functionActivity = "jumping";

 for(let i = 1; i < 3; i++) {
     let forMessage = GLOBAL_NINJA + " " + functionActivity;          Function
     assert(forMessage = "Yoshi jumping",                             environment
            "Yoshi is jumping within the for block");
     assert(i, "Current loop counter:" + i);
 }
  assert(typeof i == "undefined"                                      Block
       && typeof forMessage == "undefined",                          environment
          "Loop variables not accessible outside the loop");
}

reportActivity();
...
```


Figure 5.12 **When defining variables with keywords `let` and `const`, a variable is defined in the closest environment. In our case, variables `forMessage` and `i` are registered in the `for` block environment, the variable `functionActivity` in the `reportActivity` environment, and the `GLOBAL_NINJA` variable in the global environment (in every case, the closest environment to the respective variable).**

finally supports the same scoping rules as other C-like languages. For this reason, from this point in this book, we almost always use const and let instead of var.

Now that we understand how identifier mappings are kept within lexical environments and how lexical environments are linked to program execution, let's discuss the exact process by which identifiers are defined within lexical environments. This will help us better understand some commonly occurring bugs.

5.5.3 Registering identifiers within lexical environments

One of the driving principles behind the design of JavaScript as a programming language was its ease of use. That's one of the main reasons for not specifying function return types, function parameter types, variable types, and so on. And you already know that JavaScript code is executed line by line, in a straightforward fashion. Consider the following:

```
firstRonin = "Kiyokawa";
secondRonin = "Kondo";
```

The value Kiyokawa is assigned to the identifier firstRonin, and then the value Kondo is assigned to the identifier secondRonin. There's nothing weird about that, right? But take a look at another example:

```
const firstRonin = "Kiyokawa";
check(firstRonin);
function check(ronin) {
  assert(ronin === "Kiyokawa", "The ronin was checked! ");
}
```

In this case, we assign the value Kiyokawa to the identifier firstRonin, and then we call the check function with the identifier firstRonin as a parameter. But hold on a second—if the code is executed line by line, should we be able to call the check function? Our program execution hasn't reached its declaration, so the JavaScript engine shouldn't even know about it.

But if we check, as shown in figure 5.13, you see that all is fine and well. JavaScript isn't too picky about where we define our functions. We can choose to place function declarations before or even after their respective calls. This isn't something that the developer should need to fuss about.

Figure 5.13 The function is indeed visible, even before the execution reaches its definition.

THE PROCESS OF REGISTERING IDENTIFIERS

But ease of use aside, if code is executed line by line, how did the JavaScript engine know that a function named

check exists? It turns out that the JavaScript engine "cheats" a little, and that execution of JavaScript code occurs in two phases.

The first phase is activated whenever a new lexical environment is created. In this phase, the code isn't executed, but the JavaScript engine visits and registers all declared variables and functions within the current lexical environment. The second phase, JavaScript execution, starts after this has been accomplished; the exact behavior depends on the type of variable (let, var, const, function declaration) and the type of environment (global, function, or block).

The process is as follows:

1 If we're creating a function environment, the implicit `arguments` identifier is created, along with all formal function parameters and their argument values. If we're dealing with a nonfunction environment, this step is skipped.

2 If we're creating a global or a function environment, the current code is scanned (without going into the body of other functions) for function declarations (but not function expressions or arrow functions!). For each discovered function declaration, a new function is created and bound to an identifier in the environment with the function's name. If that identifier name already exists, its value is overwritten. If we're dealing with block environments, this step is skipped.

3 The current code is scanned for variable declarations. In function and global environments, all variables declared with the keyword var and defined outside other functions (but they can be placed within blocks!) are found, and all variables declared with the keywords let and const defined outside other functions and blocks are found. In block environments, the code is scanned only for variables declared with the keywords let and const, directly in the current block. For each discovered variable, if the identifier doesn't exist in the environment, the identifier is registered and its value initialized to undefined. But if the identifier exists, it's left with its value.

These steps are summarized in figure 5.14.

Now we'll go through the implications of these rules. You'll see some common JavaScript conundrums that can lead to weird bugs that are easy to
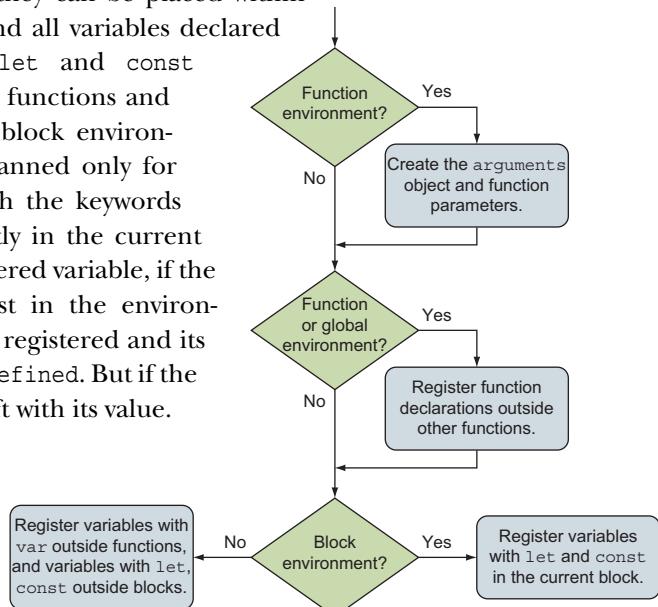
Figure 5.14   **The process of registering identifiers, depending on the type of environment**

create but tricky to understand. Let's start with why we're able to call a function before it's even declared.

CALLING FUNCTIONS BEFORE THEIR DECLARATIONS

One of the features that makes JavaScript pleasant to use is that the order of function definitions doesn't matter. Those who have used Pascal may not remember its rigid structural requirements fondly. In JavaScript, we can call a function even before it's formally declared. Check out the following listing.

Listing 5.9   Accessing a function before its declaration

> We can access a function that isn't yet defined, if the function is defined as a function declaration.

```
assert(typeof fun === "function",
  "fun is a function even though its definition isn't reached yet!");      ◁

assert(typeof myFunExp === "undefined",
  "But we cannot access function expressions");

assert(typeof myArrow === "undefined",
  "Nor arrow functions");

function fun(){}      ◁

var myFunExpr = function(){};
var myArrow = (x) => x;
```

> We can't access functions that are defined as function expressions or arrow functions.

> The fun function is defined as a function declaration.

> myFunExpr points to a function expression, and myArrow to an arrow function.

We can access the function `fun` even before we've defined it. We can do this because `fun` is defined as a function declaration, and the second step (listed previously in this section) indicates that functions created with function declarations are created and their identifiers registered when the current lexical environment is created, *before* any JavaScript code is executed. So even before we start executing our `assert` call, the `fun` function already exists.

The JavaScript engine does this to make things easier for us as developers, allowing us to forward-reference functions and not burdening us with an exact order for placing functions. Functions already exist at the time our code starts executing.

Notice that this holds only for function declarations. Function expressions and arrow functions aren't part of this process, and are created when the program execution reaches their definitions. This is why we can't access the `myFunExp` and `myArrow` functions.

OVERRIDING FUNCTIONS

The next conundrum to tackle is the problem of overriding function identifiers. Let's take a look at another example.

Listing 5.10   Overriding function identifiers

```
assert(typeof fun === "function", "We access the function");     ⟵⎺┐ fun refers to
                                                                    │ a function.

var fun = 3;    ⟵──── Defines a variable fun and assigns a number to it

assert(typeof fun === "number", "Now we access the number");     ⟵⎺┐ fun refers to
                                                                    │ a number.

function fun(){}      ⟵──── A fun function declaration

assert(typeof fun === "number", "Still a number");     ⟵──── fun still refers to a number.
```

In this example, a variable declaration and a function declaration have the same name: fun. If you run this code, you'll see that both asserts pass. In the first assert, the identifier fun refers to a function; and in the second and third, fun refers to a number.

This behavior follows as a direct consequence of the steps taken when registering identifiers. In the second step of the outlined process, functions defined with function declarations are created and associated to their identifiers before any code is evaluated; and in the third step, variable declarations are processed, and the value undefined is associated to identifiers that haven't yet been encountered in the current environment.

In this case, because the identifier fun has been encountered in the second step when function declarations are registered, the value undefined isn't assigned to the variable fun. This is why the first assertion, testing whether fun is a function, passes. After that, we have an assignment statement, var fun = 3, which assigns the number 3 to the identifier fun. By doing this, we lose the reference to the function, and from then on, the identifier fun refers to a number.

During the actual program execution, function declarations are skipped, so the definition of the fun function doesn't have any impact on the value of the fun identifier.

Variable hoisting

If you've read a bunch of JavaScript blogs or books explaining identifier resolution, you've probably run into the term *hoisting*—for example, variable and function declarations are hoisted, or *lifted*, to the top of a function or global scope.

As you've seen, though, that's a simplistic view. Variables and function declarations are technically not "moved" anywhere. They're visited and registered in lexical environments before any code is executed. Although *hoisting*, as it's most often defined, is enough to provide a basic understanding of how JavaScript scoping works, we've gone much deeper than that by looking at lexical environments, taking another step on the path of becoming a true JavaScript ninja.

In the next section, all the concepts that we've explored so far in this chapter will help you better understand closures.

5.6 Exploring how closures work

We started this chapter with closures, a mechanism that allows a function to access all variables that are in scope when the function itself is created. You've also seen some of the ways closures can help you—for example, by allowing us to mimic private object variables or by making our code more elegant when dealing with callbacks.

Closures are irrevocably tightly coupled with scopes. Closures are a straightforward side effect of the way scoping rules work in JavaScript. So in this section, we'll revisit the closure examples from the beginning of the chapter. But this time you'll take advantage of execution contexts and lexical environments that will enable you to grasp how closures work under the hood.

5.6.1 Revisiting mimicking private variables with closures

As you've already seen, closures can help us mimic private variables. Now that we have a solid understanding of how scoping rules work in JavaScript, let's revisit the private variables example. This time, we'll focus on execution contexts and lexical environments. Just to make things easier, let's repeat the listing.

Listing 5.11 Approximate private variables with closures

Declares a variable inside the constructor. Because the scope of the variable is limited to inside the constructor, it's a "private" variable.

The increment method for the value. Because the value is private, no one can screw it up behind our backs; they're limited to the access that we give them via methods.

An accessor method for the feints counter

Calls the feint method, which increments the count of the number of times that our ninja has feinted

Verifies that we can't get at the variable directly

Tests that the increment was performed

```
function Ninja() {
  var feints = 0;
  this.getFeints = function(){
    return feints;
  };
  this.feint = function(){
    feints++;
  };
}
var ninja1 = new Ninja();
assert(ninja1.feints === undefined,
       "And the private data is inaccessible to us.");
ninja1.feint();
assert(ninja1.getFeints()=== 1,
       "We're able to access the internal feint count.");

var ninja2 = new Ninja();
assert(ninja2.getFeints()=== 0,
       "The second ninja object gets its own feints variable.");
```

When we create a new ninja2 object with the Ninja constructor, the ninja2 object gets its own feints variable.

2. When the constructor function is entered, a new lexical environment is created. It keeps track of all local variables created in this scope. In this case, it keeps a reference to the feints variable.

```
function Ninja() {
  var feints = 0;
  this.getFeints = function(){
    return feints;
  };
  this.feint = function(){
    feints++;
  };
}
var ninja1 = new Ninja();
```

Ninja environment

feints: 0

ninja1

getFeints ────────── function(){}

feint ────────── function(){}

1. When the keyword new is used, a new object is instantiated.

3. During the execution of the constructor, two functions are created and assigned as properties of the newly created object (getFeints and feint). As with any function, these two functions keep a reference to the environment in which they were created (the Ninja environment).

Figure 5.15   Private variables are realized as closures that are created by object methods defined in the constructor.

Now we'll analyze the state of the application after the first `Ninja` object has been created, as shown in figure 5.15. We can use our knowledge of the intricacies of identifier resolution to better understand how closures come into play in this situation. JavaScript constructors are functions invoked with the keyword `new`. Therefore, *every* time we invoke a constructor function, we create a *new* lexical environment, which keeps track of variables local to the constructor. In this example, a new `Ninja` environment that keeps track of the `feints` variable is created.

In addition, whenever a function is created, it keeps a reference to the lexical environment in which it was created (through an internal `[[Environment]]` property). In this case, within the `Ninja` constructor function, we create two new functions: `get-Feints` and `feint`, which get a reference to the `Ninja` environment, because this is the environment in which they were created.

The `getFeints` and `feint` functions are assigned as methods of the newly created `ninja` object (which, if you remember from the previous chapter, is accessible through the `this` keyword). Therefore, `getFeints` and `feint` will be accessible from outside the `Ninja` constructor function, which in turn leads to the fact that you've effectively created a closure around the `feints` variable.

The methods keep alive the environments in which they were created, thus keeping alive the "private" variables of each instance.

Ninja environment		Ninja environment
feints: 1		feints: 0

```
Ninja
environment
feints: 1                [[Environment]]
ninja1
getFeints  →  function(){}
feint      →  function(){}

var ninja1 = new Ninja();
ninja1.feint()
```

```
Ninja
environment
feints: 0                [[Environment]]
ninja2
getFeints  →  function(){}
feint      →  function(){}

var ninja2 = new Ninja();
```

Figure 5.16 The methods of each instance create closures around the "private" instance variables.

When we create another `Ninja` object, the `ninja2` object, the whole process is repeated. Figure 5.16 shows the state of the application after creating the second `Ninja` object.

Every object created with the `Ninja` constructor gets its own methods (the `ninja1.getFeints` method is different from the `ninja2.getFeints` method) that close around the variables defined when the constructor was invoked. These "private" variables are accessible only through object methods created within the constructor, and not directly!

Now let's see how things play out when making the `ninja2.getFeints()` call. Figure 5.17 shows the details.

Before making the `ninja2.getFeints()` call, our JavaScript engine is executing global code. Our program execution is in the global execution context, which is also the only context in the execution stack. At the same time, the only active lexical environment is the global environment, the environment associated with the global execution context.

When making the `ninja2.getFeints()` call, we're calling the `getFeints` method of the `ninja2` object. Because every function call causes the creation of a new execution context, a new `getFeints` execution context is created and pushed to the execution stack. This also leads to the creation of a new `getFeints` lexical environment, which is normally used to keep track of variables defined in this function. In addition, the `getFeints` lexical environment, as its outer environment, gets the environment in which the `getFeints` function was created, the `Ninja` environment that was active when the `ninja2` object was constructed.

```
function Ninja() {
  var feints = 0;
  this.getFeints = function(){
    return feints;
  };
  this.feint = function(){
    feints++;
  };
}

var ninja1 = new Ninja();
var ninja1.feint();

var ninja2 = new Ninja();
ninja2.getFeints();
```
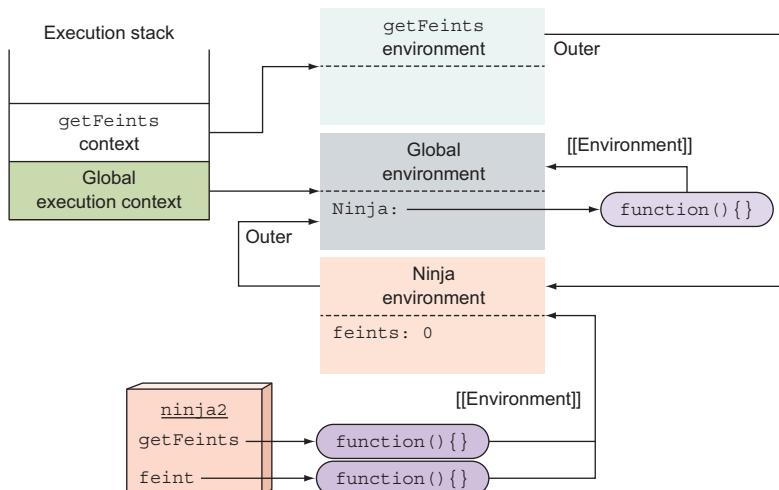

Figure 5.17 **The state of execution contexts and lexical environments when performing the `ninja2.getFeints()` call. A new `getFeints` environment is created that has the environment of the constructor function in which `ninja2` was created as its outer environment. `getFeints` can access the "private" `feints` variable.**

Now let's see how things play out when we try to get the value of the `feints` variable. First, the currently active `getFeints` lexical environment is consulted. Because we haven't defined any variables in the `getFeints` function, this lexical environment is empty and our targeted `feints` variable won't be found in there. Next, the search continues in the *outer* environment of the current lexical environment—in our case, the `Ninja` environment is active when constructing the `ninja2` object. This time around, the `Ninja` environment has a reference to the `feints` variable, and the search is done. It's as simple as that.

   Now that we understand the role that execution contexts and lexical environments play when dealing with closures, we'd like to turn our attention to "private" variables

and why we keep putting quotes around them. As you might have figured out by now, these "private" variables aren't private properties of the object, but are variables kept alive by the object methods created in the constructor. Let's take a look at one interesting side effect of this.

5.6.2 Private variables caveat

In JavaScript, there's nothing stopping us from assigning properties created on one object to another object. For example, we can easily rewrite the code from listing 5.11 into something like the following.

Listing 5.12   Private variables are accessed through functions, not through objects!

```
function Ninja() {
  var feints = 0;
  this.getFeints = function(){
    return feints;
  };
  this.feint = function(){
    feints++;
  };
}
var ninja1 = new Ninja();
ninja1.feint();

var imposter = {};
imposter.getFeints = ninja1.getFeints;      ◁——

assert(imposter.getFeints() === 1,
     "The imposter has access to the feints variable!");
```

Makes the getFeints function of ninja1 accessible through the imposter

Verifies that we can access the supposedly private variable of ninja1

This listing modifies the source code in a way that it assigns the `ninja1.getFeints` method to a completely new `imposter` object. Then, when we call the `getFeints` function on the `imposter` object, we test that we can access the value of the variable `feints` created when `ninja1` was instantiated, thus proving that we're faking this whole "private" variable thing. See figure 5.18.

This example illustrates that there aren't any private object variables in JavaScript, but that we can use closures created by object methods to have a "good enough" alternative. Still, even though it isn't the real thing, lots of developers find this way of hiding information useful.
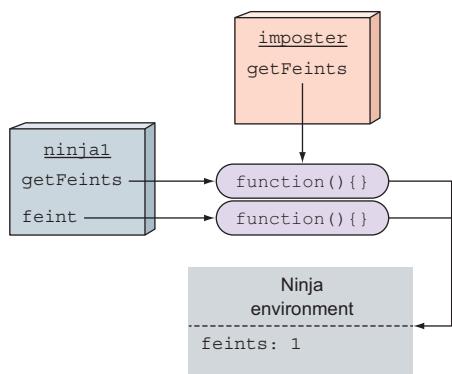
Figure 5.18   We can access the "private" variables through functions, even if that function is attached to another object!

5.6.3   Revisiting the closures and callbacks example

Let's go back to our simple animations example with callback timers. This time, we'll animate two objects, as shown in the following listing.

Listing 5.13   Using a closure in a `timer` interval callback

```
<div id="box1">First Box</div>
<div id="box2">Second Box</div>
<script>
  function animateIt(elementId) {
    var elem = document.getElementById(elementId);
    var tick = 0;
    var timer = setInterval(function(){
      if (tick < 100) {
        elem.style.left = elem.style.top = tick + "px";
        tick++;
      }
      else {
        clearInterval(timer);
        assert(tick === 100,
               "Tick accessed via a closure.");
        assert(elem,
               "Element also accessed via a closure.");
        assert(timer,
               "Timer reference also obtained via a closure." );
      }
    }, 10);
  }
  animateIt("box1");
  animateIt("box2");
</script>
```

As you saw earlier in the chapter, we use closures to simplify animating multiple objects on our pages. But now we'll consider lexical environments, as shown in figure 5.19.

Every time we call the `animateIt` function, a new function lexical environment is created ❶ ❷ that keeps track of the set of variables important for that animation (the `elementId`; `elem`, the element that's being animated; `tick`, the current number of ticks; and `timer`, the ID of the timer doing the animation). That environment will be kept alive as long as there's at least one function that works with its variables through closures. In this case, the browser will keep alive the `setInterval` callback until we call the `clearInterval` function. Later, when an interval expires, the browser calls the matching callback—and with it, through closures, come the variables defined when the callback was created. This enables us to avoid the trouble of manually mapping the callback and the active variables ❸ ❹ ❺, thereby significantly simplifying our code.

That's all we have to say about closures and scopes. Now recap this chapter and meet us in the next one, where we'll explore two completely new ES6 concepts, generators and promises, that can help when writing asynchronous code.

```
<div id="box1">First box</div>
<div id="box2">Second box</div>
<script>
function animateIt(elementId) {
    var elem = document.getElementById(elementId);
    var tick = 0;
     var timer = setInterval(function (){
      if (tick < 100) {
         var position = tick + "px";
         elem.style.left = position;
         elem.style.top = position;
         tick++;
      }
      else {
       clearInterval(timer);
      }
    }, 10);
}
animateIt("box1");
animateIt("box2");
</script>
```

❶ A function lexical environment is created on the first animateIt call.

❷ A function lexical environment is created on the second animateIt call.

animateIt("box1"); environment	animateIt("box2"); environment
elementId: "box1" elem: <div id="box1"> tick: 0 timer: 1	elementId: "box2" elem: <div id="box2"> tick: 0 timer: 2

The states of the environments after a callback function has been executed

function(){} ← function(){}

Set as interval callbacks, one for each call to the animateIt function

animateIt("box1"); environment	animateIt("box2"); environment	animateIt("box1"); environment
elementId: "box1" elem: <div id="box1"> **tick: 1** timer: 1	elementId: "box2" elem: <div id="box2"> **tick: 1** timer: 2	elementId: "box1" elem: <div id="box1"> **tick: 2** timer: 1

function(){}   function(){}   function(){}

Time

❸ The setInterval registered on the animateIt("box1") expires.

❹ The setInterval registered on the animateIt("box2") expires.

❺ The setInterval registered on the animateIt("box1") expires for the second time.

Figure 5.19 By creating multiple closures, we can do many things at once. Every time an interval expires, the callback function reactivates the environment that was active at the time of callback creation. The closure of each callback automatically keeps track of its own set of variables.

5.7   Summary

- Closures allow a function to access all variables that are in scope when the function itself was defined. They create a "safety bubble" of the function and the variables that are in scope at the point of the function's definition. This way, the function has all it needs to execute, even if the scope in which the function was created is long gone.

- We can use function closures for these advanced uses:
  - Mimic private object variables, by closing over constructor variables through method closures
  - Deal with callbacks, in a way that significantly simplifies our code

- JavaScript engines track function execution through an execution context stack (or a call stack). Every time a function is called, a new function execution context is created and placed on the stack. When a function is done executing, the matching execution context is popped from the stack.

- JavaScript engines track identifiers with lexical environments (or colloquially, scopes).

- In JavaScript, we can define globally-scoped, function-scoped, and even-block scoped variables.

- To define variables, we use `var`, `let`, and `const` keywords:
  - The `var` keyword defines a variable in the closest function or global scope (while ignoring blocks).
  - `let` and `const` keywords define a variable in the closest scope (including blocks), allowing us to create block-scoped variables, something that wasn't possible in pre-ES6 JavaScript. In addition, the keyword `const` allows us to define "variables" whose value can be assigned only once.

- Closures are merely a side effect of JavaScript scoping rules. A function can be called even when the scope in which it was created is long gone.

5.8   Exercises

1 Closures allow functions to

  a Access external variables that are in scope when the function is defined

  b Access external variables that are in scope when the function is called

2 Closures come with

  a Code-size costs

  b Memory costs

  c Processing costs

3 In the following code example, mark the identifiers accessed through closures:

```
function Samurai(name) {
  var weapon = "katana";
```

```
  this.getWeapon = function(){
    return weapon;
  };

  this.getName = function(){
    return name;
  }

  this.message = name + " wielding a " + weapon;

  this.getMessage = function(){
    return this.message;
  }
}

var samurai = new Samurai("Hattori");

samurai.getWeapon();
samurai.getName();
samurai.getMessage();
```

4   In the following code, how many execution contexts are created, and what's the largest size of the execution context stack?

```
function perform(ninja) {
  sneak(ninja);
  infiltrate(ninja);
}

function sneak(ninja) {
  return ninja + " skulking";
}

function infiltrate(ninja) {
  return ninja + " infiltrating";
}

perfom("Kuma");
```

5   Which keyword in JavaScript allows us to define variables that can't be reassigned to a completely new value?

6   What's the difference between `var` and `let`?

7   Where and why will the following code throw an exception?

```
getNinja();
getSamurai();

function getNinja() {
  return "Yoshi";
}

var getSamurai = () => "Hattori";
```

Functions for the future:
generators and promises

This chapter covers

- Continuing function execution with generators
- Handling asynchronous tasks with promises
- Achieving elegant asynchronous code by combining generators and promises

In the previous three chapters, we focused on functions, specifically on how to define functions and how to use them to great effect. Although we've introduced some ES6 features, such as arrow functions and block scopes, we've mostly been exploring features that have been part of JavaScript for quite some time. This chapter tackles the cutting edge of ES6 by presenting *generators* and *promises,* two completely new JavaScript features.

> **NOTE** Generators and promises are both introduced by ES6. You can check out current browser support at http://mng.bz/sOs4 and http://mng.bz/Du38.

Generators are a special type of function. Whereas a standard function produces at most a single value while running its code from start to finish, generators produce multiple values, on a per request basis, while suspending their execution between these requests. Although new in JavaScript, generators have existed for quite some time in Python, PHP, and C#.

Generators are often considered an almost weird or fringe language feature not often used by the average programmer. Though most of this chapter's examples are designed to teach you how generator functions work, we'll also explore some practical aspects of generators. You'll see how to use generators to simplify convoluted loops and how to take advantage of generators' capability to suspend and resume their execution, which can help you write simpler and more elegant asynchronous code.

Promises, on the other hand, are a new, built-in type of object that help you work with asynchronous code. A promise is a placeholder for value that we don't have yet but will at some later point. They're especially good for working with multiple asynchronous steps.

In this chapter, you'll see how both generators and promises work, and we'll finish off by exploring how to combine them to greatly simplify our dealings with asynchronous code. But before going into the specifics, let's take a sneak peek into how much more elegant our asynchronous code can be.

. . . .

	What are some common uses for a generator function?
	Why are promises better than simple callbacks for asynchronous code?
Do you know?	**You start a number of long-running tasks with Promise .race. When does the promise resolve? When would it fail to resolve?**

. . . .

6.1 Making our async code elegant with generators and promises

Imagine that you're a developer working at freelanceninja.com, a popular freelance ninja recruitment site enabling customers to hire ninjas for stealth missions. Your task is to implement a functionality that lets users get details about the highest-rated mission done by the most popular ninja. The data representing the ninjas, the summaries of their missions, as well as the details of the missions are stored on a remote server, encoded in JSON. You might write something like this:

```
try {
    var ninjas = syncGetJSON("ninjas.json");
    var missions = syncGetJSON(ninjas[0].missionsUrl);
    var missionDetails = syncGetJSON(missions[0].detailsUrl);
    //Study the mission description
}
catch(e){
  //Oh no, we weren't able to get the mission details
}
```

This code is relatively easy to understand, and if an error occurs in any of the steps, we can easily catch it in the catch block. But unfortunately, this code has a big problem. Getting data from a server is a long-running operation, and because JavaScript relies on a single-threaded execution model, we've just blocked our UI until the long-running operations finish. This leads to unresponsive applications and disappointed users. To solve this problem, we can rewrite it with callbacks, which will be invoked when a task finishes, without blocking the UI:

```
getJSON("ninjas.json", function(err, ninjas){
  if(err) {
    console.log("Error fetching list of ninjas", err);
    return;
  }
  getJSON(ninjas[0].missionsUrl, function(err, missions) {
    if(err) {
      console.log("Error locating ninja missions", err);
      return;
    }
  getJSON(missions[0].detailsUrl, function(err, missionDetails){
    if(err) {
      console.log("Error locating mission details", err);
      return;
    }
    //Study the intel plan
    });
  });
});
```

Although this code will be much better received by our users, you'll probably agree that it's messy, it adds a lot of boilerplate error-handling code, and it's plain ugly. This is where generators and promises jump in. By combining them, we can turn the non-blocking but awkward callback code into something much more elegant:

A generator function is defined by putting an asterisk right after the function keyword. We can use the new yield keyword in generator functions.

The promises are hidden within the getJSON method.

```
async(function*(){
  try {
    const ninjas = yield getJSON("ninjas.json");
    const missions = yield getJSON(ninjas[0].missionsUrl);
    const missionDescription = yield getJSON(missions[0].detailsUrl);
     //Study the mission details
  }
  catch(e) {
    //Oh no, we weren't able to get the mission details
  }
});
```

Don't worry if this example doesn't make much sense or if you find some of the syntax (such as function* or yield) unfamiliar. By the end of this chapter, you'll meet all

the necessary ingredients. For now, it's enough that you compare the elegance (or the lack thereof) of the nonblocking callback code and the nonblocking generators and promises code.

Let's start slowly by exploring generator functions, the first stepping stone toward elegant asynchronous code.

6.2 Working with generator functions

Generators are a completely new type of function and are significantly different from standard, run-of-the-mill functions. A *generator* is a function that generates a sequence of values, but not all at once, as a standard function would, but on a per request basis. We have to explicitly ask the generator for a new value, and the generator will either respond with a value or notify us that it has no more values to produce. What's even more curious is that after a value is produced, a generator function doesn't end its execution, as a normal function would. Instead, a generator is merely suspended. Then, when a request for another value comes along, the generator resumes where it left off.

The following listing provides a simple example of using a generator to generate a sequence of weapons.

Listing 6.1   Using a generator function to generate a sequence of values

Defines a generator function by putting *
after the function keyword

```
function* WeaponGenerator(){
  yield "Katana";
  yield "Wakizashi";
  yield "Kusarigama";
}

for(let weapon of WeaponGenerator()) {
  assert(weapon !== undefined, weapon);
}
```

Generates individual values by using the new yield keyword

Consumes the generated sequence with the new for-of loop

We start by defining a generator that will produce a sequence of weapons. Creating a generator function is simple: We append an asterisk (*) after the `function` keyword. This enables us to use the new `yield` keyword within the body of the generator to produce individual values. Figure 6.1 illustrates the syntax.

In this example, we create a generator called `WeaponGenerator` that produces a sequence of weapons: `Katana`, `Wakizashi`, and `Kusarigama`. One way of consuming that sequence of weapons is by using a new kind of loop, the `for-of` loop:

```
for(let weapon of WeaponGenerator()) {
  assert(weapon, weapon);
}
```

The result of invoking this `for-of` loop is shown in figure 6.2. (For now, don't worry much about the `for-of` loop, as we'll explore it later.)

**Place an * after the function keyword
to define a generator function.**

```
function* WeaponGenerator() {
   ...
   yield "Katana";
   ...
}
```

**Within generator functions, use
yield to produce individual values.**

**Figure 6.1   Add an asterisk (*) after the
function keyword to define a generator.**
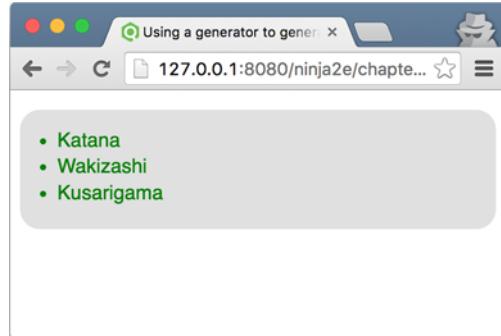
**Figure 6.2   The result of iterating
over our `WeaponGenerator()`**

On the right side of the for-of loop, we've placed the result of invoking our genera-
tor. But if you take a closer look at the body of the WeaponGenerator function, you'll
see that there's no return statement. What's up with that? In this case, shouldn't the
right side of the for-of loop evaluate to undefined, as would be the case if we were
dealing with a standard function?

   The truth is that generators are quite unlike standard functions. For starters, call-
ing a generator doesn't execute the generator function; instead it creates an object
called an *iterator*. Let's explore that object.

6.2.1   Controlling the generator through the iterator object

Making a call to a generator doesn't mean that the body of the generator function will
be executed. Instead, an iterator object is created, an object through which we can com-
municate with the generator. For example, we can use the iterator to request additional
values. Let's adjust our previous example to explore how the iterator object works.

> **Listing 6.2   Controlling a generator through an iterator object**

**Defines a generator that will produce
a sequence of two weapons**

```
function* WeaponGenerator(){
  yield "Katana";
  yield "Wakizashi";
}

const weaponsIterator = WeaponGenerator();

const result1 = weaponsIterator.next();
assert(typeof result1 === "object"
    && result1.value === "Katana"
    && !result1.done,
    "Katana received!");
```

**Calling a generator
creates an iterator
through which we
control the
generator's
execution.**

**Calling the
iterator's next
method requests
a new value from
the generator.**

**The result is an object with a
returned value and an
indicator that tells us whether
the generator has more values.**

```
const result2 = weaponsIterator.next();
assert(typeof result2 === "object"
    && result2.value === "Wakizashi"
    && !result2.done,
    "Wakizashi received!");
```

Calling next again gets another value from the generator.

```
const result3 = weaponsIterator.next();
assert(typeof result3 === "object"
    && result3.value === undefined
    && result3.done,
    "There are no more results!");
```

When there's no more code to execute, the generator returns "undefined" and indicates that it's done.

As you can see, when we call a generator, a new *iterator* is created:

```
const weaponsIterator = WeaponGenerator();
```

The iterator is used to control the execution of the generator. One of the fundamental things that the iterator object exposes is the next method, which we can use to control the generator by requesting a value from it:

```
const result1 = weaponsIterator.next();
```

As a response to that call, the generator executes its code until it reaches a yield keyword that produces an intermediary result (one item in the generated sequence of items), and returns a *new* object that encapsulates that result and tells us whether its work is done.

As soon as the current value is produced, the generator suspends its execution without blocking and patiently waits for another value request. This is an incredibly powerful feature that standard functions don't have, a feature that we'll use later to great effect.

In this case, the first call to the iterator's next method executes the generator code to the first yield expression, yield "Katana", and returns an object with the property value set to Katana and the property done set to false, signaling that there are more values to produce.

Later, we request another value from the generator, by making another call to the weaponIterator's next method:

```
const result2 = weaponsIterator.next();
```

This wakes up the generator from suspension, and the generator continues where it left off, executing its code until another intermediary value is reached: yield "Wakizashi". This suspends the generator and produces an object carrying Wakizashi.

Finally, when we call the next method for the third time, the generator resumes its execution. But this time there's no more code to execute, so the generator returns an object with value set to undefined, and done set to true, signaling that it's done with its work.

Now that you've seen how to control generators through iterators, you're ready to learn how to iterate over the produced values.

ITERATING THE ITERATOR

The iterator, created by calling a generator, exposes a `next` method through which we can request a new value from the generator. The `next` method returns an object that carries the value produced by the generator, as well as the information stored in the `done` property that tells us whether the generator has additional values to produce.

Now we'll take advantage of these facts to use a plain old `while` loop to iterate over values produced by a generator. See the following listing.

> **Listing 6.3   Iterating over generator results with a `while` loop**

Creates a variable in which we'll store items of the generated sequence

Creates an iterator

On each loop iteration, fetches one value from the generator and outputs its value. Stops iterating when the generator has no more values to produce.

```
function* WeaponGenerator(){
  yield "Katana";
  yield "Wakizashi";
}

const weaponsIterator = WeaponGenerator();
let item;
while(!(item = weaponsIterator.next()).done) {
  assert(item !== null, item.value);
}
```

Here we again create an iterator object by calling a generator function:

```
const weaponsIterator = WeaponGenerator();
```

We also create an `item` variable in which we'll store individual values produced by the generator. We continue by specifying a `while` loop with a slightly complicated looping condition, which we'll break down a bit:

```
while(!(item = weaponsIterator.next()).done) {
  assert(item !== null, item.value)
}
```

On each loop iteration, we fetch a value from the generator by calling the `next` method of our `weaponsIterator`, and we store that value in the `item` variable. Like all such objects, the object referenced by the `item` variable has a `value` property that stores the value returned from the generator, and a `done` property that signals whether the generator is finished producing values. If the generator isn't done with its work, we go into another iteration of the loop; and if it is, we stop looping.

And that's how the `for-of` loop, from our first generator example, works. The `for-of` loop is syntactic sugar for iterating over iterators:

```
for(var item of WeaponGenerator ()){
  assert(item !== null, item);
}
```

Instead of manually calling the `next` method of the matching iterator and always checking whether we're finished, we can use the `for-of` loop to do the exact same thing, only behind the scenes.

YIELDING TO ANOTHER GENERATOR

Just as we often call one standard function from another standard function, in certain cases we want to be able to delegate the execution of one generator to another. Let's take a look at an example that generates both warriors and ninjas.

> **Listing 6.4 Using `yield*` to delegate to another generator**

```
function* WarriorGenerator(){
  yield "Sun Tzu";
  yield* NinjaGenerator();          ⟵——— yield* delegates to another generator.
  yield "Genghis Khan";
}

function* NinjaGenerator(){
  yield "Hattori";
  yield "Yoshi";
}

for(let warrior of WarriorGenerator()){
  assert(warrior !== null, warrior);
}
```

If you run this code, you'll see that the output is Sun Tzu, Hattori, Yoshi, Genghis Khan. Generating Sun Tzu probably doesn't catch you off guard; it's the first value yielded by the WarriorGenerator. But the second output, Hattori, deserves an explanation.

By using the `yield*` operator on an iterator, we yield to another generator. In this example, from a WarriorGenerator we're yielding to a new NinjaGenerator; all calls to the current WarriorGenerator iterator's next method are rerouted to the Ninja-Generator. This holds until the NinjaGenerator has no work left to do. So in our example, after Sun Tzu, the program generates Hattori and Yoshi. Only when the Ninja-Generator is done with its work will the execution of the original iterator continue by outputting Genghis Khan. Notice that this is happening transparently to the code that calls the original generator. The for-of loop doesn't care that the WarriorGenerator yields to another generator; it keeps calling next until it's done.

Now that you have a grasp of how generators work in general and how delegating to another generator works, let's look at a couple of practical examples.

6.2.2 Using generators

Generating sequences of items is all nice and dandy, but let's get more practical, starting with a simple case of generating unique IDs.

USING GENERATORS TO GENERATE IDS

When creating certain objects, often we need to assign a unique ID to each object. The easiest way to do this is through a global counter variable, but that's kind of ugly because the variable can be accidently messed up from anywhere in our code. Another option is to use a generator, as shown in the following listing.

Listing 6.5   Using generators for generating IDs

Defines an IdGenerator generator function →

```
function *IdGenerator(){
  let id = 0;
  while(true){
    yield ++id;
  }
}
```

A loop that generates an infinite sequence of IDs

A variable that keeps track of IDs. This variable can't be modified from outside our generator.

An iterator through which we'll request new IDs from the generator →

```
const idIterator = IdGenerator();

const ninja1 = { id: idIterator.next().value };
const ninja2 = { id: idIterator.next().value };
const ninja3 = { id: idIterator.next().value };

assert(ninja1.id === 1, "First ninja has id 1");
assert(ninja2.id === 2, "Second ninja has id 2");
assert(ninja3.id === 3, "Third ninja has id 3");
```

Requests three new IDs

Tests that all went OK

This example starts with a generator that has one local variable, id, which represents our ID counter. The id variable is local to our generator; there's no fear that someone will accidently modify it from somewhere else in the code. This is followed by an infinite while loop, which at each iteration yields a new id value and suspends its execution until a request for another ID comes along:

```
function *IdGenerator(){
  let id = 0;
  while(true){
    yield ++id;
  }
}
```

> **NOTE**   Writing infinite loops isn't something that we generally want to do in a standard function. But with generators, everything is fine! Whenever the generator encounters a yield statement, the generator execution is suspended until the next method is called again. So every next call executes only one iteration of our infinite while loop and sends back the next ID value.

After defining the generator, we create an iterator object:

```
const idIterator = IdGenerator();
```

This allows us to control the generator with calls to the idIterator.next() method. This executes the generator until a yield is encountered, returning a new ID value that we can use for our objects:

```
const ninja1 = { id: idIterator.next().value };
```

See how simple this is? No messy global variables whose value can be accidentally changed. Instead, we use an iterator to request values from a generator. In addition, if

later we need another iterator for tracking the IDs of, for example, `samurai`, we can initialize a new generator for that.

USING GENERATORS TO TRAVERSE THE DOM

As you saw in chapter 2, the layout of a web page is based on the DOM, a tree-like structure of HTML nodes, in which every node, except the root one, has exactly one parent, and can have zero or more children. Because the DOM is such a fundamental structure in web development, a lot of our code is based around traversing it. One relatively easy way to do this is by implementing a recursive function that will be executed for each visited node. See the following code.

Listing 6.6 Recursive DOM traversal

```
<div id="subTree">
  <form>
    <input type="text"/>
  </form>
  <p>Paragraph</p>
  <span>Span</span>
</div>
<script>
  function traverseDOM(element, callback) {          Processes the current
    callback(element);                               node with a callback
    element = element.firstElementChild;
    while (element) {                                Traverses the DOM of
      traverseDOM(element, callback);                each child element
      element = element.nextElementSibling;
    }
  }
  const subTree = document.getElementById("subTree");
  traverseDOM(subTree, function(element) {           Starts the whole process by
    assert(element !== null, element.nodeName);      calling the traverseDOM
  });                                                function for our root element
</script>
```

In this example, we use a recursive function to traverse all descendants of the element with the id `subtree`, in the process logging each type of node that we visit. In this case, the code outputs DIV, FORM, INPUT, P, and SPAN.

We've been writing such DOM traversal code for a while now, and it has served us perfectly fine. But now that we have generators at our disposal, we can do it differently; see the following code.

Listing 6.7 Iterating over a DOM tree with generators

```
function* DomTraversal(element){
  yield element;                        Uses yield* to transfer the
  element = element.firstElementChild;  iteration control to another
  while (element) {                     instance of the DomTraversal
    yield* DomTraversal(element);       generator
    element = element.nextElementSibling;
```

```
    }
}

const subTree = document.getElementById("subTree");
for(let element of DomTraversal(subTree) {
  assert(element !== null, element.nodeName);
}
```

> Iterates over the nodes
> by using the for-of loop

This listing shows that we can achieve DOM traversals with generators, just as easily as with standard recursion, but with the aditional benefit of not having to use the slightly awkward syntax of callbacks. Instead of processing the subtree of each visited node by recursing another level, we create one generator function for each visited node and yield to it. This enables us to write what's conceptually recursive code in iterable fashion. The benefit is that we can consume the generated sequence of nodes with a simple `for-of` loop, without resorting to nasty callbacks.

This example is a particulary good one, because it also shows how to use generators in order to separate the code that's producing values (in this case, HTML nodes) from the code that's consuming the sequence of generated values (in this case, the `for-of` loop that logs the visited nodes), without having to resort to callbacks. In addition, using iterations is, in certain cases, much more natural than recursion, so it's always good to have our options open.

Now that we've explored some practical aspects of generators, let's go back to a slighty more theoretical topic and see how to exchange data with a running generator.

6.2.3   Communicating with a generator

In the examples presented so far, you've seen how to return multiple values *from* a generator by using `yield` expressions. But generators are even more powerful than that! We can also send data *to* a generator, thereby achieving two-way communication! With a generator, we can produce an intermediary result, use that result to calculate something else from outside the generator, and then, whenever we're ready, send completely new data back to the generator and resume its execution. We'll use this feature to great effect at the end of the chapter to deal with asynchronous code, but for now, let's keep it simple.

SENDING VALUES AS GENERATOR FUNCTION ARGUMENTS

The easiest way to send data to a generator is by treating it like any other function and using function call arguments. Take a look at the following listing.

Listing 6.8   Sending data to and receiving data from a generator

A generator can receive standard arguments, like any other function.

The value sent over next becomes the value of the yielded expression, so our imposter is Hanzo.

The magic happens. By yielding a value, the generator returns an intermediary calculation. By calling the iterator's next method with an argument, we send data back to the generator.

```
function* NinjaGenerator(action) {
  const imposter = yield ("Hattori " + action);

  assert(imposter === "Hanzo",
         "The generator has been infiltrated");
```

```
                    yield ("Yoshi (" + imposter + ") " + action);
                  }

    const ninjaIterator = NinjaGenerator("skulk");

    const result1 = ninjaIterator.next();
    assert(result1.value === "Hattori skulk", "Hattori is skulking");

    const result2 = ninjaIterator.next("Hanzo");
    assert(result2.value === "Yoshi (Hanzo) skulk",
            "We have an imposter!");
```

Normal argument passing

Triggers the execution of the generator and checks that we get the correct value

Sends data to the generator as an argument to the next method and checks whether the value was correctly transferred

A function receiving data is nothing special; plain old functions do it all the time. But remember, generators have this amazing power; they can be suspended and resumed. And it turns out that, unlike standard functions, generators can even receive data *after* their execution has started, whenever we resume them by requesting the next value.

USING THE NEXT METHOD TO SEND VALUES INTO A GENERATOR

In addition to providing data when first invoking the generator, we can send data *into* a generator by passing arguments to the next method. In the process, we wake up the generator from suspension and resume its execution. This passed-in value is used by the generator as the value of the whole yield expression, in which the generator was currently suspended, as shown in figure 6.3.

In this example, we have two calls to the ninjaIterator's next method. The first call, ninjaIterator.next(), requests the first value from the generator. Because our generator hasn't started executing, this call starts the generator, which calculates the


```
function* NinjaGenerator(action) {
  const imposter = yield ("Hattori " + action);
  ...
}
                              "Hattori skulk"

                    imposter = "Hanzo"

const ninjaIterator = NinjaGenerator("skulk");

const result1 = ninjaIterator.next();
...
const result2 = ninjaIterator.next("Hanzo");
```

❸ "Hanzo" is set as the value of the yield expression, so imposter has the value "Hanzo".

❶ The first call stores "Hattori skulk" into result1.

❷ The second call sends "Hanzo" into the generator.

Figure 6.3 The first call to `ninjaIterator.next()` requests a new value from the generator, which returns `Hattori skulk` and suspends the execution of the generator at the `yield` expression. The second call to `ninjaIterator.next("Hanzo")` requests a new value, but also sends the argument `Hanzo` into the generator. This value will be used as the value of the whole `yield` expression, and the variable `imposter` will now carry the value `Hanzo`.

value of the expression `"Hattori " + action`, yields the `Hattori skulk` value, and suspends the generator's execution. There's nothing special about this; we've done something similar multiple times throughout this chapter.

The interesting thing happens on the second call to the `ninjaIterator`'s next method: `ninjaIterator.next("Hanzo")`. This time, we're using the `next` method to pass data back to the generator. Our generator function is patiently waiting, suspended at the expression `yield ("Hattori " + action)`, so the value `Hanzo`, passed as the argument to `next()`, is used as the value of the whole `yield` expression. In our case, this means that the variable `imposter` in `imposter = yield ("Hattori " + action)` will end up with the value `Hanzo`.

That's how we achieve two-way communication with a generator. We use `yield` to return data from a generator, and the `iterator`'s `next()` method to pass data back to the generator.

> **NOTE**   The `next` method supplies the value to the waiting `yield` expression, so if there's no `yield` expression waiting, there's nothing to supply the value to. For this reason, we *can't* supply values over the first call to the `next` method. But remember, if you need to supply an initial value to the generator, you can do so when calling the generator itself, as we did with `NinjaGenerator("skulk")`.

THROWING EXCEPTIONS

There's another, slightly less orthodox, way to supply a value to a generator: by throwing an exception. Each iterator, in addition to having a `next` method, has a `throw` method that we can use to throw an exception back to the generator. Again, let's look at a simple example.

Listing 6.9   Throwing exceptions to generators

```
function* NinjaGenerator() {
  try{                                          This fail shouldn't
    yield "Hattori";                            be reached.
    fail("The expected exception didn't occur");  ◁
  }                                                       Catches exceptions
  catch(e){                                               and tests whether
    assert(e === "Catch this!", "Aha! We caught an exception");  we've received the
  }                                                       expected exception
}

const ninjaIterator = NinjaGenerator();

const result1 = ninjaIterator.next();                   Pulls one
assert(result1.value === "Hattori", "We got Hattori");  value from
                                                        the generator

ninjaIterator.throw("Catch this!");    ◁───  Throws an exception to the generator
```

Listing 6.9 starts similarly to listing 6.8, by specifying a generator called `NinjaGenerator`. But this time, the body of the generator is slightly different. We've surrounded the whole function body code with a `try-catch` block:

```
function* NinjaGenerator() {
  try{
    yield "Hattori";
    fail("The expected exception didn't occur");
  }
  catch(e){
    assert(e === "Catch this!", "Aha! We caught an exception");
  }
}
```

We then continue by creating an iterator, and getting one value from the generator:

```
const ninjaIterator = NinjaGenerator();
const result1 = ninjaIterator.next();
```

Finally, we use the `throw` method, available on all iterators, to throw an exception back to the generator:

```
ninjaIterator.throw("Catch this!");
```

By running this listing, we can see that our exception throwing works as expected, as shown in figure 6.4.

    This feature that enables us to throw exceptions back to generators might feel a bit strange at first. Why would we even want to do that? Don't worry; we won't keep you in the dark for long. At the end of this chapter, we'll use this feature to improve asynchronous server-side communication. Just be patient a bit longer.

Figure 6.4 **We can throw exceptions to generators from outside a generator.**

    Now that you've seen several aspects of generators, we're ready to take a look under the hood to see how generators work.

6.2.4 *Exploring generators under the hood*

So far we know that calling a generator doesn't execute it. Instead, it creates a new iterator that we can use to request values from the generator. After a generator produces (or yields) a value, it suspends its execution and waits for the next request. So in a way, a generator works almost like a small program, a state machine that moves between states:

- *Suspended start*—When the generator is created, it starts in this state. None of the generator's code is executed.
- *Executing*—The state in which the code of the generator is executed. The execution continues either from the beginning or from where the generator was last

suspended. A generator moves to this state when the matching iterator's `next` method is called, and there exists code to be executed.

- *Suspended yield*—During execution, when a generator reaches a `yield` expression, it creates a new object carrying the return value, yields it, and suspends its execution. This is the state in which the generator is paused and is waiting to continue its execution.

- *Completed*—If during execution the generator either runs into a `return` statement or runs out of code to execute, the generator moves into this state.

Figure 6.5 illustrates these states.

Now let's supplement this on an even deeper level, by seeing how the execution of generators is tracked with execution contexts.

```
function* NinjaGenerator(){
    yield "Hattori";
    yield "Yoshi";
}
```

❶ `const ninjaIterator = NinjaGenerator();`
Create a new generator in the Suspended start state.

❷ `const result1 = ninjaIterator.next();`
Activate generator. Move from Suspended start to Executing. Execute up to `yield "Hattori"` and pause. Move to the Suspended yield state. Return a new object: {value: "Hattori", done: false}.

❸ `const result2 = ninjaIterator.next();`
Reactivate generator. Move from Suspended yield to Executing. Execute up to `yield "Yoshi"` and pause. Move to the Suspended yield state. Return a new object: {value: "Yoshi", done: false}.

❹ `const result3 = ninjaIterator.next();`
Reactivate generator. Move from Suspended yield to Executing. No more code to execute. Move to the Completed state. Return a new object: {value: undefined, done: true}.

Figure 6.5 During execution, a generator moves between states triggered by calls to the matching iterator's `next` method.

TRACKING GENERATORS WITH EXECUTION CONTEXTS

In the previous chapter, we introduced the execution context, an internal JavaScript mechanism used to track the execution of functions. Although somewhat special, generators are still functions, so let's take a closer look by exploring the relationship between them and execution contexts. We'll start with a simple code fragment:

```
function* NinjaGenerator(action) {
  yield "Hattori " + action;
  return "Yoshi " + action;
}

const ninjaIterator = NinjaGenerator("skulk");
const result1 = ninjaIterator.next();
const result2 = ninjaIterator.next();
```

Here we reuse our generator that produces two values: `Hattori  skulk` and `Yoshi skulk`.

Now, we'll explore the state of the application, the execution context stack at various points in the application execution. Figure 6.6 gives a snapshot at two positions in the application execution. The first snapshot shows the state of the application execution *before* calling the `NinjaGenerator` function ❶. Because we're executing global code, the execution context stack contains only the global execution context, which references the global environment in which our identifiers are kept. Only the `NinjaGenerator` identifier references a function, while the values of all other identifiers are `undefined`.

When we make the call to the `NinjaGenerator` function ❷

```
const ninjaIterator = NinjaGenerator("skulk");
```

the control flow enters the generator and, as it happens when we enter any other function, a new `NinjaGenerator` execution context item is created (alongside the matching lexical environment) and pushed onto the stack. But because generators are special, *none* of the function code is executed. Instead, a new iterator, which we'll refer to in the code as `ninjaIterator`, is created and returned. Because the iterator is used to control the execution of the generator, the iterator gets a reference to the execution context in which it was created.

An interesting thing happens when the program execution leaves the generator, as shown in figure 6.7. Typically, when program execution returns from a standard function, the matching execution context is popped from the stack and completely discarded. But this isn't the case with generators.
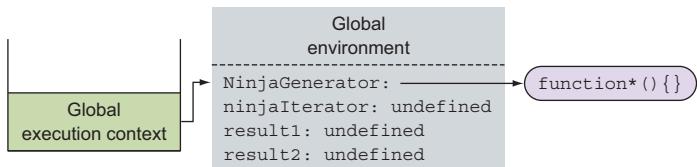
```
function* NinjaGenerator(action) {
  yield "Hattori " + action;
  return "Yoshi " + action;
}

const ninjaIterator = NinjaGenerator("skulk");

const result1 = ninjaIterator.next();
const result2 = ninjaIterator.next();
```
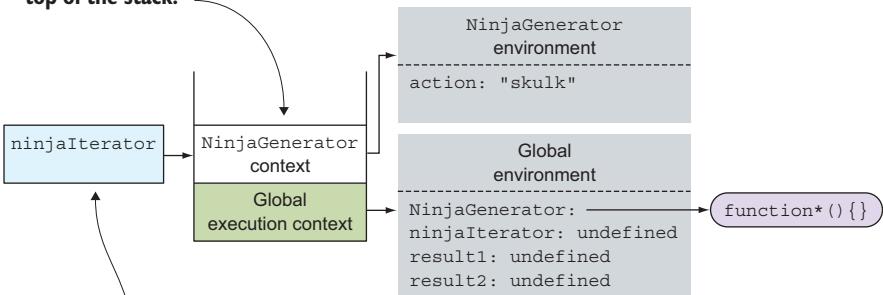
❶ The state of the application before calling the NinjaGenerator function

Global environment

NinjaGenerator: ⟶ function*(){}
ninjaIterator: undefined
result1: undefined
result2: undefined

Global execution context

Suspended start

When entering a generator function, a new stack item is created and placed at the top of the stack.

Executing

ninjaIterator

NinjaGenerator context

Global execution context

Suspended yield

Completed

A new ninjaIterator object is created, with a reference to the current generator context stack item.

The generator starts in the Suspended start state.

❷ The state of the application when calling the NinjaGenerator function

NinjaGenerator environment

action: "skulk"

Global environment

NinjaGenerator: ⟶ function*(){}
ninjaIterator: undefined
result1: undefined
result2: undefined
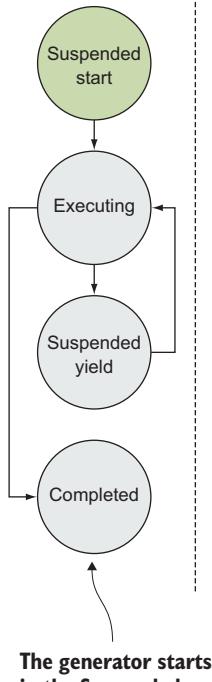
Figure 6.6 **The state of the execution context stack *before* calling the `NinjaGenerator` function ❶, and *when* calling the `NinjaGenerator` function ❷**

```
function* NinjaGenerator(action) {
  yield "Hattori " + action;
  return "Yoshi " + action;
}

const ninjaIterator = NinjaGenerator("skulk");

const result1 = ninjaIterator.next();
const result2 = ninjaIterator.next();
```

The generator is still in the Suspended start state.

The NinjaGenerator context is popped from the stack, but it is kept alive through the ninjaIterator object!
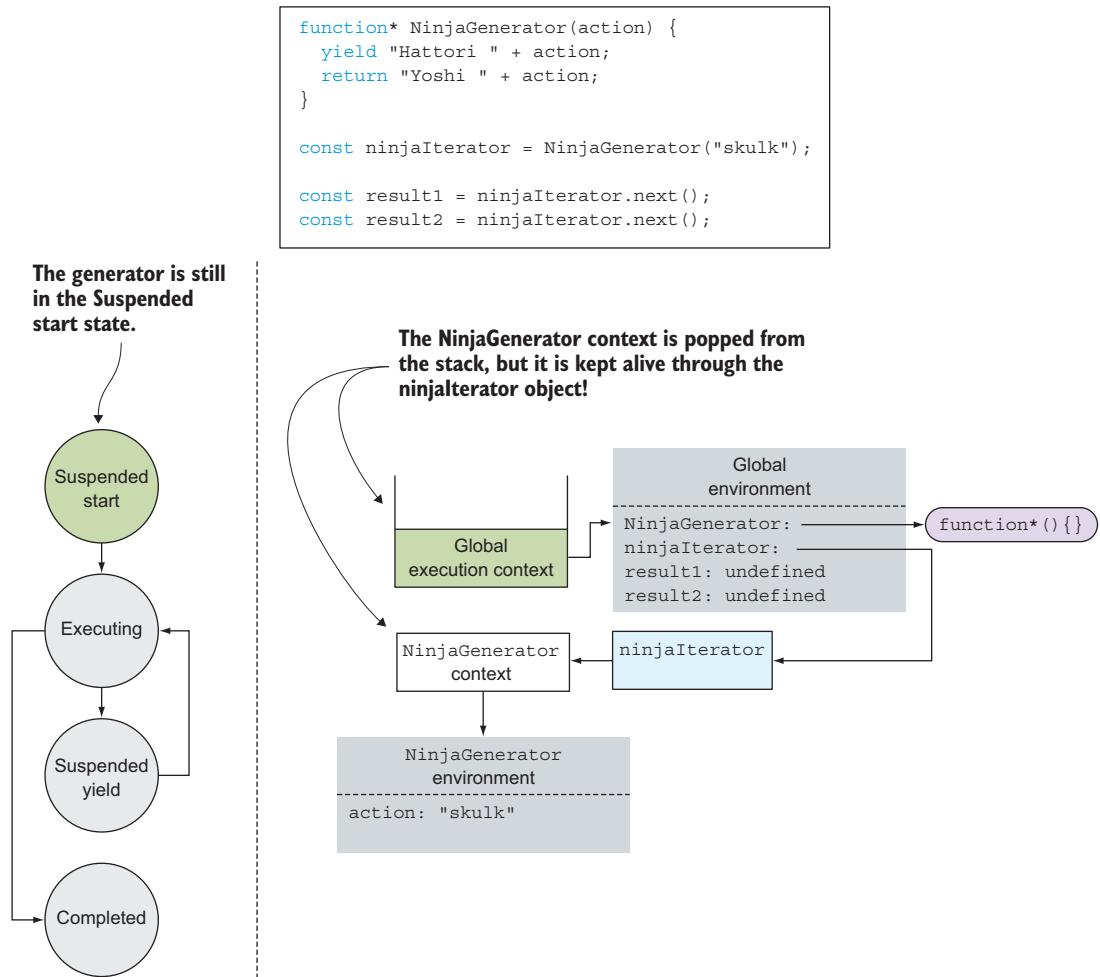
Figure 6.7 **The state of the application when returning from the `NinjaGenerator` call**

The matching `NinjaGenerator` stack item *is* popped from the stack, but it's *not* discarded, because the `ninjaIterator` keeps a reference to it. You can see it as an analogue to closures. In closures, we need to keep alive the variables that are alive at the moment the function closure is created, so our functions keep a reference to the environment in which they were created. In this way, we make sure that the environment and its variables are alive as long as the function itself. Generators, on the other hand, have to be able to resume their execution. Because the execution of all functions is handled by execution contexts, the iterator keeps a reference to its execution context, so that it's alive for as long as the iterator needs it.

Another interesting thing happens when we call the `next` method on the iterator:

```
const result1 = ninjaIterator.next();
```

If this was a standard straightforward function call, this would cause the creation of a *new* next() execution context item, which would be placed on the stack. But as you might have noticed, generators are anything but standard, and a call to the next method of an iterator behaves a lot differently. It reactivates the matching execution context, in this case, the NinjaGenerator context, and places it on top of the stack, continuing the execution where it left off, as shown in figure 6.8.

Figure 6.8 illustrates a crucial difference between standard functions and generators. Standard functions can only be called anew, and each call creates a *new* execution context. In contrast, the execution context of a generator can be temporarily suspended and resumed at will.

In our example, because this is the first call to the next method, and the generator hasn't started executing, the generator starts its execution and moves to the Executing state. The next interesting thing happens when our generator function reaches this point:

```
yield "Hattori " + action
```


The generator moves to the Executing state.

When calling and entering the next() method on the iterator, the existing NinjaGenerator context is reactivated and pushed onto the stack.

```
function* NinjaGenerator(action) {
  yield "Hattori " + action;
  return "Yoshi " + action;
}

const ninjaIterator = NinjaGenerator("skulk");

const result1 = ninjaIterator.next();
const result2 = ninjaIterator.next();
```

Figure 6.8   Calling the iterator's `next` method reactivates the execution context stack item of the matching generator, pushes it on the stack, and continues where it left off the last time.

The generator determines that the expression equals `Hattori skulk`, and the evalua-
tion reaches the `yield` keyword. This means that `Hattori skulk` is the first interme-
diary result of our generator and that we want to suspend the execution of the
generator and return that value. In terms of the application state, a similar thing hap-
pens as before: the `NinjaGenerator` context is taken off the stack, but it's not com-
pletely discarded, because `ninjaIterator` keeps a reference to it. The generator is
now suspended, and has moved to the Suspended Yield state, without blocking. The
program execution resumes in global code, by storing the yielded value to `result1`.
The current state of the application is shown in figure 6.9.

The code continues by reaching another iterator call:

```
const result2 = ninjaIterator.next();
```

```
function* NinjaGenerator(action) {
  yield "Hattori " + action;
  return "Yoshi " + action;
}

const ninjaIterator = NinjaGenerator("skulk");

const result1 = ninjaIterator.next();
const result2 = ninjaIterator.next();
```
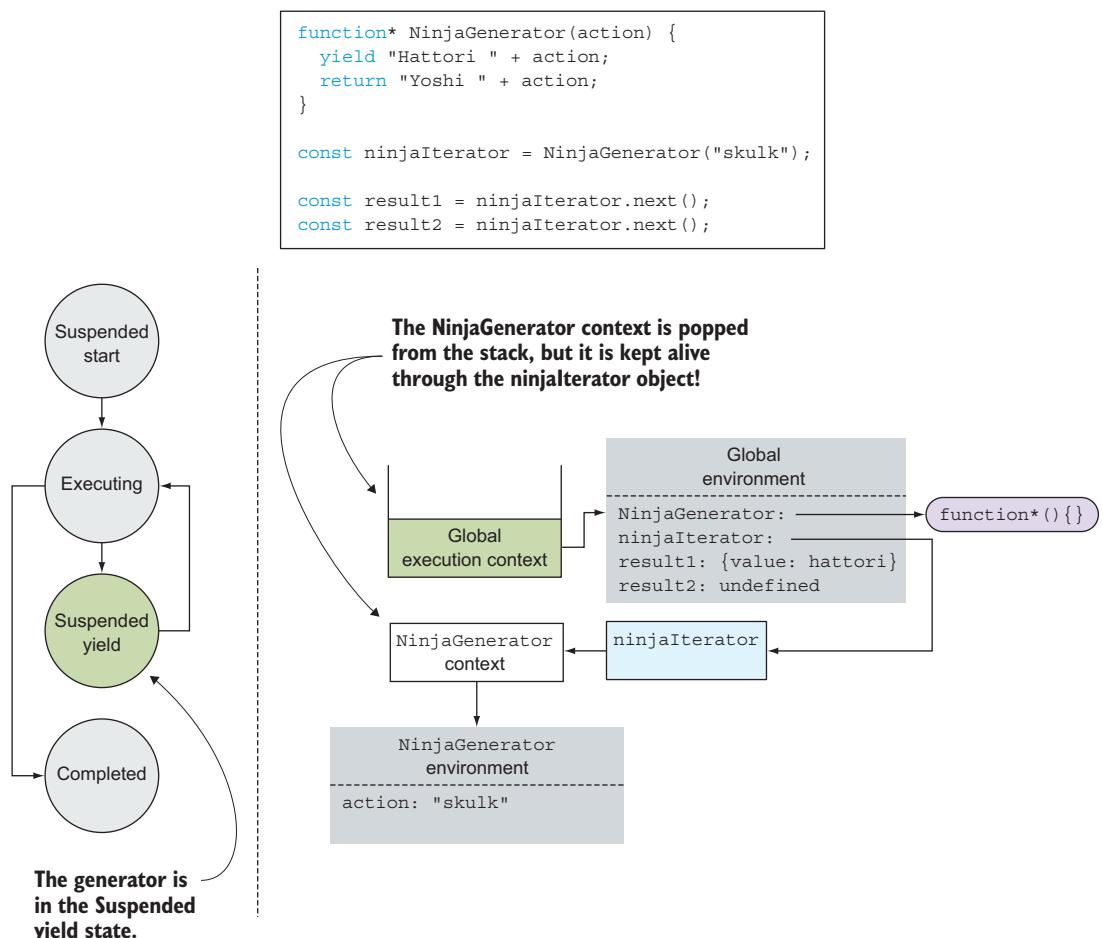

Figure 6.9 After yielding a value, the generator's execution context is popped from the stack (but isn't discarded,
because `ninjaIterator` keeps a reference to it), and the generator execution is suspended (the generator moves
to the Suspended yield state).

At this point, we go through the whole procedure once again: we reactivate the `Ninja-Generator` context referenced by `ninjaIterator`, push it onto the stack, and continue the execution where we left off. In this case, the generator evaluates the expression `"Yoshi " + action`. But this time there's no `yield` expression, and instead the program encounters a `return` statement. This returns the value `Yoshi skulk` and completes the generator's execution by moving the generator into the Completed state.

Uff, this was something! We went deep into how generators work under the hood to show you that all the wonderful benefits of generators are a side effect of the fact that a generator's execution context is kept alive if we yield from a generator, and not destroyed as is the case with return values and standard functions.

Now we recommend that you take a quick breather before continuing on to the second key ingredient required for writing elegant asynchronous code: promises.

6.3   Working with promises

In JavaScript, we rely a lot on asynchronous computations, computations whose results we don't have yet but will at some later point. So ES6 has introduced a new concept that makes handling asynchronous tasks easier: promises.

A *promise* is a placeholder for a value that we don't have now but will have later; it's a guarantee that we'll eventually know the result of an asynchronous computation. If we make good on our promise, our result will be a value. If a problem occurs, our result will be an error, an excuse for why we couldn't deliver. One great example of using promises is fetching data from a server; we promise that we'll eventually get the data, but there's always a chance that problems will occur.

Creating a new promise is easy, as you can see in the following example.

Listing 6.10   Creating a simple promise

A promise is successfully resolved by calling the passed-in resolve function (and rejected by calling the reject function).

Creates a promise by calling a built-in Promise constructor and passing in a callback function with two parameters: resolve and reject

```
const ninjaPromise = new Promise((resolve, reject) => {
  resolve("Hattori");
  //reject("An error resolving a promise!");
});

ninjaPromise.then(ninja => {
  assert(ninja === "Hattori", "We were promised Hattori!");
}, err => {
  fail("There shouldn't be an error")
});
```

And the second is called if an error occurs.

By using the then method on a promise, we can pass in two callbacks; the first is called if a promise is successfully resolved.

To create a promise, we use the new, built-in `Promise` constructor, to which we pass a function, in this case an arrow function (but we could just as easily use a function expression). This function, called an *executor* function, has two parameters: `resolve` and `reject`. The executor is called *immediately* when constructing the `Promise` object with two built-in functions as arguments: `resolve`, which we manually call if we want the promise to resolve successfully, and `reject`, which we call if an error occurs.

This code uses the promise by calling the built-in `then` method on the `Promise` object, a method to which we pass two callback functions: a *success* callback and a *failure* callback. The former is called if the promise is resolved successfully (if the `resolve` function is called on the promise), and the latter is called if there's a problem (either an unhandled exception occurs or the `reject` function is called on a promise).

In our example code, we create a promise and immediately resolve it by calling the `resolve` function with the argument `Hattori`. Therefore, when we call the `then` method, the first, success, callback is executed and the test that outputs `We were promised Hattori!` passes.

Now that we have a general idea of what promises are and how they work, let's take a step back to see some of the problems that promises tackle.

6.3.1 *Understanding the problems with simple callbacks*

We use asynchronous code because we don't want to block the execution of our application (thereby disappointing our users) while long-running tasks are executing. Currently, we solve this problem with callbacks: To a long-running task we provide a function, a callback that's invoked when the task is finally done.

For example, fetching a JSON file from a server is a long-running task, during which we don't want to make the application unresponsive for our users. Therefore, we provide a callback that will be invoked when the task is done:

```
getJSON("data/ninjas.json", function() {
  /*Handle results*/
});
```

Naturally, during this long-running task, errors can happen. And the problem with callbacks is that you can't use built-in language constructs, such as `try-catch` statements, in the following way:

```
try {
  getJSON("data/ninjas.json", function() {
    //Handle results
  });
} catch(e) {/*Handle errors*/}
```

This happens because the code invoking the callback usually isn't executed in the same step of the event loop as the code that starts the long-running task (you'll see exactly what this means when you learn more about the event loop in chapter 13).

As a consequence, errors usually get lost. Many libraries, therefore, define their own conventions for reporting errors. For example, in the Node.js world, callbacks customarily take two arguments, `err` and `data`, where `err` will be a non-null value if an error occurs somewhere along the way. This leads to the first problem with callbacks: *difficult error handling.*

After we've performed a long-running task, we often want to do something with the obtained data. This can lead to starting another long-running task, which can eventually trigger yet another long-running task, and so on—leading to a series of interdependent, asynchronous, callback-processed steps. For example, if we want to execute a sneaky plan to find all ninjas at our disposal, get the location of the first ninja, and send him some orders, we'd end up with something like this:

```
getJSON("data/ninjas.json", function(err, ninjas){
  getJSON(ninjas[0].location, function(err, locationInfo){
    sendOrder(locationInfo, function(err, status){
     /*Process status*/
    })
  })
});
```

You've probably ended up, at least once or twice, with similarly structured code—a bunch of nested callbacks that represent a series of steps that have to be made. You might notice that this code is difficult to understand, inserting new steps is a pain, and error handling complicates your code significantly. You get this "pyramid of doom" that keeps growing and is difficult to manage. This leads us to the second problem with callbacks: *performing sequences of steps is tricky.*

Sometimes, the steps that we have to go through to get to the final result don't depend on each other, so we don't have to make them in sequence. Instead, to save precious milliseconds, we can do them in parallel. For example, if we want to set a plan in motion that requires us to know which ninjas we have at our disposal, the plan itself, and the location where our plan will play out, we could take advantage of jQuery's `get` method and write something like this:

```
var ninjas, mapInfo, plan;

$.get("data/ninjas.json", function(err, data){
  if(err) { processError(err); return; }
  ninjas = data;
  actionItemArrived();
});

$.get("data/mapInfo.json", function(err, data){
  if(err) { processError(err); return; }
  mapInfo = data;
  actionItemArrived();
});
```

```
$.get("plan.json", function(err, data) {
  if(err) { processError(err); return; }

  plan = data;
  actionItemArrived ();
});

function actionItemArrived(){
  if(ninjas != null && mapInfo != null && plan != null){
    console.log("The plan is ready to be set in motion!");
  }
}

function processError(err){
  alert("Error", err)
}
```

In this code, we execute the actions of getting the ninjas, getting the map info, and getting the plan in parallel, because these actions don't depend on each other. We only care that, in the end, we have all the data at our disposal. Because we don't know the order in which the data is received, every time we get some data, we have to check whether it's the last piece of the puzzle that we're missing. Finally, when all pieces are in place, we can set our plan in motion. Notice that we have to write a lot of boilerplate code just to do something as common as executing a number of actions in parallel. This leads us to the third problem with callbacks: *performing a number of steps in parallel is also tricky.*

When presenting the first problem with callbacks—dealing with errors—we showed how we can't use some of the fundamental language constructs, such as `try-catch` statements. A similar thing holds with loops: If you want to perform asynchronous actions for each item in a collection, you have to jump through some more hoops to get it done.

It's true that you can make a library to simplify dealing with all these problems (and many people have). But this often leads to a lot of slightly different ways of dealing with the same problems, so the people behind JavaScript have bestowed upon us *promises*, a standard approach for dealing with asynchronous computation.

Now that you understand most of the reasons behind the introduction of promises, as well as have a basic understanding of them, let's take it up a notch.

6.3.2 Diving into promises

A promise is an object that serves as a placeholder for a result of an asynchronous task. It represents a value that we don't have but hope to have in the future. For this reason, during its lifetime, a promise can go through a couple of states, as shown in figure 6.10.

A promise starts in the *pending* state, in which we know nothing about our promised value. That's why a promise in the pending state is also called an `unresolved` promise. During program execution, if the promise's `resolve` function is called, the
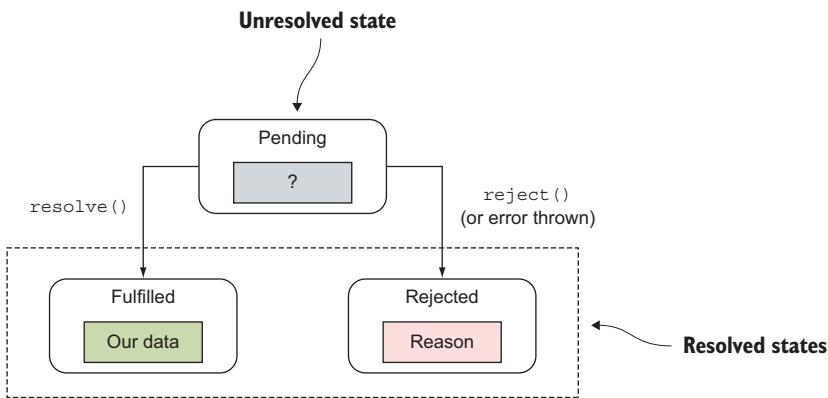
Figure 6.10   **States of a promise**

promise moves into the *fulfilled* state, in which we've successfully obtained the promised value. On the other hand, if the promise's `reject` function is called, or if an unhandled exception occurs during promise handling, the promise moves into the *rejected* state, in which we weren't able to obtain the promised value, but in which we at least know why. Once a promise has reached either the *fulfilled* state or the *rejected* state, it can't switch (a promise can't go from fulfilled to rejected or vice versa), and it always stays in that state. We say that a promise is *resolved* (either successfully or not).

The following listing provides a closer look at what's going on when we use promises.

Listing 6.11   A closer look at promise order of execution

```
report("At code start");

var ninjaDelayedPromise = new Promise((resolve, reject) => {
  report("ninjaDelayedPromise executor");
  setTimeout(() => {
    report("Resolving ninjaDelayedPromise");
    resolve("Hattori");
  }, 500);
});

assert(ninjaDelayedPromise !== null, "After creating ninjaDelayedPromise");

ninjaDelayedPromise.then(ninja => {
  assert(ninja === "Hattori",
        "ninjaDelayedPromise resolve handled with Hattori");
});
```

Calling the Promise constructor immediately invokes the passed-in function.

We'll resolve this promise as successful after a 500ms timeout expires.

The Promise then method is used to set up a callback that will be called when the promise resolves, in our case when the timeout expires.

```
const ninjaImmediatePromise = new Promise((resolve, reject) => {
  report("ninjaImmediatePromise executor. Immediate resolve.");
  resolve("Yoshi");
});

ninjaImmediatePromise.then(ninja => {
  assert(ninja === "Yoshi",
        "ninjaImmediatePromise resolve handled with Yoshi");
});

report("At code end");
```

Creates a new promise that gets immediately resolved

Sets up a callback to be invoked when the promise resolves. But our promise is already resolved!

The code in listing 6.11 outputs the results shown in figure 6.11. As you can see, the code starts by logging the "At code start" message by using our custom-made `report` function (appendix C) that outputs the message onscreen. This enables us to easily track the order of execution.

Next we create a new promise by calling the `Promise` constructor. This immediately invokes the executor function in which we set up a timeout:

```
setTimeout(() => {
  report("Resolving ninjaDelayedPromise");
  resolve("Hattori");
}, 500);
```

The timeout will resolve the promise after 500ms. This could have been any other asynchronous task, but we chose the humble timeout because of its simplicity.

After the `ninjaDelayedPromise` has been created, it still doesn't know the value that it will eventually have, or whether it will even be successful. (Remember, it's still waiting for the timeout that will resolve it.) So after construction, the `ninjaDelayedPromise` is in the first promise state, *pending*.

Next we use the `then` method on the `ninjaDelayedPromise` to schedule a callback to be executed when the promise successfully resolves:

Figure 6.11 The result of executing listing 6.11

```
ninjaDelayedPromise.then(ninja => {
  assert(ninja === "Hattori",
        "ninjaDelayedPromise resolve handled with Hattori");
});
```

This callback will *always* be called asynchronously, regardless of the current state of the promise.

We continue by creating another promise, `ninjaImmediatePromise`, which is resolved immediately during its construction, by calling the `resolve` function. Unlike the `ninjaDelayedPromise`, which after construction is in the *pending* state, the `ninja-ImmediatePromise` finishes construction in the *resolved* state, and the promise already has the value `Yoshi`.

Afterward, we use the `ninjaImmediatePromise`'s `then` method to register a callback that will be executed when the promise successfully resolves. But our promise is already settled; does this mean that the success callback will be immediately called or that it will be ignored? The answer is *neither*.

Promises are designed to deal with asynchronous actions, so the JavaScript engine *always* resorts to asynchronous handling, to make the promise behavior predictable. The engine does this by executing the `then` callbacks after all the code in the current step of the event loop is executed (once again, we'll explore exactly what this means in chapter 13). For this reason, if we study the output in figure 6.11, we'll see that we first log "At code end" and then we log that the `ninjaImmediatePromise` was resolved. In the end, after the 500ms timeout expires, the `ninjaDelayedPromise` is resolved, which causes the execution of the matching `then` callback.

In this example, for the sake of simplicity, we've worked only with the rosy scenario in which everything goes great. But the real world isn't all sunshine and rainbows, so let's see how to deal with all sorts of crazy problems that can occur.

6.3.3 Rejecting promises

There are two ways of rejecting a promise: *explicitly*, by calling the passed-in `reject` method in the executor function of a promise, and *implicitly*, if during the handling of a promise, an unhandled exception occurs. Let's start our exploration with the following listing.

Listing 6.12   Explicitly rejecting promises

```
const promise = new Promise((resolve, reject) => {
  reject("Explicitly reject a promise!");          ◁──  A promise can be explicitly
});                                                       rejected by calling the
                                                          passed-in reject function.

promise.then(
  () => fail("Happy path, won't be called!"),           If a promise is rejected,
  error => pass("A promise was explicitly rejected!")  ◁── the second, error,
);                                                          callback is invoked.
```

We can explicitly reject a promise, by calling the passed-in `reject` method: `reject("Explicitly reject a promise!")`. If a promise is rejected, when registering callbacks through the `then` method, the second, error, callback will always be invoked.

In addition, we can use an alternative syntax for handling promise rejections, by using the built-in `catch` method, as shown in the following listing.

Listing 6.13 Chaining a `catch` method

```
var promise = new Promise((resolve, reject) => {
  reject("Explicitly reject a promise!");
});

promise.then(()=> fail("Happy path, won't be called!"))
       .catch(() => pass("Promise was also rejected"));
```

> Instead of supplying the second, error, callback, we can chain in the catch method, and pass to it the error callback. The end result is the same.

As listing 6.13 shows, we can chain in the `catch` method after the `then` method, to also provide an error callback that will be invoked when a promise gets rejected. In this example, this is a matter of personal style. Both options work equally well, but later, when working with chains of promises, we'll see an example in which chaining the `catch` method is useful.

In addition to explicit rejection (via the `reject` call), a promise can also be rejected implicitly, if an exception occurs during its processing. Take a look at the following example.

Listing 6.14 Exceptions implicitly reject a promise

```
const promise = new Promise((resolve, reject) => {
  undeclaredVariable++;
});

promise.then(() => fail("Happy path, won't be called!"))
       .catch(error => pass("Third promise was also rejected"));
```

> A promise is implicitly rejected if an unhandled exception occurs when processing the promise.

> If an exception occurs, the second, error, callback is invoked.

Within the body of the promise executor, we try to increment `undeclaredVariable`, a variable that isn't defined in our program. As expected, this results in an exception. Because there's no `try-catch` statement within the body of the executor, this results in an implicit rejection of the current promise, and the `catch` callback is eventually invoked. In this situation, we could have just as easily supplied the second callback to the `then` method, and the end effect would be the same.

This way of treating all problems that happen while working with promises in a uniform way is extremely handy. Regardless of how the promise was rejected, whether explicitly by calling the `reject` method or even implicitly, if an exception occurs, all errors and rejection reasons are directed to our rejection callback. This makes our lives as developers a little easier.

Now that we understand how promises work, and how to schedule success and failure callbacks, let's take a real-world scenario, getting JSON-formatted data from a server, and "promisify" it.

6.3.4 Creating our first real-world promise

One of the most common asynchronous actions on the client is fetching data from the server. As such, this is an excellent little case study on the use of promises. For the underlying implementation, we'll use the built-in `XMLHttpRequest` object.

Listing 6.15   Creating a `getJSON` promise

Creates an XMLHttpRequest object

Registers an onload handler that will be called if the server has responded

Try to parse the JSON string; if it succeeds, resolve the promise as successful with the parsed object.

Sends the request

Creates and returns a new promise

Initializes the request

Even if the server has responded, it doesn't mean everything went as expected. Use the result only if the server responds with status 200 (everything OK).

If the server responds with a different status code, or if there's an exception parsing the JSON string, reject the promise.

If there's an error while communicating with the server, reject the promise.

```javascript
function getJSON(url) {
  return new Promise((resolve, reject) => {
    const request = new XMLHttpRequest();

    request.open("GET", url);

    request.onload = function() {
      try {
        if(this.status === 200 ){
          resolve(JSON.parse(this.response));
        } else{
          reject(this.status + " " + this.statusText);
        }
      } catch(e){
        reject(e.message);
      }
    };

    request.onerror = function() {
      reject(this.status + " " + this.statusText);
    };

    request.send();
  });
}

getJSON("data/ninjas.json").then(ninjas => {
  assert(ninjas !== null, "Ninjas obtained!");
}).catch(e => fail("Shouldn't be here:" + e));
```

Uses the promise created by the getJSON function to register resolve and reject callbacks

> **NOTE** Executing this example, and all subsequent examples that reuse this function, requires a running server. You can, for example, use www.npmjs.com/package/http-server.

Our goal is to create a `getJSON` function that returns a promise that will enable us to register success and failure callbacks for asynchronously getting JSON-formatted data from the server. For the underlying implementation, we use the built-in `XMLHttpRequest` object that offers two events: `onload` and `onerror`. The `onload` event is triggered when the browser receives a response from the server, and `onerror` is triggered when an error in communication happens. These event handlers will be called asynchronously by the browser, as they occur.

If an error in the communication happens, we definitely won't be able to get our data from the server, so the honest thing to do is to reject our promise:

```
request.onerror = function(){
  reject(this.status + " " + this.statusText);
};
```

If we receive a response from the server, we have to analyze that response and consider the exact situation. Without going into too much detail, a server can respond with various things, but in this case, we care only that the response is successful (status 200). If it isn't, again we reject the promise.

Even if the server has successfully responded with data, this still doesn't mean that we're in the clear. Because our goal was to get JSON-formatted objects from the server, the JSON code could always have syntax errors. This is why, when calling the JSON.parse method, we surround the code with a try-catch statement. If an exception occurs while parsing the server response, we also reject the promise. With this, we've taken care of all bad scenarios that can happen.

If everything goes according to plan, and we successfully obtain our objects, we can safely resolve the promise. Finally, we can use our getJSON function to fetch ninjas from the server:

```
getJSON("data/ninjas.json").then(ninjas => {
  assert(ninjas !== null, "Ninjas obtained!");
}).catch(e => fail("Shouldn't be here:" + e));
```

In this case, we have three potential sources of errors: errors in establishing the communication between the server and the client, the server responding with unanticipated data (invalid response status), and invalid JSON code. But from the perspective of the code that uses the getJSON function, we don't care about the specifics of error sources. We only supply a callback that gets triggered if everything goes okay and the data is properly received, and a callback that gets triggered if any error occurs. This makes our lives as developers so much easier.

Now we're going to take it up a notch and explore another big advantage of promises: their elegant composition. We'll start by chaining several promises in a series of distinct steps.

6.3.5 *Chaining promises*

You've already seen how handling a sequence of interdependent steps leads to the pyramid of doom, a deeply nested and difficult-to-maintain sequence of callbacks. Promises are a step toward solving that problem, because they have the ability to be chained.

Earlier in the chapter, you saw how, by using the then method on a promise, we can register a callback that will be executed if a promise is successfully resolved. What we didn't tell you is that calling the then method also returns a new promise. So

there's nothing stopping us from chaining as many then methods as we want; see the following code.

Listing 6.16   Chaining promises with `then`

```
getJSON("data/ninjas.json")
 .then(ninjas => getJSON(ninjas[0].missionsUrl))
 .then(missions => getJSON(missions[0].detailsUrl))
 .then(mission => assert(mission !== null, "Ninja mission obtained!"))
 .catch(error => fail("An error has occurred"));
```

Catches promise rejections in any of the steps

Specifies multiple sequential steps by chaining in then calls

---

This creates a sequence of promises that will be, if everything goes according to plan, resolved one after another. First, we use the getJSON("data/ninjas.json") method to fetch a list of ninjas from the file on the server. After we receive that list, we take the information about the first ninja, and we request a list of missions the ninja is assigned to: getJSON(ninjas[0].missionsUrl). Later, when these missions come in, we make yet another request for the details of the first mission: getJSON(missions[0].details-Url). Finally, we log the details of the mission.

Writing such code using standard callbacks would result in a deeply nested sequence of callbacks. Identifying the exact sequence of steps wouldn't be easy, and God forbid we decide to add in an extra step somewhere in the middle.

CATCHING ERRORS IN CHAINED PROMISES

When dealing with sequences of asynchronous steps, an error can occur in any step. We already know that we either can provide a second, error callback to the then call, or can chain in a catch call that takes an error callback. When we care about only the success/failure of the entire sequence of steps, supplying each step with special error handling might be tedious. So, as shown in listing 6.16, we can take advantage of the catch method that you saw earlier:

```
...catch(error => fail("An error has occurred:" + err));
```

If a failure occurs in any of the previous promises, the catch method catches it. If no error occurs, the program flow continues through it, unobstructed.

Dealing with a sequence of steps is much nicer with promises than with regular callbacks, wouldn't you agree? But it's still not as elegant as it could be. We'll get to that soon, but first let's see how to use promises to take care of parallel asynchronous steps.

6.3.6   *Waiting for a number of promises*

In addition to helping us deal with sequences of interdependent, asynchronous steps, promises significantly reduce the burden of waiting for several independent asynchronous tasks. Let's revisit our example in which we want to, in parallel, gather information about the ninjas at our disposal, the intricacies of the plan, and the map of the

location where the plan will be set in motion. With promises, this is as simple as shown in the following listing.

Listing 6.17 Waiting for a number of promises with `Promise.all`

The result is an array of succeed values, in the order of passed-in promises.

```
Promise.all([getJSON("data/ninjas.json"),
             getJSON("data/mapInfo.json"),
             getJSON("data/plan.json")]).then(results => {
  const ninjas = results[0], mapInfo = results[1], plan = results[2];

  assert(ninjas !== undefined
 && mapInfo !== undefined && plan !== undefined,
        "The plan is ready to be set in motion!");
}).catch(error => {
  fail("A problem in carrying out our plan!");
});
```

The Promise.all method takes an array of promises, and creates a new promise that succeeds if all promises succeed, and fails if even one promise fails.

As you can see, we don't have to care about the order in which tasks are executed, and whether some of them have finished, while others didn't. We state that we want to wait for a number of promises by using the built-in `Promise.all` method. This method takes in an array of promises and creates a *new* promise that successfully resolves when all passed-in promises resolve, and rejects if even one of the promises fails. The succeed callback receives an array of succeed values, one for each of the passed-in promises, in order. Take a minute to appreciate the elegance of code that processes multiple parallel asynchronous tasks with promises.

The `Promise.all` method waits for all promises in a list. But at times we have numerous promises, but we care only about the first one that succeeds (or fails). Meet the `Promise.race` method.

RACING PROMISES

Imagine that we have a group of ninjas at our disposal, and that we want to give an assignment to the first ninja who answers our call. When dealing with promises, we can write something like the following listing.

Listing 6.18 Racing promises with `Promise.race`

```
Promise.race([getJSON("data/yoshi.json"),
             getJSON("data/hattori.json"),
             getJSON("data/hanzo.json")])
      .then(ninja => {
        assert(ninja !== null, ninja.name + " responded first");
      }).catch(error => fail("Failure!"));
```

It's simple as that. There's no need for manually tracking everything. We use the `Promise.race` method to take an array of promises and return a completely *new* promise that resolves or rejects as soon as the first of the promises resolves or rejects.

So far you've seen how promises work, and how we can use them to greatly simplify dealing with a series of asynchronous steps, either in series or in parallel. Although

the improvements, when compared to plain old callbacks in terms of error handling and code elegance, are great, promisified code still isn't on the same level of elegance as simple synchronous code. In the next section, the two big concepts that we've introduced in this chapter, *generators* and *promises*, come together to provide the simplicity of synchronous code with the nonblocking nature of asynchronous code.

6.4 *Combining generators and promises*

In this section, we'll combine generators (and their capability to pause and resume their execution) with promises, in order to achieve more elegant asynchronous code. We'll use the example of a functionality that enables users to get details of the highest-rated mission done by the most popular ninja. The data representing the ninjas, the summaries of their missions, as well as the details of the missions are stored on a remote server, encoded in JSON.

All of these subtasks are long-running and mutually dependent. If we were to implement them in a synchronous fashion, we'd get the following straightforward code:

```
try {
   const ninjas = syncGetJSON("data/ninjas.json");
   const missions = syncGetJSON(ninjas[0].missionsUrl);
   const missionDetails = syncGetJSON(missions[0].detailsUrl);
   //Study the mission description
} catch(e){
  //Oh no, we weren't able to get the mission details
}
```

Although this code is great for its simplicity and error handling, it blocks the UI, which results in unhappy users. Ideally, we'd like to change this code so that no blocking occurs during a long-running task. One way of doing this is by combining generators and promises.

As we know, yielding from a generator suspends the execution of the generator without blocking. To wake up the generator and continue its execution, we have to call the `next` method on the generator's iterator. Promises, on the other hand, allow us to specify a callback that will be triggered in case we were able to obtain the promised value, and a callback that will be triggered in case an error has occurred.

The idea, then, is to combine generators and promises in the following way: We put the code that uses asynchronous tasks in a generator, and we execute that generator function. When we reach a point in the generator execution that calls an asynchronous task, we create a promise that represents the value of that asynchronous task. Because we have no idea when that promise will be resolved (or even if it will be resolved), at this point of generator execution, we yield from the generator, so that we don't cause blocking. After a while, when the promise gets settled, we continue the execution of our generator by calling the iterator's `next` method. We do this as many times as necessary. See the following listing for a practical example.

Listing 6.19 Combining generators and promises

The function using asynchronous results should be able to pause while waiting for results. Notice the function*. We're using generators!

Yield on each asynchronous task.

```
async(function*(){
  try {
    const ninjas = yield getJSON("data/ninjas.json");
    const missions = yield getJSON(ninjas[0].missionsUrl);
    const missionDescription = yield getJSON(missions[0].detailsUrl);
    //Study the mission details
  }
  catch(e) {
    //Oh no, we weren't able to get the mission details
  }
});

function async(generator) {
  var iterator = generator();

  function handle(iteratorResult) {
    if(iteratorResult.done) { return; }

    const iteratorValue = iteratorResult.value;

    if(iteratorValue instanceof Promise) {
      iteratorValue.then(res => handle(iterator.next(res))
                     .catch(err => iterator.throw(err));
    }
  }

  try {
    handle(iterator.next());
  }
  catch (e) { iterator.throw(e); }
}
```

Defines a helper function that will control our generator

Defines the function that will handle each value generated by the generator

Stops when the generator has no more results

Restarts the generator execution.

We can still use all standard language constructs such as try-catch statements or loops.

Creates an iterator through which we'll control the generator

If the generated value is a promise, register a success and a failure callback. This is the asynchronous part. If the promise succeeds, great, resume the generator and send in the promised value. If there's an error, throw an exception to the generator.

The `async` function takes a generator, calls it, and creates an iterator that will be used to resume the generator execution. Inside the `async` function, we declare a `handle` function that handles one return value from the generator—one "iteration" of our iterator. If the generator result is a promise that gets resolved successfully, we use the iterator's `next` method to send the promised value back to the generator and resume the generator's execution. If an error occurs and the promise gets rejected, we throw that error to the generator by using the iterator's `throw` method (told you it would come in handy). We keep doing this until the generator says it's done.

NOTE This is a rough sketch, a minimum amount of code needed to combine generators and promises. We don't recommend that you use this code in production.

Now let's take a closer look at the generator. On the first invocation of the iterator's `next` method, the generator executes up to the first `getJSON("data/ninjas.json")`

call. This call creates a promise that will eventually contain the list of information about our `ninjas`. Because this value is fetched asynchronously, we have no idea how much time it will take the browser to get it. But we know one thing: We don't want to block the application execution while we're waiting. For this reason, at this moment of execution, the generator yields control, which pauses the generator, and returns the control flow to the invocation of the `handle` function. Because the yielded value is a `getJSON` promise, in the `handle` function, by using the `then` and `catch` methods of the promise, we register a success and an error callback, and continue execution. With this, the control flow leaves the execution of the `handle` function and the body of the `async` function, and continues after the call to the `async` function (in our case, there's no more code after, so it idles). During this time, our generator function patiently waits suspended, without blocking the program execution.

Much, much later, when the browser receives a response (either a positive or a negative one), one of the promise callbacks is invoked. If the promise was resolved successfully, the success callback is invoked, which in turn causes the execution of the iterator's `next` method, which asks the generator for another value. This brings back the generator from suspension and sends to it the value passed in by the callback. This means that we reenter the body of our generator, after the first `yield` expression, whose value becomes the `ninjas` list that was asynchronously fetched from the server. The execution of the generator function continues, and the value is assigned to the `plan` variable.

In the next line of the generator, we use some of the obtained data, `ninjas[0] .missionUrl`, to make another `getJSON` call that creates another promise that should eventually contain a list of missions done by the most popular ninja. Again, because this is an asynchronous task, we have no idea how long it's going to take, so we again yield the execution and repeat the whole process.

This process is repeated as long as there are asynchronous tasks in the generator.

This was a tad on the complex side, but we like this example because it combines a lot of things that you've learned so far:

- *Functions as first-class objects*—We send a function as an argument to the `async` function.
- *Generator functions*—We use their ability to suspend and resume execution.
- *Promises*—They help us deal with asynchronous code.
- *Callbacks*—We register success and failure callbacks on our promises.
- *Arrow functions*—Because of their simplicity, for callbacks we use arrow functions.
- *Closures*—The iterator, through which we control the generator, is created in the `async` function, and we access it, through closures, in the promise callbacks.

Now that we've gone through the whole process, let's take a minute to appreciate how much more elegant the code that implements our business logic is. Consider this:

```
getJSON("data/ninjas.json", (err, ninjas) => {
  if(err) { console.log("Error fetching ninjas", err); return; }

  getJSON(ninjas[0].missionsUrl, (err, missions) => {
    if(err) { console.log("Error locating ninja missions", err); return; }
    console.log(misssions);
  })
});
```

Instead of mixed control-flow and error handling, and slightly confusing code, we end up with something like this:

```
async(function*() {
  try {
    const ninjas = yield getJSON("data/ninjas.json");
    const missions = yield getJSON(ninjas[0].missionsUrl);

    //All information recieved
  }
  catch(e) {
    //An error has occurred
  }
});
```

This end result combines the advantages of synchronous and asynchronous code. From synchronous code, we have the ease of understanding, and the ability to use all standard control-flow and exception-handling mechanisms such as loops and `try-catch` statements. From asynchronous code, we get the nonblocking nature; the execution of our application isn't blocked while waiting for long-running asynchronous tasks.

6.4.1 *Looking forward—the async function*

Notice that we still had to write some boilerplate code; we had to develop an `async` function that takes care of handling promises and requesting values from the generator. Although we can write this function only once and then reuse it throughout our code, it would be even nicer if we didn't have to think about it. The people in charge of JavaScript are well aware of the usefulness of the combination of generators and promises, and they want to make our lives even easier by building in direct language support for mixing generators and promises.

For these situations, the current plan is to include two new keywords, `async` and `await`, that would take care of this boilerplate code. Soon, we'll be able to write something like this:

```
(async function (){
  try {
    const ninjas = await getJSON("data/ninjas.json");
    const missions = await getJSON(missions[0].missionsUrl);

    console.log(missions);
  }
```

```
  catch(e) {
    console.log("Error: ", e);
  }
})()
```

We use the `async` keyword in front of the function keyword to specify that this func-
tion relies on asynchronous values, and at every place where we call an asynchronous
task, we place the `await` keyword that says to the JavaScript engine, please wait for this
result without blocking. In the background, everything happens as we've discussed
previously throughout the chapter, but now we don't need to worry about it.

> **NOTE**   Async functions will appear in the next installment of JavaScript. Cur-
> rently no browser supports it, but you can use transpilers such as Babel or Tra-
> ceur if you wish to use async in your code today.

6.5   *Summary*

- Generators are functions that generate sequences of values—not all at once,
  but on a per request basis.
- Unlike standard functions, generators can suspend and resume their execu-
  tion. After a generator has generated a value, it suspends its execution without
  blocking the main thread and patiently waits for the next request.
- A generator is declared by putting an asterisk (`*`) after the function keyword.
  Within the body of the generator, we can use the new `yield` keyword that yields
  a value and suspends the execution of the generator. If we want to yield to
  another generator, we use the `yield*` operator.
- Calling a generator creates an iterator object through which we control the exe-
  cution of the generator. We request new values from the generator by using the
  iterator's `next` method, and we can even throw exceptions into the generator by
  calling the iterator's `throw` method. In addition, the `next` method can be used
  to send in values to the generator.
- A promise is a placeholder for the results of a computation; it's a guarantee that
  eventually we'll know the result of the computation, most often an asynchro-
  nous computation. A promise can either succeed or fail, and after it has done
  so, there will be no more changes.
- Promises significantly simplify our dealings with asynchronous tasks. We can
  easily work with sequences of interdependent asynchronous steps by using the
  `then` method to chain promises. Parallel handling of multiple asynchronous
  steps is also greatly simplified; we use the `Promise.all` method.
- We can combine generators and promises to deal with asynchronous tasks with
  the simplicity of synchronous code.

6.6   *Exercises*

1 After running the following code, what are the values of variables a1 to a4?

```
function *EvenGenerator(){
  let num = 2;
  while(true){
    yield num;
    num = num + 2;
  }
}

let generator = EvenGenerator();

let a1 = generator.next().value;
let a2 = generator.next().value;
let a3 = EvenGenerator().next().value;
let a4 = generator.next().value;
```

2 What's the content of the ninjas array after running the following code? (Hint: think about how the for-of loop can be implemented with a while loop.)

```
function* NinjaGenerator(){
  yield "Yoshi";
  return "Hattori";
  yield "Hanzo";
}

var ninjas = [];
for(let ninja of NinjaGenerator()){
  ninjas.push(ninja);
}

ninjas;
```

3 What are the values of variables a1 and a2, after running the following code?

```
function *Gen(val){
  val = yield val * 2;
  yield val;
}

let generator = Gen(2);
let a1 = generator.next(3).value;
let a2 = generator.next(4).value;
```

4 What's the output of the following code?

```
const promise = new Promise((resolve, reject) => {
  reject("Hattori");
});

promise.then(val => alert("Success: " + val))
       .catch(e => alert("Error: " + e));
```

5 What's the output of the following code?

```
const promise = new Promise((resolve, reject) => {
  resolve("Hattori");
  setTimeout(()=> reject("Yoshi"), 500);
});

promise.then(val => alert("Success: " + val))
       .catch(e => alert("Error: " + e));
```

Part 3

Digging into objects and fortifying your code

Now that you've learned the ins and outs of functions, we'll continue our exploration of JavaScript by taking a closer look at object fundamentals in chapter 7.

In chapter 8, we'll study how to control access to and monitor our objects with getters and setters, and with proxies, a completely new type of object in JavaScript.

We'll take a look at collections in chapter 9—traditional ones such as arrays, as well as completely new types such as maps and sets.

From there, we'll move on to regular expressions in chapter 10. You'll learn that many tasks that used to take reams of code to accomplish can be condensed to a mere handful of statements through the proper use of JavaScript regular expressions.

Finally, in chapter 11, we'll show you how to structure your JavaScript applications into smaller, well-organized units of functionality called modules.

Object orientation
with prototypes

7

This chapter covers
- Exploring prototypes
- Using functions as constructors
- Extending objects with prototypes
- Avoiding common gotchas
- Building classes with inheritance

You've learned that functions are first-class objects in JavaScript, that closures make them incredibly versatile and useful, and that you can combine generator functions with promises to tackle the problem of asynchronous code. Now we're ready to tackle another important aspect of JavaScript: object prototypes.

A *prototype* is an object to which the search for a particular property can be delegated to. Prototypes are a convenient means of defining properties and functionality that will be automatically accessible to other objects. Prototypes serve a similar purpose to that of classes in classical object-oriented languages. Indeed, the main use of prototypes in JavaScript is in producing code written in an object-oriented

167

way, similar to, but not exactly like, code in more conventional, class-based languages such as Java or C#.

In this chapter, we'll delve into how prototypes work, study their connection with constructor functions, and see how to mimic some of the object-oriented features often used in other, more conventional object-oriented languages. We'll also explore a new addition to JavaScript, the `class` keyword, which doesn't exactly bring full-featured classes to JavaScript but does enable us to easily mimic classes and inheritance. Let's start exploring.

. . . .

Do you know?

How do you test whether an object has access to a particular property?

Why is a prototype chain important for working with objects in JavaScript?

Do ES6 classes change how JavaScript works with objects?

. . . .

7.1 Understanding prototypes

In JavaScript, objects are collections of named properties with values. For example, we can easily create new objects with object-literal notation:

```
let obj = {
  prop1: 1,               ⟵──┐  Assigns a simple value
  prop2: function(){},    ⟵──── Assigns a function
  prop3: {}               ⟵──┐
}                              Assigns another object
```

As we can see, object properties can be simple values (such as numbers or strings), functions, and even other objects. In addition, JavaScript is a highly dynamic language, and the properties assigned to an object can be easily changed by modifying and deleting existing properties:

```
obj.prop1 = 1;       ⟵──┐  prop1 stores a     Assigns a value of a
obj.prop1 = [];      ⟵───  simple number.     completely different
delete obj.prop2;    ⟵──┐                     type, here an array
                         Removes the property from the object
```

We can even add completely new properties:

```
obj.prop4 = "Hello";    ⟵──── Adds a completely new property
```

In the end, all these modifications have left our simple object in the following state:

```
{
  prop1: [],
  prop3: {},
  prop4: "Hello"
};
```

When developing software, we strive not to reinvent the wheel, so we want to reuse as much code as possible. One form of code reuse that also helps organize our programs is *inheritance*, extending the features of one object into another. In JavaScript, inheritance is implemented with prototyping.

The idea of prototyping is simple. Every object can have a reference to its *prototype*, an object to which the search for a particular property can be delegated to, if the object itself doesn't have that property. Imagine that you're in a game quiz with a group of people, and that the game show host asks you a question. If you know the answer, you give it immediately, and if you don't, you ask the person next to you. It's as simple as that.

Let's take a look at the following listing.

Listing 7.1 With prototypes, objects can access properties of other objects

```
const yoshi = { skulk: true };          Creates three
const hattori = { sneak: true };        objects, each with
const kuma = { creep: true };           its own property

assert("skulk" in yoshi, "Yoshi can skulk");            yoshi has access to
assert(!("sneak" in yoshi)), "Yoshi cannot sneak");     only its own, skulk,
assert(!("creep" in yoshi)), "Yoshi cannot creep");     property.

Object.setPrototypeOf(yoshi, hattori);

assert("sneak" in yoshi, "Yoshi can now sneak");        By setting hattori as
assert(!("creep" in hattori)), "Hattori cannot creep"); yoshi's prototype,
                                                        yoshi now has
Object.setPrototypeOf(hattori, kuma);                   access to hattori's
assert("creep" in hattori, "Hattori can now creep");    properties.
assert("creep" in yoshi, "Yoshi can also creep");
```

Use the Object .setPrototypeOf method to set one object as the prototype of another object.

Currently, hattori can't creep.

Sets kuma as a prototype of hattori

yoshi has access to only its own, skulk, property.

By setting hattori as yoshi's prototype, yoshi now has access to hattori's properties.

Now hattori has access to creep.

yoshi also has access to creep, through hattori.

In this example, we start by creating three objects: `yoshi`, `hattori`, and `kuma`. Each has one specific property accessible only to that object: Only `yoshi` can `skulk`, only `hattori` can `sneak`, and only `kuma` can `creep`. See figure 7.1.

```
const yoshi = { skulk: true };
const hattori = { sneak: true };
const kuma = { creep: true };
```

yoshi	hattori	kuma
skulk: true	sneak: true	creep: true
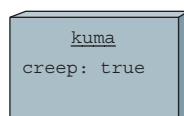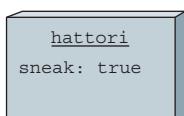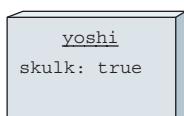
Figure 7.1 Initially, each object has access to only its own properties.

To test whether an object has access to a particular property, we can use the `in` operator. For example, executing `skulk in yoshi` returns `true`, because `yoshi` has access to the `skulk` property; whereas executing `sneak in yoshi` returns `false`.

In JavaScript, the object's prototype property is an internal property that's not directly accessible (so we mark it with `[[prototype]]`). Instead, the built-in method `Object.setPrototypeOf` takes in two object arguments and sets the second object as the prototype of the first. For example, calling `Object.setPrototypeOf(yoshi, hattori);` sets up `hattori` as a prototype of `yoshi`.

As a result, whenever we ask `yoshi` for a property that it doesn't have, `yoshi` delegates that search to `hattori`. We can access `hattori`'s `sneak` property through `yoshi`. See figure 7.2.

We can do a similar thing with `hattori` and `kuma`. By using the `Object.setPrototypeOf` method, we can set `kuma` as the prototype of `hattori`. If we then ask `hattori` for a property that he doesn't have, that search will be delegated to `kuma`. In this case, `hattori` now has access to `kuma`'s `creep` property. See figure 7.3.

It's important to emphasize that every object can have a prototype, and an object's prototype can also have a prototype, and so on, forming a *prototype chain*. The search delegation for a particular property occurs up the whole chain, and it stops only when there are no more prototypes to explore. For example, as shown in figure 7.3, asking `yoshi` for the value of the `creep` property triggers the search for the property first in `yoshi`. Because the property isn't found, `yoshi`'s prototype, `hattori`, is searched. Again, `hattori` doesn't have a property named `creep`, so `hattori`'s prototype, `kuma`, is searched, and the property is finally found.

```
const yoshi = { skulk: true };
const hattori = { sneak: true };
const kuma = { creep: true };

Object.setPrototypeOf(yoshi, hattori);
```

```
      yoshi                    hattori                    kuma
  skulk: true              sneak: true               creep: true
  [[prototype]]
```

**Accessing the property
sneak on yoshi will return
true, even though yoshi
doesn't have a sneak
property.**

**hattori is set as
yoshi's prototype.**
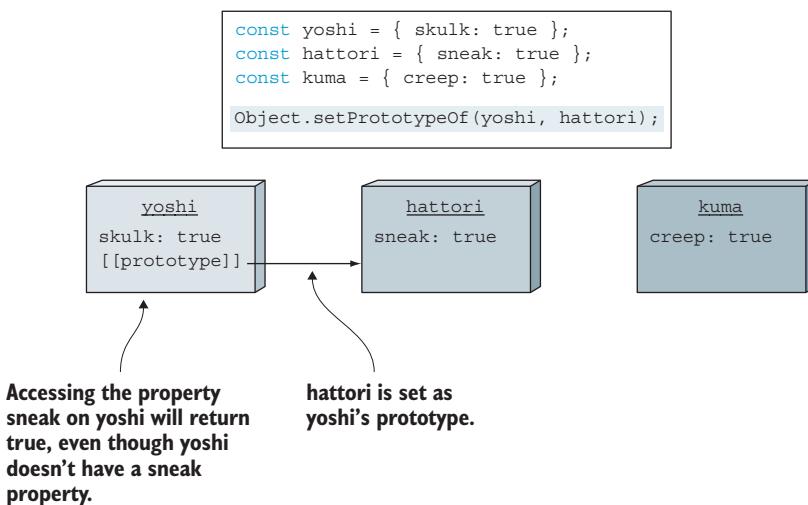
Figure 7.2   When we access a property that the object doesn't have, the object's prototype is searched for that property. Here, we can access `hattori`'s `sneak` property through `yoshi`, because `yoshi` is `hattori`'s prototype.

```
const yoshi = { skulk: true };
const hattori = { sneak: true };
const kuma = { creep: true };

Object.setPrototypeOf(yoshi, hattori);
Object.setPrototypeOf(hattori, kuma);
```

```
         yoshi                  hattori                  kuma
    skulk: true            sneak: true             creep: true
    [[prototype]]  ───▶    [[prototype]]  ───▶
```

**Accessing the property creep
on yoshi will return true.**
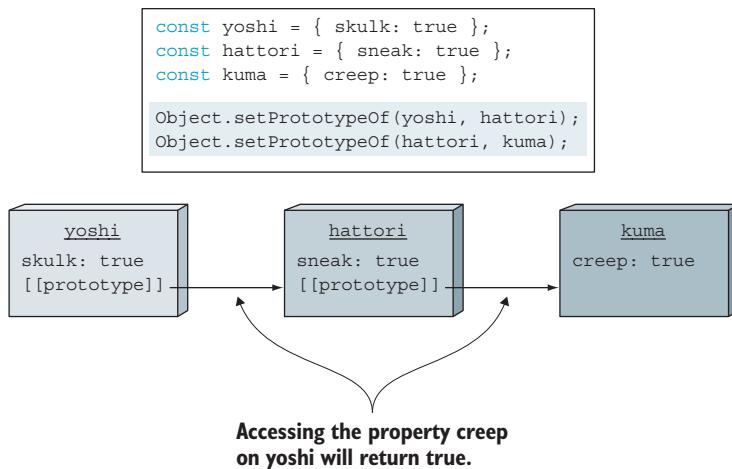
**Figure 7.3 The search for a particular property stops when there are no more prototypes to explore.
Accessing `yoshi.creep` triggers the search first in `yoshi`, then in `hattori`, and finally in `kuma`.**

Now that we have a basic idea of how the search for a particular property occurs
through the prototype chain, let's see how prototypes are used when constructing new
objects with constructor functions.

7.2 *Object construction and prototypes*

The simplest way to create a new object is with a statement like this:

```
const warrior = {};
```

This creates a new and empty object, which we can then populate with properties via
assignment statements:

```
const warrior = {};
warrior.name = 'Saito';
warrior.occupation = 'marksman';
```

But those coming from an object-oriented background might miss the encapsulation
and structuring that comes with a class constructor, a function that serves to initialize
an object to a known initial state. After all, if we're going to create multiple instances
of the same type of object, assigning the properties individually isn't only tedious but
also highly error-prone. We'd like to be able to consolidate the set of properties and
methods for a class of objects in one place.

JavaScript provides such a mechanism, though in a different form than most other
languages. Like object-oriented languages such as Java and C++, JavaScript employs
the `new` operator to instantiate new objects via constructors, but there's no true class
definition in JavaScript. Instead, the `new` operator, applied to a constructor function
(as you saw in chapter 3), triggers the creation of a newly allocated object.

What we didn't learn in the previous chapters was that every function has a prototype object that's automatically set as the prototype of the objects created with that function. Let's see how that works in the following listing.

Listing 7.2 Creating a `new` instance with a prototyped method

```
function Ninja(){}
Ninja.prototype.swingSword = function(){
  return true;
};

const ninja1 = Ninja();
assert(ninja1 === undefined,
       "No instance of Ninja created.");

const ninja2 = new Ninja();
assert(ninja2 &&
       ninja2.swingSword &&
       ninja2.swingSword(),
       "Instance exists and method is callable." );
```

Defines a function that does nothing and returns nothing

Every function has a built-in prototype object, which we can freely modify.

Calls the function as a function. Testing confirms that nothing at all seems to happen.

Calls the function as a constructor. Testing confirms that not only is a new object instance created, but it possesses the method from the prototype of the function.

In this code, we define a seemingly do-nothing function named `Ninja` that we'll invoke in two ways: as a "normal" function, `const ninja1 = Ninja();` and as a constructor, `const ninja2 = new Ninja();`.

When the function is created, it immediately gets a new object assigned to its prototype object, an object that we can extend just like any other object. In this case, we add a `swingSword` method to it:

```
Ninja.prototype.swingSword = function(){
  return true;
};
```

Then we put the function through its paces. First we call the function normally and store its result in variable `ninja1`. Looking at the function body, we see that it returns no value, so we'd expect `ninja1` to test as `undefined`, which we assert to be `true`. As a simple function, `Ninja` doesn't appear to be all that useful.

Then we call the function via the `new` operator, invoking it as a *constructor*, and something completely different happens. The function is once again called, but this time a newly allocated object has been created and set as the context of the function (and is accessible through the `this` keyword). The result returned from the `new` operator is a reference to this new object. We then test that `ninja2` has a reference to the newly created object, and that that object has a `swingSword` method that we can call. See figure 7.4 for a glimpse of the current application state.

```
function Ninja(){}
Ninja.prototype.swingSword = function(){
  return true;
};
…
const ninja2 = new Ninja();
```

Every function has a prototype object.

```
function Ninja(){}
prototype:
```

A function's prototype has a constructor property that references back to the function.

```
ninja2

[[prototype]]
```

```
Ninja prototype
constructor
swingSword
```

```
function(){}
```

The constructor object's prototype is set as the prototype of the newly created object.

Figure 7.4 Every function, when created, gets a new prototype object. When we use a function as a constructor, the constructed object's prototype is set to the function's prototype.

As you can see, a function, when created, gets a new object that's assigned to its prototype property. The prototype object initially has only one property, `constructor`, that references back to the function (we'll revisit the `constructor` property later).

When we use a function as a constructor (for example, by calling `new Ninja()`), the prototype of the newly constructed object is set to the object referenced by the constructor function's prototype.

In this example, we've extended the `Ninja.prototype` with the `swingSword` method, and when the `ninja2` object is created, its `prototype` property is set to `Ninja`'s prototype. Therefore, when we try to access the `swingSword` property on `ninja2`, the search for that property is delegated to the `Ninja` prototype object. Notice that *all* objects created with the `Ninja` constructor will have access to the `swingSword` method. Now that's code reuse!

The `swingSword` method is a property of the `Ninja`'s prototype, and not a property of `ninja` instances. Let's explore this difference between instance properties and prototype properties.

7.2.1 *Instance properties*

When the function is called as a constructor via the `new` operator, its context is defined as the new object instance. In addition to exposing properties via the prototype, we can

initialize values within the constructor function via the `this` parameter. Let's examine the creation of such instance properties in the next listing.

Listing 7.3   Observing the precedence of initialization activities

```
function Ninja(){
 this.swung = false;
 this.swingSword = function(){
   return !this.swung;
 };
}
Ninja.prototype.swingSword = function(){
  return this.swung;
};

const ninja = new Ninja();
assert(ninja.swingSword(),
     "Called the instance method, not the prototype method.");
```

Creates an instance variable that holds a Boolean value initialized to false

Creates an instance method that returns the inverse of the swung instance variable value

Defines a prototype method with the same name as the instance method. Which will take precedence?

Constructs a Ninja instance and asserts that the instance method will override the prototype method of the same name

Listing 7.3 is similar to the previous example in that we define a `swingSword` method by adding it to the `prototype` property of the constructor:

```
Ninja.prototype.swingSword = function(){
   return this.swung;
};
```

But we also add an identically named method within the constructor function itself:

```
function Ninja(){
  this.swung = false;
  this.swingSword = function(){
    return !this.swung;
  };
}
```

The two methods are defined to return opposing results so we can tell which will be called.

> **NOTE**   This isn't anything we'd advise doing in real-world code; quite the opposite. We're doing it here just to demonstrate the precedence of properties.

When you run the test, you see that it passes! This shows that instance members will hide properties of the same name defined in the prototype. See figure 7.5.

```
function Ninja(){
  this.swung = false;
  this.swingSword = function(){
    return !this.swung;
  };
}
Ninja.prototype.swingSword = function(){
  return this.swung;
};
const ninja = new Ninja();
```
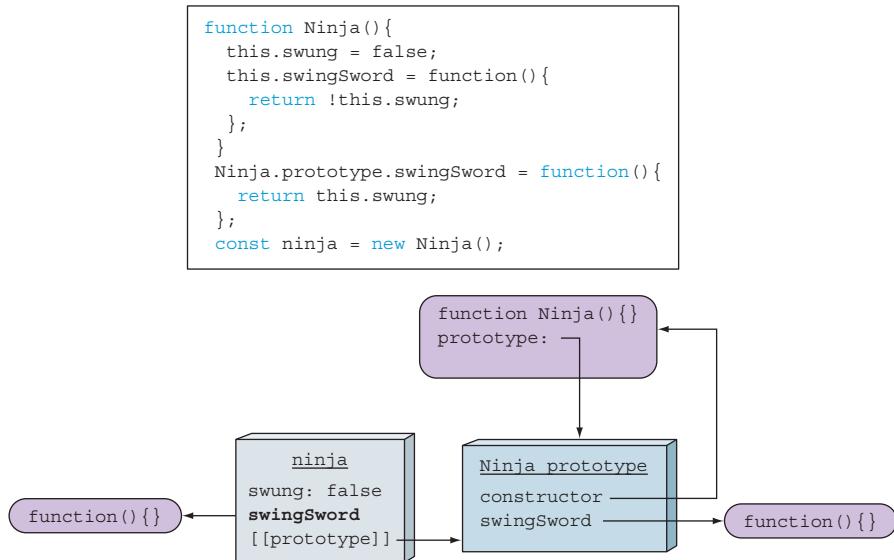

Figure 7.5 If a property can be found on the instance itself, the prototype isn't even consulted!

Within the constructor function, the `this` keyword refers to the newly created object, so the properties added within the constructor are created directly on the new `ninja` instance. Later, when we access the property `swingSword` on `ninja`, there's no need to traverse the prototype chain (as shown in figure 7.4); the property created within the constructor is immediately found and returned (see figure 7.5).

This has an interesting side effect. Take a look at figure 7.6, which shows the state of the application if we create three `ninja` instances.
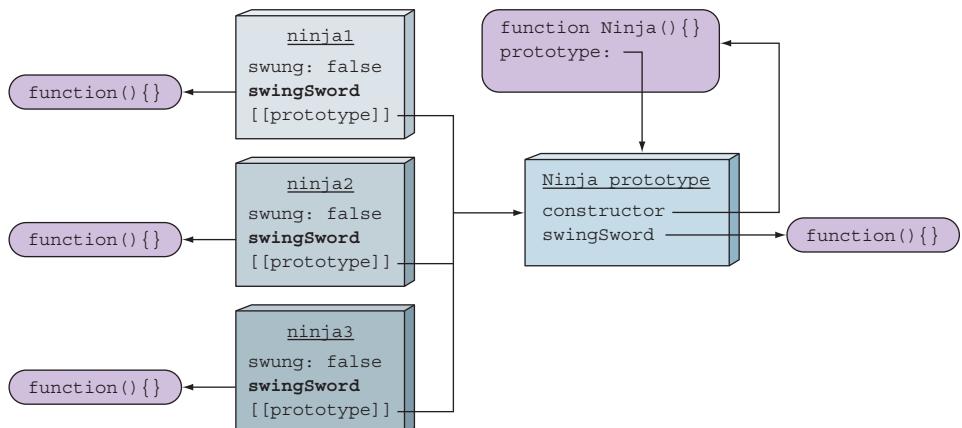
Figure 7.6 Every instance gets its own version of the properties created within the constructor, but they all have access to the same prototype's properties.

As you can see, every `ninja` instance gets its own version of the properties that were created within the constructor, while they all have access to the same prototype's properties. This is okay for value properties (for example, `swung`) that are specific to each object instance. But in certain cases it might be problematic for methods.

In this example, we'd have three versions of the `swingSword` method that all perform the same logic. This isn't a problem if we create a couple of objects, but it's something to pay attention to if we plan to create large numbers of objects. Because each method copy behaves the same, creating multiple copies often doesn't make sense, because it only consumes more memory. Sure, in general, the JavaScript engine might perform some optimizations, but that's not something to rely on. From that perspective, it makes sense to place object methods only on the function's prototype, because in that way we have a single method shared by all object instances.

> **NOTE**   Remember chapter 5 on closures: Methods defined within constructor functions allow us to mimic private object variables. If this is something we need, specifying methods within constructors is the only way to go.

7.2.2   *Side effects of the dynamic nature of JavaScript*

You've already seen that JavaScript is a dynamic language in which properties can be easily added, removed, and modified at will. The same thing holds for prototypes, both function prototypes and object prototypes. See the following listing.

Listing 7.4   With prototypes, everything can be changed at runtime

```
function Ninja(){                        Defines a constructor that
  this.swung = true;                     creates a Ninja with a single
}                                        Boolean property
                                                                        Creates an instance of
                                                                        Ninja by calling the
const ninja1 = new Ninja();                                             constructor function via
                                                                        the "new" operator

Ninja.prototype.swingSword = function(){   Adds a method to the
  return this.swung;                       prototype after the
};                                         object has been created
assert(ninja1.swingSword(),
       "Method exists, even out of order.");        Shows that the method
                                                     exists in the object

Ninja.prototype = {          Completely
 pierce: function() {        overrides the Ninja's
   return true;              prototype with a          Even though we've completely replaced
 }                           new object via the        the Ninja constructor's prototype, our
}                            pierce method             Ninja can still swing a sword, because
                                                       it keeps a reference to the old Ninja
assert(ninja1.swingSword(),                            prototype.
       "Our ninja can still swing!");

const ninja2 = new Ninja();                                      Newly created ninjas
assert(ninja2.pierce(),"Newly created ninjas can pierce");       reference the new
assert(!ninja2.swingSword, "But they cannot swing!");            prototype, so they can
                                                                 pierce but can't swing.
```

```
function Ninja(){
    this.swung = true;
}

const ninja1 = new Ninja();
```
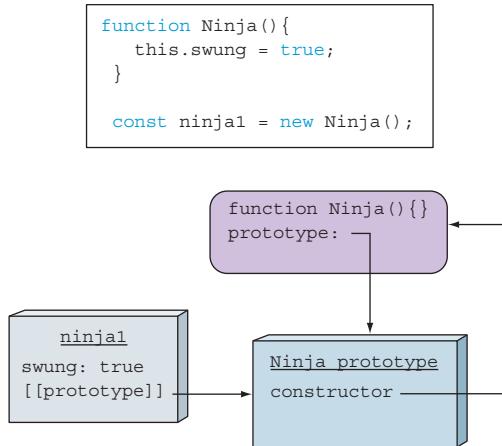

Figure 7.7 **After construction, `ninja1` has the property `swung`, and its prototype is the `Ninja` prototype that has only a `constructor` property.**

Here we again define a `Ninja` constructor and proceed to use it to create an object instance. The state of the application at this moment is shown in figure 7.7.

After the instance has been created, we add a `swingSword` method to the prototype. Then we run a test to show that the change we made to the prototype after the object was constructed takes effect. The current state of the application is shown in figure 7.8.

```
function Ninja(){
    this.swung = true;
}

const ninja1 = new Ninja();

Ninja.prototype.swingSword = function(){
    return this.swung;
};
```
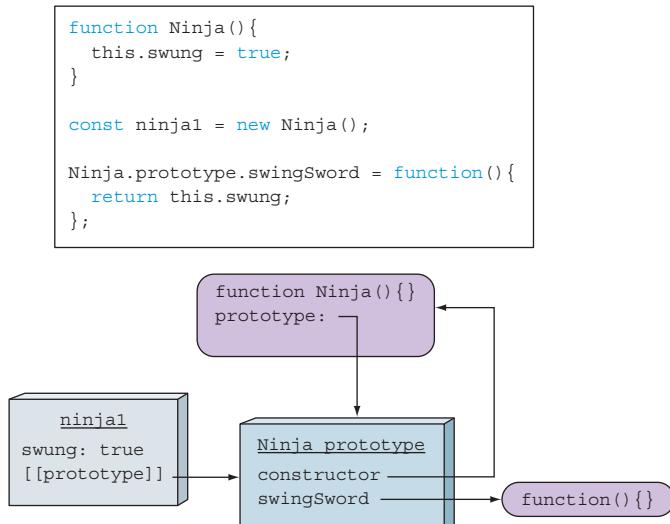

Figure 7.8 **Because the `ninja1` instance references the `Ninja` prototype, even changes made after the instance was constructed are accessible.**

```
function Ninja(){
    this.swung = true;
}

const ninja1 = new Ninja();

Ninja.prototype.swingSword = function(){
    return this.swung;
};
…
Ninja.prototype = {
  pierce: function() {
    return true;
  }
}
```

New prototype created (shown below as Ninja prototype′)
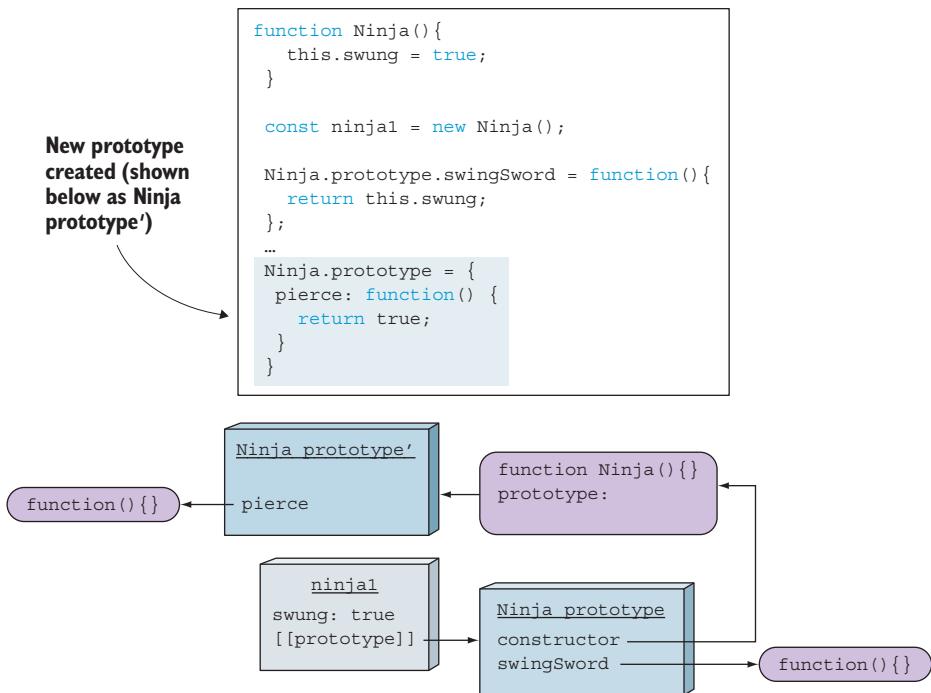
Figure 7.9   **The function's prototype can be replaced at will. The already constructed instances reference the old prototype!**

Later, we override the Ninja function's prototype by assigning it to a completely new object that has a pierce method. This results in the application state shown in figure 7.9.

As you can see, even though the Ninja function doesn't reference the old Ninja prototype, the old prototype is still kept alive by the ninja1 instance, which can still, through the prototype chain, access the swingSword method. But if we create new objects after this prototype switcheroo, the state of the application will be as shown in figure 7.10.

The reference between an object and the function's prototype is established at the time of object instantiation. Newly created objects will have a reference to the new prototype and will have access to the pierce method, whereas the old, pre-prototype-change objects keep their original prototype, happily swinging their swords.

We've explored how prototypes work and how they're related to object instantiation. Well done! Now take a quick breath, so we can continue onward by learning more about the nature of those objects.

```
function Ninja(){
   this.swung = true;
 }

 const ninja1 = new Ninja();

 Ninja.prototype.swingSword = function(){
   return this.swung;
 };
 …
 Ninja.prototype = {
  pierce: function() {
    return true;
  }
 }
 …
 const ninja2 = new Ninja();
```
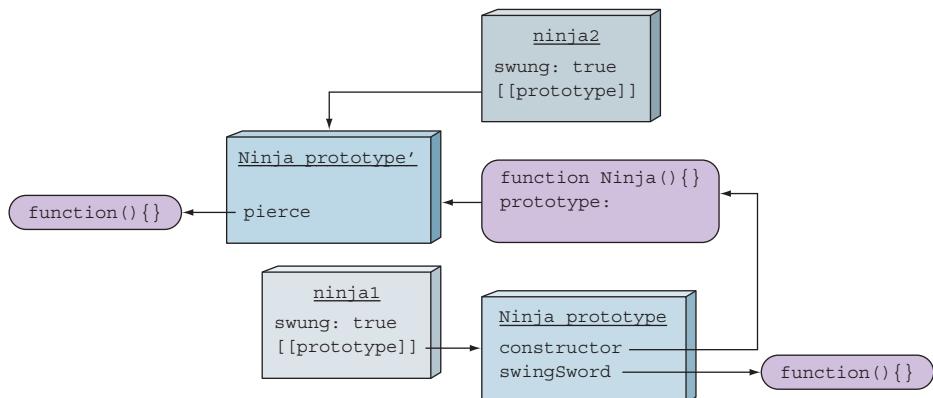

Figure 7.10   All newly created instances reference the new prototype.

7.2.3   *Object typing via constructors*

Although it's great to know how JavaScript uses the prototype to find the correct property references, it's also handy to know which function constructed an object instance. As you've seen earlier, the constructor of an object is available via the constructor property of the constructor function prototype. For example, figure 7.11 shows the state of the application when we instantiate an object with the Ninja constructor.

```
function Ninja(){}
const ninja1 = new Ninja();
```
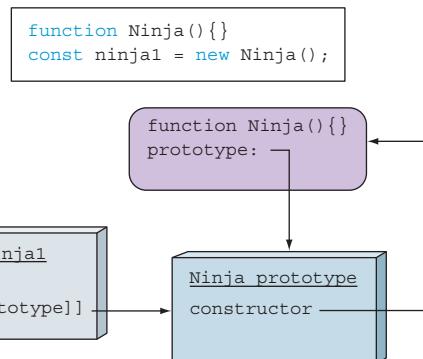

Figure 7.11   The prototype object of each function has a constructor property that references the function.

By using the `constructor` property, we can access the function that was used to create the object. This information can be used as a form of type checking, as shown in the next listing.

Listing 7.5   Examining the type of an instance and its constructor

```
function Ninja(){}
const ninja = new Ninja();

assert(typeof ninja === "object",
      "The type of the instance is object.");
assert(ninja instanceof Ninja,
       "instanceof identifies the constructor." );
assert(ninja.constructor === Ninja,
       "The ninja object was created by the Ninja function.");
```

Tests the type of ninja via typeof. This tells us it's an object, but not much else.

Tests the type of ninja via instanceof. This provides more information—that it was constructed from Ninja.

Tests the type of ninja via the constructor reference. This gives a reference to the constructor function.

We define a constructor and create an object instance using it. Then we examine the type of the instance by using the `typeof` operator. This doesn't reveal much, as all instances will be objects, thus always returning `object` as the result. Much more interesting is the `instanceof` operator, which gives us a way to determine whether an instance was created by a particular function constructor. You'll learn more about how the `instanceof` operator works later in the chapter.

In addition, we can use the `constructor` property, that we now know is accessible to all instances, as a reference to the original function that created it. We can use this to verify the origin of the instance (much as we can with the `instanceof` operator).

Additionally, because this is just a reference to the original constructor, we can instantiate a new `Ninja` object using it, as shown in the next listing.

Listing 7.6   Instantiating a new object using a reference to a constructor

```
function Ninja(){}

const ninja = new Ninja();
const ninja2 = new ninja.constructor();

assert(ninja2 instanceof Ninja, "It's a Ninja!");
assert(ninja !== ninja2, "But not the same Ninja!");
```

Constructs a second Ninja from the first

Proves the new object's Ninja-ness

They aren't the same object, but two distinct instances.

Here we define a constructor and create an instance using that constructor. Then we use the `constructor` property of the created instance to construct a second instance. Testing shows that a second `Ninja` has been constructed and that the variable doesn't merely point to the same instance.

What's especially interesting is that we can do this without even having access to the original function; we can use the reference completely behind the scenes, even if the original constructor is no longer in scope.

> **NOTE** Although the `constructor` property of an object can be changed, doing so doesn't have any immediate or obvious constructive purpose (though we might be able to think of some malicious ones). The property's reason for being is to indicate from where the object was constructed. If the `constructor` property is overwritten, the original value is lost.

That's all useful, but we've just scratched the surface of the superpowers that prototypes confer on us. Now things get interesting.

7.3 Achieving inheritance

Inheritance is a form of reuse in which new objects have access to properties of existing objects. This helps us avoid the need to repeat code and data across our code base. In JavaScript, inheritance works slightly differently than in other popular object-oriented languages. Consider the following listing, in which we attempt to achieve inheritance.

Listing 7.7 Trying to achieve inheritance with prototypes

```
function Person(){}                              Defines a dancing Person via a
Person.prototype.dance = function(){};           constructor and its prototype

function Ninja(){}          ◁——┐ Defines a Ninja      Attempts to make Ninja
Ninja.prototype = { dance: Person.prototype.dance }; ◁   a dancing Person by
                                                         copying the dance
const ninja = new Ninja();                               method from the
assert(ninja instanceof Ninja,                           Person prototype
       "ninja receives functionality from the Ninja prototype" );
assert(ninja instanceof Person, "... and the Person prototype" );
assert(ninja instanceof Object, "... and the Object prototype" );
```

Because the prototype of a function is an object, there are multiple ways of copying functionality (such as properties or methods) to effect inheritance. In this code, we define a `Person` and then a `Ninja`. And because a `Ninja` is clearly a person, we want `Ninja` to inherit the attributes of `Person`. We attempt to do so by copying the dance property of the `Person` prototype's method to a similarly named property in the `Ninja` prototype.

Running our test reveals that although we may have taught the ninja to dance, we failed to make the `Ninja` a `Person`, as shown in figure 7.12. We taught the `Ninja` to mimic the dance
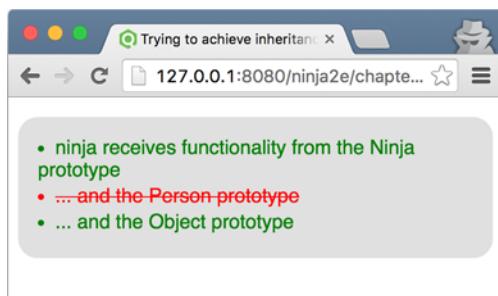
Figure 7.12 Our `Ninja` isn't really a `Person`. No happy dance!

of a person, but that hasn't *made* the `Ninja` a `Person`. That's not inheritance—it's just copying.

Apart from the fact that this approach isn't exactly working, we'd also need to copy each property of `Person` to the `Ninja` prototype individually. That's no way to do inheritance. Let's keep exploring.

What we really want to achieve is a *prototype chain* so that a `Ninja` can *be* a `Person`, and a `Person` can be a `Mammal`, and a `Mammal` can be an `Animal`, and so on, all the way to `Object`. The best technique for creating such a prototype chain is to use an instance of an object as the other object's prototype:

```
SubClass.prototype = new SuperClass();
```

For example:

```
Ninja.prototype = new Person();
```

This preserves the prototype chain, because the prototype of the `SubClass` instance will be an instance of the `SuperClass`, which has a prototype with all the properties of `Super-Class`, and which will in turn have a prototype pointing to an instance of *its* superclass, and on and on. In the next listing, we change listing 7.7 slightly to use this technique.

Listing 7.8   Achieving inheritance with prototypes

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();          ◁——  Makes a Ninja a Person by
                                                making the Ninja prototype
                                                an instance of Person
const ninja = new Ninja();
assert(ninja instanceof Ninja,
      "ninja receives functionality from the Ninja prototype");
assert(ninja instanceof Person, "... and the Person prototype");
assert(ninja instanceof Object, "... and the Object prototype");
assert(typeof ninja.dance === "function", "... and can dance!")
```

The only change to the code is to use an instance of `Person` as the prototype for `Ninja`. Running the tests shows that we've succeeded, as shown in figure 7.13. Now we'll take a closer look at the inner workings by looking at the state of the application after we've created the new `ninja` object, as shown in figure 7.14.

Figure 7.14 shows that when we define a `Person` function, a `Person`
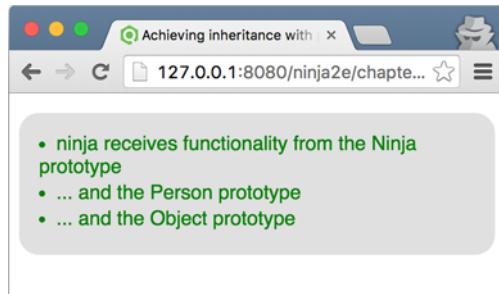
Figure 7.13   Our `Ninja` is a `Person`! Let the victory dance begin.

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();
const ninja = new Ninja();
```

Ninja.prototype = new Person() sets the prototype of the Ninja constructor to a newly created person.

ninja's prototype is a person; a ninja has access to all person methods.

Notice how we've lost our connection to the Ninja constructor.

The old prototype is left alone; nobody is referencing it. It will be deleted.

Figure 7.14 We've achieved inheritance by setting the prototype of the `Ninja` constructor to a new instance of a `Person` object.

prototype is also created that references the `Person` function through its `constructor` property. Normally, we can extend the `Person` prototype with additional properties, and in this case, we specify that every person, created with the `Person` constructor, has access to the `dance` method:

```
function Person(){}
Person.prototype.dance = function(){};
```

We also define a `Ninja` function that gets its own prototype object with a `constructor` property referencing the `Ninja` function: `function Ninja(){}`.

Next, in order to achieve inheritance, we replace the prototype of the `Ninja` function with a new `Person` instance. Now, when we create a new `Ninja` object, the internal prototype property of the newly created `ninja` object will be set to the object to which the current `Ninja` prototype property points to, the previously constructed `Person` instance:

```
function Ninja(){}
Ninja.prototype = new Person();
var ninja = new Ninja();
```

When we try to access the `dance` method through the `ninja` object, the JavaScript run-time will first check the `ninja` object itself. Because it doesn't have the `dance` property, its prototype, the `person` object, is searched. The `person` object also doesn't have the `dance` property, so its prototype is searched, and the property is finally found. This is how to achieve inheritance in JavaScript!

Here's the important implication: When we perform an `instanceof` operation, we can determine whether the function inherits the functionality of any object in its prototype chain.

> **NOTE**   Another technique that may have occurred to you, and that we advise strongly against, is to use the `Person` prototype object directly as the `Ninja` prototype, like this: `Ninja.prototype = Person.prototype`. Any changes to the `Ninja` prototype will then also change the `Person` prototype (because they're the same object), and that's bound to have undesirable side effects.

An additional happy side effect of doing prototype inheritance in this manner is that all inherited function prototypes will continue to live-update. Objects that inherit from the prototype always have access to the current prototype properties.

7.3.1   *The problem of overriding the constructor property*

If we take a closer look at figure 7.14, we'll see that by setting the new `Person` object as a prototype of the `Ninja` constructor, we've lost our connection to the `Ninja` constructor that was previously kept by the original `Ninja` prototype. This is a problem, because the `constructor` property can be used to determine the function with which the object was created. Somebody using our code could make a perfectly reasonable assumption that the following test will pass:

```
assert(ninja.constructor === Ninja,
       "The ninja object was created by the Ninja constructor");
```

But in the current state of the application, this test fails. As figure 7.14 shows, if we search the `ninja` object for the `constructor` property, we won't find it. So we go over to its prototype, which also doesn't have a constructor property, and again, we follow the prototype and end up in the prototype object of `Person`, which has a `constructor` property referencing the `Person` function. In effect, we get the wrong answer: If we ask the `ninja` object which function has constructed it, we'll get `Person` as the answer. This can be the source of some serious bugs.

It's up to us to fix this situation! But before we can do that, we have to take a detour and see how JavaScript enables us to configure properties.

CONFIGURING OBJECT PROPERTIES

In JavaScript, every object property is described with a *property descriptor* through which we can configure the following keys:

- `configurable`—If set to `true`, the property's descriptor can be changed and the property can be deleted. If set to `false`, we can do neither of these things.
- `enumerable`—If set to `true`, the property shows up during a `for-in` loop over the object's properties (we'll get to the `for-in` loop soon).
- `value`—Specifies the value of the property. Defaults to `undefined`.
- `writable`—If set to `true`, the property value can be changed by using an assignment.
- `get`—Defines the *getter* function, which will be called when we access the property. Can't be defined in conjunction with `value` and `writable`.
- `set`—Defines the *setter* function, which will be called whenever an assignment is made to the property. Also can't be defined in conjunction with `value` and `writable`.

Say we create a property through a simple assignment, for example:

```
ninja.name = "Yoshi";
```

This property will be configurable, enumerable, and writable, its value will be set to `Yoshi`, and functions `get` and `set` would be `undefined`.

When we want to fine-tune our property configuration, we can use the built-in `Object.defineProperty` method, which takes an object on which the property will be defined, the name of the property, and a property descriptor object. As an example, take a look at the following code.

Listing 7.9 Configuring properties

```
var ninja = {};
ninja.name = "Yoshi";
ninja.weapon = "kusarigama";

Object.defineProperty(ninja, "sneaky", {
  configurable: false,
  enumerable: false,
  value: true,
  writable: true
});

assert("sneaky" in ninja, "We can access the new property");

for(let prop in ninja){
  assert(prop !== undefined, "An enumerated property: " + prop);
}
```

Creates an empty object; uses assignments to add two properties

The built-in Object.defineProperty method is used to fine-tune the property configuration details.

Uses the for-in loop to iterate over ninja's enumerable properties

We start with the creation of an empty object, to which we add two properties: name and weapon, in the good old-fashioned way, by using assignments. Next, we use the built-in Object .defineProperty method to define the property sneaky, which isn't configurable, isn't enumerable, and has its value set to true. This value can be changed because it's writable.

Finally, we test that we can access the newly created sneaky property, and we use the for-in loop to go through all enumerable properties of the object. Figure 7.15 shows the result.

Figure 7.15 Properties name and weapon will be visited in the for-in loop, whereas our specially added sneaky property won't (even though we can access it normally).

By setting enumerable to false, we can be sure that the property won't appear when using the for-in loop. To understand why we'd want to do something like this, let's go back to the original problem.

FINALLY SOLVING THE PROBLEM OF OVERRIDING THE CONSTRUCTOR PROPERTY

When trying to extend Person with Ninja (or to make Ninja a subclass of Person), we ran into the following problem: When we set a new Person object as a prototype to the Ninja constructor, we lose the original Ninja prototype that keeps our constructor property. We don't want to lose the constructor property, because it's useful for determining the function used to create our object instances and it might be expected by other developers working on our code base.

We can solve this problem by using the knowledge that we've just obtained. We'll define a new constructor property on the new Ninja.prototype by using the Object.defineProperty method. See the following listing.

Listing 7.10   Fixing the constructor property problem

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();

Object.defineProperty(Ninja.prototype, "constructor", {
  enumerable: false,
  value: Ninja,
  writable: true
});

var ninja = new Ninja();
```

> We define a new non-enumerable constructor property pointing back to Ninja.

We've reestablished the connection.

```
assert(ninja.constructor === Ninja,
    "Connection from ninja instances to Ninja constructor
        reestablished!");
for(let prop in Ninja.prototype){
  assert(prop === "dance", "The only enumerable property is dance!");
}
```

We haven't added any enumerable properties to the Ninja.prototype.

Now if we run the code, we'll see that everything is peachy. We've reestablished the connection between `ninja` instances and the `Ninja` function, so we can know that they were constructed by the `Ninja` function. In addition, if anybody tries to loop through the properties of the `Ninja.prototype` object, we've made sure that our patched-on property `constructor` won't be visited. Now that's the mark of a true ninja; we went in, did our job, and got out, without anybody noticing anything from the outside!

7.3.2 *The instanceof operator*

In most programming languages, the straightforward approach for checking whether an object is a part of a class hierarchy is to use the `instanceof` operator. For example, in Java, the `instanceof` operator works by checking whether the object on the left side is either the same class or a subclass of the class type on the right.

Although certain parallels could be made with how the `instanceof` operator works in JavaScript, there's a little twist. In JavaScript, the `instanceof` operator works on the prototype chain of the object. For example, say we have the following expression:

```
ninja instanceof Ninja
```

The `instanceof` operator works by checking whether the *current* prototype of the `Ninja` function is in the prototype chain of the `ninja` instance. Let's go back to our persons and ninjas, for a more concrete example.

Listing 7.11 Studying the `instanceof` operator

```
function Person(){}
function Ninja(){}

Ninja.prototype = new Person();

const ninja = new Ninja();

assert(ninja instanceof Ninja, "Our ninja is a Ninja!");
assert(ninja instanceof Person, "A ninja is also a Person. ");
```

A ninja instance is both a Ninja and a Person.

As expected, a ninja is, at the same time, a `Ninja` and a `Person`. But, to nail down this point, figure 7.16 shows how the whole thing works behind the scenes.

```
function Person(){}
function Ninja(){}

Ninja.prototype = new Person();
const ninja = new Ninja();
```
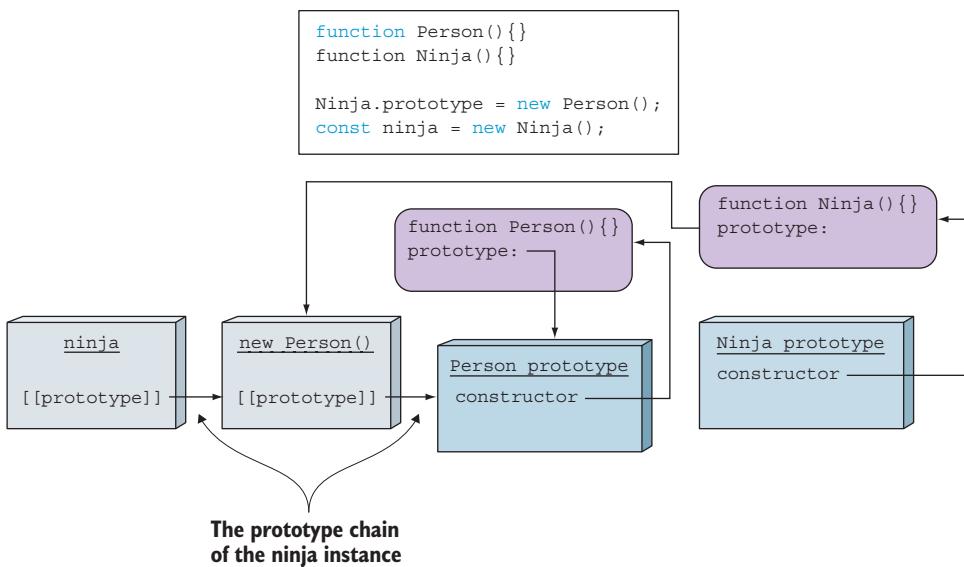

Figure 7.16 The prototype chain of a `ninja` instance is composed of a `new Person()` object and the `Person prototype`.

The prototype chain of a ninja instance is composed of a new Person() object, through which we've achieved inheritance, and the Person prototype. When evaluating the expression ninja instanceof Ninja, the JavaScript engine takes the prototype of the Ninja function, the new Person() object, and checks whether it's in the prototype chain of the ninja instance. Because the new Person() object is a direct prototype of the ninja instance, the result is true.

In the second case, where we check ninja instanceof Person, the JavaScript engine takes the prototype of the Person function, the Person prototype, and checks whether it can be found in the prototype chain of the ninja instance. Again, it can, because it's the prototype of our new Person() object, which, as we've already seen, is the prototype of the ninja instance.

And that's all there is to know about the instanceof operator. Although its most common use is in providing a clear way to determine whether an instance was created by a particular function constructor, it doesn't exactly work like that. Instead, it checks whether the prototype of the right-side function is in the prototype chain of the object on the left. Therefore, there is a caveat that we should be careful about.

THE INSTANCEOF CAVEAT

As you've seen multiple times throughout this chapter, JavaScript is a dynamic language in which we can modify a *lot* of things during program execution. For example, there's nothing stopping us from changing the prototype of a constructor, as shown in the following listing.

Listing 7.12   Watch out for changes to constructor prototypes

```
                 function Ninja(){}                    Even though our ninja instance was created by the
                                                       Ninja constructor, the instanceof operator now says
We change                                              that ninja isn't an instance of Ninja anymore!
the prototype    const ninja = new Ninja();
of the Ninja
constructor      assert(ninja instanceof Ninja, "Our ninja is a Ninja!");
function.
              ⯈  Ninja.prototype = {};

                 assert(!(ninja instanceof Ninja), "The ninja is now not a Ninja!?");  ◁
```

In this example, we again repeat all the basic steps of making a ninja instance, and
our first test goes fine. But if we change the prototype of the Ninja constructor func-
tion *after* the creation of the ninja instance, and again test whether ninja is an
instanceof Ninja, we'll see that the situation has changed. This will surprise us only
if we cling to the inaccurate assumption that the instanceof operator tells us whether
an instance was created by a particular function constructor. If, on the other hand, we
take the real semantics of the instanceof operator—that it checks only whether the
prototype of the function on the right side is in the prototype chain of the object on
the left side—we won't be surprised. This situation is shown in figure 7.17.
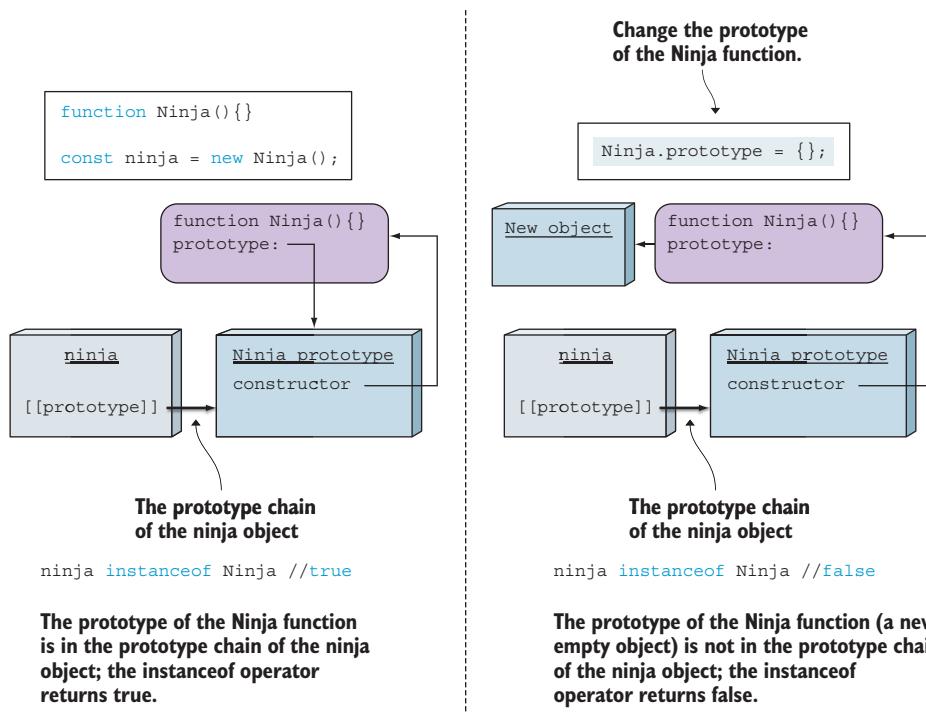
**Figure 7.17   The `instanceof` operator checks whether the prototype of the function on the right is in
the prototype chain of the object on the left. Be careful; the function's prototype can be changed anytime!**

Now that we understand how prototypes work in JavaScript, and how to use prototypes in conjunction with constructor functions to implement inheritance, let's move on to a new addition in the ES6 version of JavaScript: classes.

7.4 Using JavaScript "classes" in ES6

It's great that JavaScript lets us use a form of inheritance via prototypes. But many developers, especially those from a classical object-oriented background, would prefer a simplification or abstraction of JavaScript's inheritance system into one that they're more familiar with.

This inevitably leads toward the realm of classes, even though JavaScript doesn't support classical inheritance natively. As a response to this need, several JavaScript libraries that simulate classical inheritance have popped up. Because each library implements classes in its own way, the ECMAScript committee has standardized the syntax for simulating class-based inheritance. Notice how we said *simulating*. Even though now we can use the class keyword in JavaScript, the underlying implementation is still based on prototype inheritance!

> **NOTE**   The class keyword has been added to the ES6 version of JavaScript, and not all browsers implement it (see http://mng.bz/3ykA for current support).

Let's start by studying the new syntax.

7.4.1 Using the class keyword

ES6 introduces a new class keyword that provides a much more elegant way of creating objects and implementing inheritance than manually implementing it ourselves with prototypes. Using the class keyword is easy, as shown in the following listing.

Listing 7.13   Creating a class in ES6

```
class Ninja{
  constructor(name){
    this.name = name;
  }

  swingSword(){
    return true;
  }
}

var ninja = new Ninja("Yoshi");

assert(ninja instanceof Ninja, "Our ninja is a Ninja");
assert(ninja.name === "Yoshi", "named Yoshi");
assert(ninja.swingSword(), "and he can swing a sword");
```

Uses the class keyword to start specifying an ES6 class

Defines a constructor function that will be called when we call the class with the keyword new

Defines an additional method accessible to all Ninja instances

Instantiates a new ninja object with the keyword new

Tests for the expected behavior

Listing 7.13 shows that we can create a `Ninja` class by using the `class` keyword. When creating ES6 classes, we can explicitly define a `constructor` function that will be invoked when instantiating a `Ninja` instance. In the constructor's body, we can access the newly created instance with the `this` keyword, and we can easily add new properties, such as the `name` property. Within the class body, we can also define methods that will be accessible to all `Ninja` instances. In this case, we've defined a `swingSword` method that returns `true`:

```
class Ninja{
  constructor(name){
    this.name = name;
  }

  swingSword(){
    return true;
  }
}
```

Next we can create a `Ninja` instance by calling the `Ninja` class with the keyword `new`, just as we would if `Ninja` was a simple constructor function (as earlier in the chapter):

```
var ninja = new Ninja("Yoshi");
```

Finally, we can test that the `ninja` instance behaves as expected, that it's an `instanceof` `Ninja`, has a `name` property, and has access to the `swingSword` method:

```
assert(ninja instanceof Ninja, "Our ninja is a Ninja");
assert(ninja.name === "Yoshi", "named Yoshi");
assert(ninja.swingSword(), "and he can swing a sword");
```

CLASSES ARE SYNTACTIC SUGAR

As mentioned earlier, even though ES6 has introduced the `class` keyword, under the hood we're still dealing with good old prototypes; classes are syntactic sugar designed to make our lives a bit easier when mimicking classes in JavaScript.

Our class code from listing 7.13 can be translated to functionally identical ES5 code:

```
function Ninja(name) {
  this.name = name;
}
Ninja.prototype.swingSword = function() {
  return true;
};
```

As you can see, there's nothing especially new with ES6 classes. The code is more elegant, but the same concepts are applied.

In the previous examples, you saw how to define object methods (prototype methods), accessible to all object instances. In addition to such methods, classical object-oriented languages such as Java use static methods, methods defined on a class level. Check out the following example.

Listing 7.14   Static methods in ES6

```
class Ninja{
  constructor(name, level){
    this.name = name;
    this.level = level;
  }

  swingSword() {
    return true;
  }

  static compare(ninja1, ninja2){      Uses the static
    return ninja1.level - ninja2.level;  keyword to make
  }                                      a static method
}

var ninja1 = new Ninja("Yoshi", 4);
var ninja2 = new Ninja("Hattori", 3);           ninja instances
                                                don't have access
assert(!("compare" in ninja1) && !("compare" in ninja2),   to compare.
      "A ninja instance doesn't know how to compare");

assert(Ninja.compare(ninja1, ninja2) > 0,        The class Ninja
      "The Ninja class can do the comparison!");  has access to the
                                                  compare method.
assert(!("swingSword" in Ninja),
      "The Ninja class cannot swing a sword");
```

We again create a `Ninja` class that has a `swingSword` method accessible from all `ninja` instances. We also define a static method, `compare`, by prefixing the method name with the keyword `static`.

```
static compare(ninja1, ninja2){
    return ninja1.level - ninja2.level;
}
```

The `compare` method, which compares the skill levels of two ninjas, is defined on the class level, and not the instance level! Later we test that this effectively means that the `compare` method isn't accessible from `ninja` instances but is accessible from the `Ninja` class:

```
assert(!("compare" in ninja1) && !("compare" in ninja2),
       "The ninja instance doesn't know how to compare");
assert(Ninja.compare(ninja1, ninja2) > 0,
      "The Ninja class can do the comparison!");
```

We can also look at how "static" methods can be implemented in pre-ES6 code. For this, we have to remember only that classes are implemented through functions. Because static methods are class-level methods, we can implement them by taking advantage of functions as first-class objects, and adding a method property to our constructor function, as in the following example:

```
function Ninja(){}
Ninja.compare = function(ninja1, ninja2){...}
```

Extends the constructor function with a method to mimic static methods in pre-ES6 code

Now let's move on to inheritance.

7.4.2 *Implementing inheritance*

To be honest, performing inheritance in pre-ES6 code can be a pain. Let's go back to our trusted Ninjas, Persons example:

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();

Object.defineProperty(Ninja.prototype, "constructor", {
  enumerable: false,
  value: Ninja,
  writable: true
});
```

There's a lot to keep in mind here: Methods accessible to all instances should be added directly to the prototype of the constructor function, as we did with the `dance` method and the `Person` constructor. If we want to implement inheritance, we have to set the prototype of the derived "class" to the instance of the base "class." In this case, we assigned a new instance of `Person` to `Ninja.prototype`. Unfortunately, this messes up the `constructor` property, so we have to manually restore it with the `Object.defineProperty` method. This is a lot to keep in mind when trying to achieve a relatively simple and commonly used feature (inheritance). Luckily, with ES6, all of this is significantly simplified.

Let's see how it's done in the following listing.

Listing 7.15 Inheritance in ES6

```
class Person {
  constructor(name){
    this.name = name;
  }

  dance(){
    return true;
  }
```

```
}
class Ninja extends Person {
  constructor(name, weapon){
    super(name);
    this.weapon = weapon;
  }

  wieldWeapon(){
    return true;
  }
}

var person = new Person("Bob");

assert(person instanceof Person, "A person's a person");
assert(person.dance(), "A person can dance.");
assert(person.name === "Bob", "We can call it by name.");
assert(!(person instanceof Ninja), "But it's not a Ninja");
assert(!("wieldWeapon" in person), "And it cannot wield a weapon");


var ninja = new Ninja("Yoshi", "Wakizashi");
assert(ninja instanceof Ninja, "A ninja's a ninja");
assert(ninja.wieldWeapon(), "That can wield a weapon");
assert(ninja instanceof Person, "But it's also a person");
assert(ninja.name === "Yoshi" , "That has a name");
assert(ninja.dance(), "And enjoys dancing");
```

Uses the extends keyword to inherit from another class

Uses the super keyword to call the base class constructor

Listing 7.15 shows how to achieve inheritance in ES6; we use the `extends` keyword to inherit from another class:

```
class Ninja extends Person
```

In this example, we create a `Person` class with a constructor that assigns a `name` to each `Person` instance. We also define a `dance` method that will be accessible to all `Person` instances:

```
class Person {
  constructor(name){
    this.name = name;
  }
  dance(){
    return true;
  }
}
```

Next we define a `Ninja` class that extends the `Person` class. It has an additional `weapon` property, and a `wieldWeapon` method:

```
class Ninja extends Person {
  constructor(name, weapon){
    super(name);
```

```
    this.weapon = weapon;
  }

  wieldWeapon(){
    return true;
  }
}
```

In the constructor of the derived, `Ninja` class, there's a call to the constructor of the base, `Person` class, through the keyword `super`. This should be familiar, if you've worked with any class-based language.

  We continue by creating a `person` instance and checking that it's an instance of the `Person` class that has a `name` and can `dance`. Just to be sure, we also check that a person who *isn't* a Ninja can't wield a weapon:

```
var person = new Person("Bob");

assert(person instanceof Person, "A person's a person");
assert(person.dance(), "A person can dance.");
assert(person.name === "Bob", "We can call it by name.");
assert(!(person instanceof Ninja), "But it's not a Ninja");
assert(!("wieldWeapon" in person), "And it cannot wield a weapon");
```

We also create a `ninja` instance and check that it's an instance of `Ninja` and can wield a weapon. Because every `ninja` is also a `Person`, we check that a ninja is an instance of `Person`, that it has a `name`, and that it also, in the interim of fighting, enjoys dancing:

```
var ninja = new Ninja("Yoshi", "Wakizashi");
assert(ninja instanceof Ninja, "A ninja's a ninja");
assert(ninja.wieldWeapon(), "That can wield a weapon");
assert(ninja instanceof Person, "But it's also a person");
assert(ninja.name === "Yoshi" , "That has a name");
assert(ninja.dance(), "And enjoys dancing");
```

See how easy this is? There's no need to think about prototypes or the side effects of certain overridden properties. We define classes and specify their relationship by using the `extends` keyword. Finally, with ES6, hordes of developers coming from languages such as Java or C# can be at peace.

  And that's it. With ES6, we build class hierarchies almost as easily as in any other, more conventional object-oriented language.

7.5 Summary

- JavaScript objects are simple collections of named properties with values.
- JavaScript uses prototypes.
- Every object can have a reference to a *prototype*, an object to which we delegate the search for a particular property, if the object itself doesn't have the searched-for property. An object's prototype can have its own prototype, and so on, forming a *prototype chain*.

- We can define the prototype of an object by using the `Object.setPrototypeOf` method.
- Prototypes are closely linked to constructor functions. Every function has a `prototype` property that's set as the prototype of objects that it instantiates.
- A function's `prototype` object has a `constructor` property pointing back to the function itself. This property is accessible to all objects instantiated with that function and, with certain limitations, can be used to find out whether an object was created by a particular function.
- In JavaScript, almost everything can be changed at runtime, including an object's prototypes and a function's prototypes!
- If we want the instances created by a `Ninja` constructor function to "inherit" (more accurately, have access to) properties accessible to instances created by the `Person` constructor function, set the prototype of the `Ninja` constructor to a new instance of the `Person` class.
- In JavaScript, properties have attributes (configurable, enumerable, writable). These properties can be defined by using the built-in `Object.defineProperty` method.
- JavaScript ES6 adds support for a `class` keyword that enables us to more easily mimic classes. Behind the scenes, prototypes are still in play!
- The `extends` keyword enables elegant inheritance.

7.6   Exercises

1 Which of the following properties points to an object that will be searched if the target object doesn't have the searched-for property?

   a `class`

   b `instance`

   c `prototype`

   d `pointTo`

2 What's the value of variable `a1` after the following code is executed?

```
function Ninja(){}
Ninja.prototype.talk = function (){
  return "Hello";
};

const ninja = new Ninja();
const a1 = ninja.talk();
```

3 What's the value of `a1` after running the following code?

```
function Ninja(){}
Ninja.message = "Hello";

const ninja = new Ninja();

const a1 = ninja.message;
```

4 Explain the difference between the `getFullName` method in these two code fragments:

```
//First fragment
function Person(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;

  this.getFullName = function () {
    return this.firstName + " " + this.lastName;
  }
}

//Second fragment
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Person.prototype.getFullName = function () {
  return this.firstName + " " + this.lastName;
}
```

5 After running the following code, what will `ninja.constructor` point to?

```
function Person() { }
function Ninja() { }

const ninja = new Ninja();
```

6 After running the following code, what will `ninja.constructor` point to?

```
function Person() { }
function Ninja() { }
Ninja.prototype = new Person();
const ninja = new Ninja();
```

7 Explain how the `instanceof` operator works in the following example.

```
function Warrior() { }

function Samurai() { }
Samurai.prototype = new Warrior();

var samurai = new Samurai();

samurai instanceof Warrior; //Explain
```

8 Translate the following ES6 code into ES5 code.

```
class Warrior {
  constructor(weapon){
```

```
    this.weapon = weapon;
  }

  wield() {
    return "Wielding " + this.weapon;
  }

  static duel(warrior1, warrior2){
    return warrior1.wield() + " " + warrior2.wield();
  }
}
```

Controlling access to objects

8

This chapter covers

- Using getters and setters to control access to object properties
- Controlling access to objects through proxies
- Using proxies for cross-cutting concerns

In the previous chapter, you saw that JavaScript objects are dynamic collections of properties. We can easily add new properties, change the values of properties, and even completely remove existing properties. In many situations (for example, when validating property values, logging, or displaying data in the UI), we need to be able to monitor exactly what's going on with our objects. So in this chapter, you'll learn techniques for controlling access to and monitoring all of the changes that occur in your objects.

We'll start with getters and setters, methods that control access to specific object properties. You first saw these methods in action in chapters 5 and 7. In this chapter, you'll see some of their built-in language support and how to use them for logging, performing data validation, and defining computed properties.

We'll continue with proxies, a completely new type of object introduced in ES6. These objects control access to other objects. You'll learn how they work and how to use them to great effect to easily expand your code with cross-cutting concerns

such as performance measurement or logging, and how to avoid null exceptions by autopopulating object properties. Let's start the journey with something we already know to a certain degree: getters and setters.

. . . .

	What are some of the benefits of accessing a property's value through getters and setters?
Do you know?	**What is the main difference between proxies and getters and setters?**
	What are proxy traps? Name three types of trap.

. . . .

8.1   *Controlling access to properties with getters and setters*

In JavaScript, objects are relatively simple collections of properties. The primary way to keep track of our program state is by modifying those properties. For example, consider the following code:

```
function Ninja (level) {
  this.skillLevel = level;
}
const ninja = new Ninja(100);
```

Here we define a `Ninja` constructor that creates `ninja` objects with a property `skill-Level`. Later, if we want to change the value of that property, we can write the following code: `ninja.skillLevel = 20`.

That's all nice and convenient, but what happens in the following cases?

- We want to safeguard against accidental mistakes, such as assigning unanticipated data. For example, we want to stop ourselves from doing something like assigning a value of a wrong type: `ninja.skillLevel = "high"`.
- We want to log all changes to the `skillLevel` property.
- We need to show the value of our `skillLevel` property somewhere in the UI of our web page. Naturally, we want to present the last, up-to-date value of the property, but how can we easily do this?

We can handle all of these cases elegantly with getter and setter methods.

In chapter 5, you saw a glimpse of getters and setters as a means of mimicking private object properties in JavaScript through closures. Let's revisit the material you've learned so far, by working with ninjas that have a private `skillLevel` property accessible only through getters and setters, as shown in the following listing.

> **Listing 8.1   Using getters and setters to guard private properties**

```
function Ninja () {
  let skillLevel;                    ⟵——— Defines a private skillLevel variable
```

```
  this.getSkillLevel = () => skillLevel;
```

The getter method controls access to our private skillLevel variable.

```
  this.setSkillLevel = value => {
    skillLevel = value;
  };
}
```

The setter method controls the values we can assign to skillLevel.

```
const ninja = new Ninja();
ninja.setSkillLevel(100);
assert(ninja.getSkillLevel() === 100,
      "Our ninja is at level 100!");
```

Sets a new value of skillLevel through the setter method

Retrieves the value of skillLevel with the getter method

We define a `Ninja` constructor that creates ninjas with a "private" `skillLevel` variable accessible only through our `getSkillLevel` and `setSkillLevel` methods: The property value can be obtained only through the `getSkillLevel` method, whereas a new property value can be set only through the `setSkillLevel` method (remember chapter 5 on closures?).

   Now, if we want to log all read attempts of the `skillLevel` property, we expand the `getSkillLevel` method; and if we want to react to all write attempts, we expand the `setSkillLevel` method, as in the following snippet:

```
function Ninja () {
  let skillLevel;

  this.getSkillLevel = () => {
    report("Getting skill level value");
    return skillLevel;
  };

  this.setSkillLevel = value => {
    report("Modifying skillLevel property from:",
          skillLevel, "to: ", value);
    skillLevel = value;
  }
}
```

Using getters, we can know whenever code accesses a property.

Using setters, we can know whenever code wants to set a new value to a property.

This is great. We can easily react to all interactions with our properties by plugging in, for example, logging, data validation, or other side effects such as UI modifications.

   But one nagging concern might be popping into your mind. The `skillLevel` property is a value property; it references data (the number 100), and not a function. Unfortunately, in order to take advantage of all the benefits of controlled access, all our interactions with the property have to be made by explicitly calling the associated methods, which is, to be honest, slightly awkward.

   Luckily, JavaScript has built-in support for true getters and setters: properties that are accessed as normal data properties (for example, `ninja.skillLevel`), but that are methods that can compute the value of a requested property, validate the passed-in value, or whatever else we need them to do. Let's take a look at this built-in support.

8.1.1 Defining getters and setters

In JavaScript, getter and setter methods can be defined in two ways:

- By specifying them within object literals or within ES6 class definitions
- By using the built-in `Object.defineProperty` method

Explicit support for getters and setters has existed for quite some time now, since the days of ES5. As always, let's explore the syntax through an example. In this case, we have an object storing a list of ninjas, and we want to be able to get and set the first ninja in the list.

Listing 8.2 Defining getters and setters in object literals

```
const ninjaCollection = {
  ninjas: ["Yoshi", "Kuma", "Hattori"],
  get firstNinja(){
    report("Getting firstNinja");
    return this.ninjas[0];
  },
  set firstNinja(value){
    report("Setting firstNinja");
    this.ninjas[0] = value;
  }
};

assert(ninjaCollection.firstNinja === "Yoshi",
    "Yoshi is the first ninja");

ninjaCollection.firstNinja = "Hachi";

assert(ninjaCollection.firstNinja === "Hachi"
    && ninjaCollection.ninjas[0] ===  "Hachi",
      "Now Hachi is the first ninja");
```

> **Defines a getter method for the firstNinja property that returns the first ninja in our collection and logs a message**

> **Defines a setter method for the firstNinja property that modifies the first ninja in our collection and logs a message**

> **Accesses the firstNinja property as if it were a standard object property**

> **Modifies the firstNinja property as if it were a standard object property**

> **Tests that the property modification is stored**

This example defines a `ninjaCollection` object that has a standard property, `ninjas`, which references an array of ninjas, and a getter and a setter for the property `first-Ninja`. The general syntax for getters and setters is shown in figure 8.1.

Define a getter method by prefixing the property name with the get keyword.

Define a setter method by prefixing the property name with the set keyword.

Implicitly call the getter by reading the property value.

```
obj = {
  get name() {
    ...
  },
  set name(value) {
    ...
  }
};

obj.name

obj.name = "Yoshi"
```

A getter doesn't receive any arguments.

A setter receives one argument (the right side of an assignment expression).

Implicitly call the setter by assigning a value to a property.

Figure 8.1 The syntax for defining getters and setters. Prefix the property name with either the `get` or the `set` keyword.

As you can see, we define a getter property by prefixing the name with a `get` keyword, and a setter property with a `set` keyword.

In listing 8.2, both the getter and the setter log a message. In addition, the getter returns the value of the ninja at index `0`, and the setter assigns a new value to the ninja at the same index:

```
get firstNinja(){
  report("Getting firstNinja");
  return this.ninjas[0];
},
set firstNinja(value){
  report("Setting firstNinja");
  this.ninjas[0] = value;
}
```

Next, we test that accessing the getter property returns the first ninja, Yoshi:

```
assert(ninjaCollection.firstNinja === "Yoshi",
       "Yoshi is the first ninja");
```

Notice that the getter property is accessed as if it were a standard object property (and not as the method that it is).

After we access a getter property, the associated getter method is implicitly called, the message `Getting firstNinja` is logged, and the value of the ninja at index `0` is returned.

We continue by taking advantage of our setter method, and writing to the `first-Ninja` property, again, just as we would assign a new value to a normal object property:

```
ninjaCollection.firstNinja = "Hachi";
```

Similar to the previous case, because the `firstNinja` property has a setter method, whenever we assign a value to that property, the setter method is implicitly called. This logs the message `Setting firstNinja` and modifies the value of the ninja at index `0`.

Finally, we can test that our modification has done the work and that the new value of the ninja at index `0` can be accessed both through the `ninjas` collection and through our getter method:

```
assert(ninjaCollection.firstNinja === "Hachi"
    && ninjaCollection.ninjas[0] ===  "Hachi",
       "Now Hachi is the first ninja");
```

Figure 8.2 shows the output generated by listing 8.2. When we access a property with a getter (for example, through `ninjaCollection.firstNinja`), the getter method is immediately called, and in this case, the message `Getting firstNinja` is logged. Later, we test that the output is `Yoshi` and that the message `Yoshi is the first ninja` is logged. We proceed similarly by assigning a new value to the `firstNinja` property,
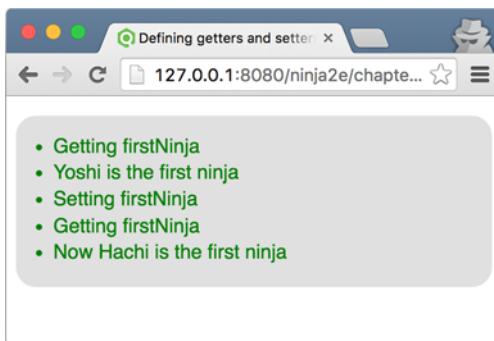
Figure 8.2   The output from listing 8.2: if a property has a getter and a setter method, the getter method is implicitly called whenever we read the property value, and the setter method is called whenever we assign a new value to the property.

and as we can see in the output, this implicitly triggers the execution of the setter method, which outputs the message `Setting firstNinja`.

An important point to take from all this is that native getters and setters allow us to specify properties that are accessed as standard properties, but that are methods whose execution is triggered immediately when the property is accessed. This is further emphasized in figure 8.3.

Accessing a getter property immediately triggers the execution of the matching getter method!

The getter method is called, and a matching execution context is created and pushed on the stack. The process is the same as when a standard function is called.
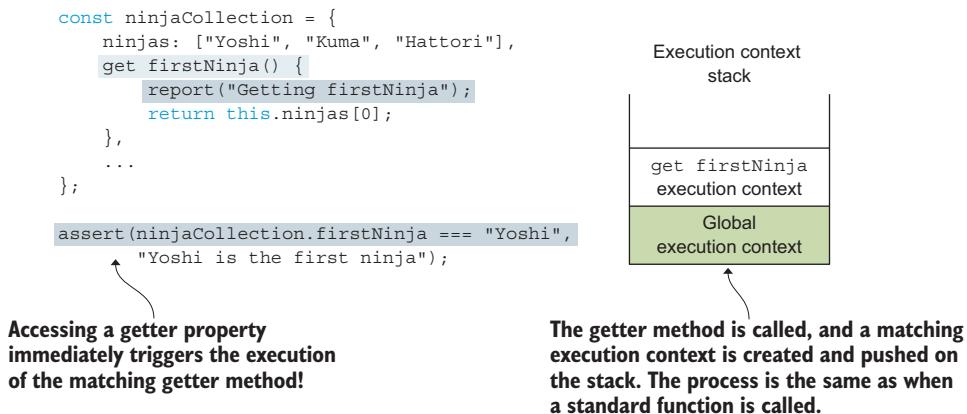
Figure 8.3   Accessing a property with a getter method implicitly calls the matching getter. The process is the same as if this were a standard method call, and the getter method gets executed. A similar thing happens when we assign a value to a property through a setter method.

This syntax for defining a getter and a setter is straightforward, so it's no wonder that we can use the exact same syntax to define getters and setters in other situations. The following example uses ES6 classes.

Listing 8.3   Using getters and setters with ES6 classes

```
class NinjaCollection {
  constructor(){
    this.ninjas = ["Yoshi", "Kuma", "Hattori"];
  }
```

```
  get firstNinja(){
    report("Getting firstNinja");
    return this.ninjas[0];
  }
  set firstNinja(value){
    report("Setting firstNinja");
    this.ninjas[0] = value;
  }
}
const ninjaCollection = new NinjaCollection();

assert(ninjaCollection.firstNinja === "Yoshi",
       "Yoshi is the first ninja");

ninjaCollection.firstNinja = "Hachi";

assert(ninjaCollection.firstNinja === "Hachi"
    && ninjaCollection.ninjas[0] ===  "Hachi",
       "Now Hachi is the first ninja");
```

Defines a getter and a setter within an ES6 class

This modifies the code from listing 8.2 to include ES6 classes. We keep all the tests to verify that the example still works as expected.

> **NOTE** We don't always have to define both a getter and a setter for a given property. For example, often we'll want to provide only a getter. If in that case we still attempt to write a value to that property, the exact behavior depends on whether the code is in strict or nonstrict mode. If the code is in nonstrict mode, assigning a value to a property with only a getter achieves nothing; the JavaScript engine will silently ignore our request. If, on the other hand, the code is in strict mode, the JavaScript engine will throw a type error, indicating that we're trying to assign a value to a property that has a getter but no setter.

Although specifying getters and setters through ES6 classes and object literals is easy, you've probably noticed something missing. Traditionally, getters and setters are used to control access to private object properties, as in listing 8.1. Unfortunately, as we already know from chapter 5, JavaScript doesn't have private object properties. Instead, we can mimic them through closures, by defining variables and specifying object methods that will close over those variables. Because with object literals and classes our getter and setter methods aren't created within the same function scope as variables that we could use for private object properties, we can't do this. Luckily, there's an alternative way, through the `Object.defineProperty` method.

In chapter 7, you saw that the `Object.defineProperty` method can be used to define new properties by passing in a property descriptor object. Among other things, the property descriptor can include a `get` and a `set` property that define the property's getter and setter methods.

We'll use this feature to modify listing 8.1 to implement built-in getters and setters that control access to a "private" object property, as shown in the following listing.

Listing 8.4   Defining getters and setters with `Object.defineProperty`

Defines a constructor function

Defines a "private" variable that will be accessible through function closures

Uses the built-in Object.defineProperty to define a skillLevel property

```
function Ninja() {
  let _skillLevel = 0;

  Object.defineProperty(this, 'skillLevel', {
    get: () => {
      report("The get method is called");
      return _skillLevel;
    },
    set: value => {
      report("The set method is called");
      _skillLevel = value;
    }
  });
}

const ninja = new Ninja();

assert(typeof ninja._skillLevel === "undefined",
       "We cannot access a 'private' property");
assert(ninja.skillLevel === 0, "The getter works fine!");

ninja.skillLevel = 10;
assert(ninja.skillLevel === 10, "The value was updated");
```

A get method that will be called whenever we read the skillLevel property.

A set method that will be called whenever we assign a value to the skillLevel property.

Creates a new Ninja instance

The private variable isn't accessible directly, but through the skillLevel getter.

The set method is implicitly called when assigning to the skillLevel property.

In this example, we first define a `Ninja` constructor function with a `_skillLevel` variable that we'll use as a private variable, just as in listing 8.1.

Next, on the newly created object, referenced by the `this` keyword, we define a `skillLevel` property by using the built-in `Object.defineProperty` method:

```
Object.defineProperty(this, 'skillLevel', {
  get: () => {
    report("The get method is called");
    return _skillLevel;
  },
  set: value => {
    report("The set method is called");
    _skillLevel = value;
  }
});
```

Because we want the `skillLevel` property to control access to a private variable, we specify a `get` and a `set` method that will be called whenever the property is accessed.

Notice that, unlike getters and setters specified on object literals and classes, the `get` and `set` methods defined through `Object.defineProperty` are created in the same scope as the "private" `skillLevel` variable. Both methods create a closure

around the private variable, and we can access that private variable only through these two methods.

The rest of the code works exactly as in the previous examples. We create a new `Ninja` instance and check that we can't access the private variable directly. Instead, all interactions have to go through the getter and setter, which we now use just as if they were standard object properties:

```
ninja.skillLevel === 0      ◁─┐      Activates the getter method
ninja.skillLevel = 10        ◁──────  Activates the setter method
```

As you can see, the approach with `Object.defineProperty` is more verbose and complicated than getters and setters in object literals and classes. But in certain cases, when we need private object properties, it's well worth it.

Regardless of the way we define them, getters and setters allow us to define object properties that are used like standard object properties, but are methods that can execute additional code whenever we read or write to a particular property. This is an incredibly useful feature that enables us to perform logging, validate assignment values, and even notify other parts of the code when certain changes occur. Let's explore some of these applications.

8.1.2 *Using getters and setters to validate property values*

As we've established so far, a setter is a method that's executed whenever we write a value to the matching property. We can take advantage of setters to perform an action whenever code attempts to update the value of a property. For example, one of the things we can do is validate the passed-in value. Take a look at the following code, which ensures that our `skillLevel` property can be assigned only integer values.

Listing 8.5 Validating property value assignments with setters

```
function Ninja() {
  let _skillLevel = 0;

  Object.defineProperty(this, 'skillLevel', {        Checks whether the passed-in
    get: () => _skillLevel,                           value is an integer. If it isn't,
    set: value => {                                   an exception is thrown.
      if(!Number.isInteger(value)){
        throw new TypeError("Skill level should be a number");
      }
      _skillLevel = value;
    }
  });
}

const ninja = new Ninja();

ninja.skillLevel = 10;                                We can assign an
assert(ninja.skillLevel === 10, "The value was updated");   integer value to
                                                      the property.
```

```
try {
  ninja.skillLevel = "Great";
  fail("Should not be here");
} catch(e){
  pass("Setting a non-integer value throws an exception");
}
```

This example is a straightforward extension to listing 8.4. The only major difference is that now, whenever a new value is assigned to the skillLevel property, we check whether the passed-in value is an integer. If it isn't, an exception is thrown, and the private _skillLevel variable won't be modified. If everything went okay and an integer value is received, we end up with a new value of the private _skillLevel variable:

```
set: value => {
  if(!Number.isInteger(value)){
    throw new TypeError("Skill level should be a number");
  }
  _skillLevel = value;
}
```

When testing this code, we first check that all goes okay if we assign an integer:

```
ninja.skillLevel = 10;
assert(ninja.skillLevel === 10, "The value was updated");
```

And then we test the situation in which we mistakenly assign a value of another type, such as a string. In that case, we should end up with an exception.

```
try {
  ninja.skillLevel = "Great";
  fail("Should not be here");
} catch(e){
  pass("Setting a non-integer value throws an exception");
}
```

This is how you avoid all those silly little bugs that happen when a value of the wrong type ends up in a certain property. Sure, it adds overhead, but that's a price that we sometimes have to pay to safely use a highly dynamic language such as JavaScript.

This is just one example of the usefulness of setter methods; there are many more that we won't explicitly explore. For example, you can use the same principle to track value history, perform logging, provide change notification, and more.

8.1.3 Using getters and setters to define computed properties

In addition to being able to control access to certain object properties, getters and setters can be used to define *computed properties*, properties whose value is calculated per request. Computed properties don't store a value; they provide a get and/or a set method to retrieve and set other properties indirectly. In the following example, the object has two properties, name and clan, which we'll use to compute the property fullTitle.

Listing 8.6   Defining computed properties

```
const shogun = {
  name: "Yoshiaki",
  clan: "Ashikaga",
  get fullTitle(){
    return this.name + " " + this.clan;
  },
  set fullTitle(value) {
    const segments = value.split(" ");
    this.name = segments[0];
    this.clan = segments[1];
  }
};
```

Defines a getter method on a fullTitle property of an object literal that calculates the value by concatenating two object properties

Defines a setter method on a fullTitle property of an object literal that splits the passed-in value and updates two standard properties

```
assert(shogun.name === "Yoshiaki", "Our shogun Yoshiaki");
assert(shogun.clan === "Ashikaga", "Of clan Ashikaga");
assert(shogun.fullTitle === "Yoshiaki Ashikaga",
       "The full name is now Yoshiaki Ashikaga");

shogun.fullTitle = "Ieyasu Tokugawa";
assert(shogun.name === "Ieyasu", "Our shogun Ieyasu");
assert(shogun.clan === "Tokugawa", "Of clan Tokugawa");
assert(shogun.fullTitle === "Ieyasu Tokugawa",
       "The full name is now Ieyasu Tokugawa");
```

The name and clan properties are normal properties whose values are directly obtained. Accessing the fullTitle property calls the get method, which computes the value.

Assigning a value to the fullTitle property calls the set method, which computes and assigns new values to the name and clan properties.

Here we define a shogun object, with two standard properties, name and clan. In addition, we specify a getter and a setter method for a computed property, fullTitle:

```
const shogun = {
  name: "Yoshiaki",
  clan: "Ashikaga",
  get fullTitle(){
    return this.name + " " + this.clan;
  },
  set fullTitle(value) {
    const segments = value.split(" ");
    this.name = segments[0];
    this.clan = segments[1];
  }
};
```

The get method computes the value of the fullTitle property, on request, by concatenating the name and clan properties. The set method, on the other hand, uses the built-in split method, available to all strings, to split the assigned string into segments by the space character. The first segment represents the name and is assigned to the name property, whereas the second segment represents the clan and is assigned to the clan property.

This takes care of both routes: Reading the `fullTitle` property computes its value, and writing to the `fullTitle` property modifies the properties that constitute the property value.

To be honest, we don't have to use computed properties. A method called `get-FullTitle` could be equally useful, but computed properties can improve the conceptual clarity of our code. If a certain value (in this case, the `fullTitle` value) depends *only* on the internal state of the object (in this case, on the `name` and `clan` properties), it makes perfect sense to represent it as a data field, a property, instead of a function.

This concludes our exploration of getters and setters. You've seen that they're a useful addition to the language that can help us deal with logging, data validation, and detecting changes in property values. Unfortunately, sometimes this isn't enough. In certain cases, we need to control all types of interactions with our objects, and for this, we can use a completely new type of object: a *proxy*.

8.2   *Using proxies to control access*

A *proxy* is a surrogate through which we control access to another object. It enables us to define custom actions that will be executed when an object is being interacted with—for example, when a property value is read or set, or when a method is called. You can think of proxies as almost a generalization of getters and setters; but with each getter and setter, you control access to only a single object property, whereas proxies enable you to generically handle all interactions with an object, including even method calls.

We can use proxies when we'd traditionally use getters and setters, such as for logging, data validation, and computed properties. But proxies are even more powerful. They allow us to easily add profiling and performance measurements to our code, autopopulate object properties in order to avoid pesky null exceptions, and to wrap host objects such as the DOM in order to reduce cross-browser incompatibilities.

> **NOTE**   Proxies are introduced by ES6. For current browser support, see http://mng.bz/9uEM.

In JavaScript, we can create proxies by using the built-in `Proxy` constructor. Let's start simple, with a proxy that intercepts all attempts to read and write to properties of an object.

Listing 8.7   Creating proxies with the `Proxy` constructor

The emperor is our target object.

Creates a proxy with the Proxy constructor that takes in the object the proxy wraps...

```
const emperor = { name: "Komei" };          ◁
const representative = new Proxy(emperor, {   ◁
```

Accesses the name property both through the emperor object and through the proxy object

...and an object with traps that will be called when reading (get) and writing (set) to properties.

Accessing a non-existing property directly on the object returns undefined.

Accessing a property through a proxy detects that the object doesn't exist in our target object, so a warning message is returned.

```
    get: (target, key) => {
      report("Reading " + key + " through a proxy");
      return key in target ? target[key]
                            : "Don't bother the emperor!"
    },
    set: (target, key, value) => {
      report("Writing " + key + " through a proxy");
      target[key] = value;
    }
});

assert(emperor.name === "Komei", "The emperor's name is Komei");
assert(representative.name === "Komei",
      "We can get the name property through a proxy");

assert(emperor.nickname === undefined,
      "The emperor doesn't have a nickname ");
assert(representative.nickname === "Don't bother the emperor!",
      "The proxy jumps in when we make inproper requests");

representative.nickname = "Tenno";
assert(emperor.nickname === "Tenno",
      "The emperor now has a nickname");
assert(representative.nickname === "Tenno",
      "The nickname is also accessible through the proxy");
```

Adds a property through the proxy. The property is accessible both through the target object and through the proxy.

We first create our base emperor object that has only a name property. Next, by using the built-in Proxy constructor, we wrap our emperor object (or *target* object, as it's commonly called) into a proxy object named representative. During proxy construction, as a second argument, we also send an object that specifies *traps*, functions that will be called when certain actions are performed on an object:

```
const representative = new Proxy(emperor, {
  get: (target, key) => {
    report("Reading " + key + " through a proxy");
    return key in target ? target[key]
                          : "Don't bother the emperor!"
  },
  set: (target, key, value) => {
    report("Writing " + key + " through a proxy");
    target[key] = value;
  }
});
```

In this case, we've specified two traps: a get trap that will be called whenever we try to read a value of a property through the proxy, and a set trap that will be called whenever we set a property value through the proxy. The *get* trap performs the following functionality: If the target object has a property, that property is returned; and if the

object doesn't have a property, we return a message warning our user not to bother the emperor with frivolous details.

```
get: (target, key) => {
  report("Reading " + key + " through a proxy");
  return key in target ? target[key]
                       : "Don't bother the emperor!"
}
```

Next, we test that we can access the name property both directly through the target emperor object as well as through our proxy object:

```
assert(emperor.name === "Komei", "The emperor's name is Komei");
assert(representative.name === "Komei",
       "We can get the name property through a proxy");
```

If we access the name property directly through the emperor object, the value Komei is returned. But if we access the name property through the proxy object, the get trap is implicitly called. Because the name property is found in the target emperor object, the value Komei is also returned. See figure 8.4.

> **NOTE** It's important to emphasize that proxy traps are activated in the same way as getters and setters. As soon as we perform an action (for example, accessing a property value on a proxy), the matching trap gets implicitly called, and the JavaScript engine goes through a similar process as if we've explicitly invoked a function.

On the other hand, if we access a nonexisting nickname property directly on the target emperor object, we'll get, unsurprisingly, an undefined value. But if we try to access it through our proxy object, the get handler will be activated. Because the target
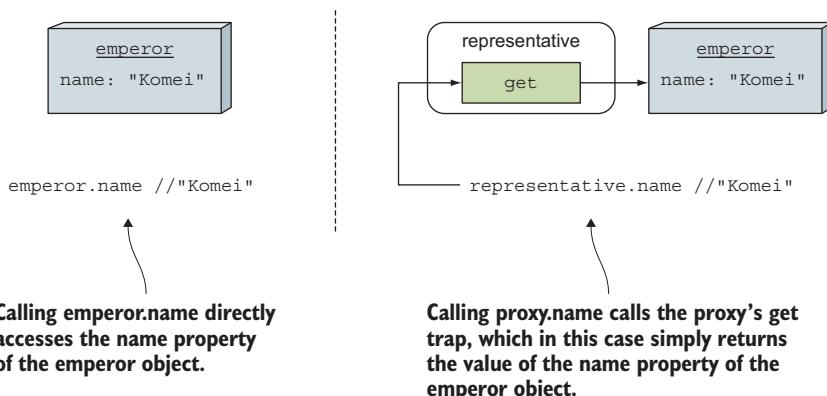
Figure 8.4 Accessing the `name` property directly (on the left) and indirectly, through a proxy (on the right)

emperor object doesn't have a `nickname` property, the proxy's `get` trap will return the
`Don't bother the emperor!` message.

We'll continue the example by assigning a new property through our proxy object:
`representative.nickname = "Tenno"`. Because the assignment is done through a
proxy, and not directly, the `set` trap, which logs a message and assigns a property to
our target `emperor` object, is activated:

```
set: (target, key, value) => {
    report("Writing " + key + " through a proxy");
    target[key] = value;
}
```

Naturally, the newly created property can be accessed both through the proxy object
and the target object:

```
assert(emperor.nickname === "Tenno",
      "The emperor now has a nickname");
assert(representative.nickname === "Tenno",
      "The nickname is also accessible through the proxy");
```

This is the gist of how to use proxies: Through the `Proxy` constructor, we create a
proxy object that controls access to the target object by activating certain traps, when-
ever an operation is performed directly on a proxy.

In this example, we've used the `get` and `set` traps, but many other built-in traps allow
us to define handlers for various object actions (see http://mng.bz/ba55). For
example:

- The *apply* trap will be activated when calling a function, and the *construct* trap
  when using the new operator.
- The *get* and *set* traps will be activated when reading/writing to a property.
- The *enumerate* trap will be activated for `for-in` statements.
- `getPrototypeOf` and `setPrototypeOf` will be activated for getting and setting
  the prototype value.

We can intercept many operations, but going through all of them is outside the scope
of this book. For now, we turn our attention to a few operations that we can't override:
equality (== or ===), `instanceof`, and the `typeof` operator.

For example, the expression x == y (or a stricter x === y) is used to check whether
x and y refer to identical objects (or are of the same value). This equality operator has
some assumptions. For example, comparing two objects should always return the
same value for the same two objects, which isn't something that we can guarantee if
that value is determined by a user-specified function. In addition, the act of compar-
ing two objects shouldn't give access to one of those objects, which would be the case
if equality could be trapped. For similar reasons, the `instanceof` and the `typeof` oper-
ators can't be trapped.

Now that we know how proxies work and how to create them, let's explore some of their practical aspects, such as how to use proxies for logging, performance measurement, autopopulating properties, and implementing arrays that can be accessed with negative indexes. We'll start with logging.

8.2.1 Using proxies for logging

One of the most powerful tools when trying to figure out how code works or when trying to get to the root of a nasty bug is *logging*, the act of outputting information that we find useful at a particular moment. We might, for example, want to know which functions are called, how long they've been executing, what properties are read or written to, and so on.

Unfortunately, when implementing logging, we usually scatter logging statements throughout the code. Take a look at the `Ninja` example used earlier in the chapter.

Listing 8.8   Logging without proxies

```
function Ninja() {
    let _skillLevel = 0;

    Object.defineProperty(this, 'skillLevel', {
      get: () => {
        report("skillLevel get method is called");      ◁   We log whenever the
        return _skillLevel;                                  skillLevel property is read...
      },
      set: value => {
        report("skillLevel set method is called");      ◁   ...and whenever the skillLevel
        _skillLevel = value;                                 property is written to.
      }
    });
  }

  const ninja = new Ninja();
  ninja.skillLevel;                    ◁   Reads the skillLevel
  ninja.skillLevel = 4;        ◁           property and triggers
                                           the get method
                             Writes to the skillLevel property
                             and triggers the set method
```

We define a `Ninja` constructor function that adds a getter and a setter to the `skill-Level` property, which log all attempts of reading and writing to that property.

Notice that this isn't an ideal solution. We've cluttered our domain code that deals with reading and writing to an object property with logging code. In addition, if in the future we need more properties on the `ninja` object, we have to be careful not to forget to add additional logging statements to each new property.

Luckily, one of the straightforward uses of proxies is to enable logging whenever we read or write to a property, but in a much nicer and cleaner way. Consider the following example.

Listing 8.9   Using proxies makes it easier to add logging to objects

Defines a function that takes a target object and makes it loggable

Creates a new proxy with that target object

```
function makeLoggable(target){
  return new Proxy(target, {
    get: (target, property) => {
      report("Reading " + property);
      return target[property];
    },

    set: (target, property, value) => {
      report("Writing value " + value + " to " + property);
      target[property] = value;
    }
  });
}

let ninja = { name: "Yoshi"};
ninja = makeLoggable(ninja);

assert(ninja.name === "Yoshi", "Our ninja Yoshi");
ninja.weapon = "sword";
```

A get trap that logs property reads

Creates a new ninja object that will serve as our target object and make it loggable

A set trap that logs property writes

Reads and writes to our proxy object. These actions are logged by the proxy traps.

Here we define a `makeLoggable` function that takes a `target` object and returns a new `Proxy` that has a handler with a `get` and a `set` trap. These traps, besides reading and writing to the property, log the information about which property is read or written to.

Next, we create a `ninja` object with a `name` property, and we pass it to the `make-Loggable` function, in which it will be used as a target for a newly created proxy. We then assign the proxy back to the `ninja` identifier, overriding it. (Don't worry, our original `ninja` object is kept alive as the target object of our proxy.)

Whenever we try to read a property (for example, with `ninja.name`), the `get` trap will be called, and the information about which property has been read will be logged. A similar thing will happen when writing to a property: `ninja.weapon = "sword"`.

Notice how much easier and more transparent this is when compared to the standard way of using getters and setters. We don't have to mix our domain code with our logging code, and there's no need to add separate logging for each object property. Instead, all property reads and writes go through our proxy object trap methods. Logging has been specified in only one place and is reused as many times as necessary, on as many objects as necessary.

8.2.2   *Using proxies for measuring performance*

Besides being used for logging property accesses, proxies can be used for measuring the performance of function invocations, without even modifying the source code of a

function. Say we want to measure the performance of a function that calculates whether a number is a prime, as shown in the following listing.

Listing 8.10 Measuring performance with proxies

```
                function isPrime(number){
                  if(number < 2) { return false; }

                  for(let i = 2; i < number; i++) {              Defines a primitive
                    if(number % i === 0) { return false; }       implementation of the
                  }                                              isPrime function

                  return true;
                }

  Wraps the
    isPrime     isPrime = new Proxy(isPrime, {            Provides an apply trap that will
   function       apply: (target, thisArg, args) => {     be called whenever a proxy is
   within a          console.time("isPrime");             called as a function
     proxy
  Starts a timer
  called isPrime
                      const result = target.apply(thisArg, args);    Invokes
                                                                     the target
  Stops the timer     console.timeEnd("isPrime");                    function
  and outputs the
  result              return result;
                    }                          Calls the function. The call
                });                            works the same as if we'd
                                               called the original function.
                isPrime(1299827);
```

In this example, we have a simple `isPrime` function. (The exact function doesn't matter; we're using it as an example of a function whose execution can last a nontrivial amount of time.)

Now imagine that we need to measure the performance of the `isPrime` function, but without modifying its code. We could wrap the function into a proxy that has a trap that will be called whenever the function is called:

```
isPrime = new Proxy(isPrime, {
  apply: (target, thisArg, args) => {
...
  }
});
```

We use the `isPrime` function as the target object of a newly constructed proxy. In addition, we supply a handler with an `apply` trap that will be executed on function invocation.

Similarly, as in the previous example, we've assigned the newly created proxy to the `isPrime` identifier. In that way, we don't have to change any of the code that calls the function whose execution time we want to measure; the rest of the program code is completely oblivious to our changes. (How's that for some ninja stealth action?)

Whenever the `isPrime` function is called, that call is rerouted to our proxy's `apply` trap, which will start a stopwatch with the built-in `console.time` method (remember chapter 1), call the original `isPrime` function, log the elapsed time, and finally return the result of the `isPrime` invocation.

8.2.3 *Using proxies to autopopulate properties*

In addition to simplifying logging, proxies can be used for autopopulating properties. For example, imagine that you have to model your computer's folder structure, in which a folder object can have properties that can also be folders. Now imagine that you have to model a file at the end of a long path, such as this:

```
rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";
```

To create this, you might write something along the following lines:

```
const rootFolder = new Folder();
rootFolder.ninjasDir = new Folder();
rootFolder.ninjasDir.firstNinjaDir = new Folder();
rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";
```

Seems a tad more tedious than necessary, doesn't it? This is where autopopulating properties comes into play; just take a look at the following example.

Listing 8.11 Autopopulating properties with proxies

```
function Folder() {
  return new Proxy({}, {
    get: (target, property) => {               Logs all readings
      report("Reading " + property);           to our object

      if(!(property in target)) {              If the accessed
        target[property] = new Folder();        property doesn't
      }                                         exist, we create it.

      return target[property];
    }
  });
}
                                        Whenever a property is accessed, the
const rootFolder = new Folder();         get trap, which creates a property if
                                             it doesn't exist, is activated.
try {
  rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";
  pass("An exception wasn't raised");
}                                        No exception will be raised.
catch(e){
  fail("An exception has occurred");
}
```

Normally, if we consider only the following code, we'd expect an exception to be raised:

```
const rootFolder = new Folder();
rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";
```

We're accessing a property, `firstNinja-Dir`, of an undefined property, `ninjas-Dir`, of the `rootFolder` object. But if we run the code, you see that all is well, as shown in figure 8.5.

This happens because we're using proxies. Every time we access a property, the proxy `get` trap is activated. If our folder object already contains the requested property, its value is returned, and if it doesn't, a new folder is created and assigned to the property. This is how two of our properties, `ninjasDir` and `firstNinjaDir`, are created. Requesting a value of an uninitialized property triggers its creation.

Figure 8.5   **The output of running the code from listing 8.11**

Finally, we have a tool for ridding ourselves of some cases of the pesky null exception!

8.2.4   *Using proxies to implement negative array indexes*

In our day-to-day programming, we'll usually work with *a lot* of arrays. Let's explore how to take advantage of proxies to make our dealings with arrays a little more pleasant.

If your programming background is from languages such as Python, Ruby, or Perl, you might be used to negative array indexes, which enable you to use negative indexes to access array items from the back, as shown in the following snippet:

```
const ninjas = ["Yoshi", "Kuma", "Hattori"];

ninjas[0]; //"Yoshi"          Standard access to array
ninjas[1]; //"Kuma"           items, with positive
ninjas[2]; //"Hattori"        array indexes

ninjas[-1]; //"Hattori"       Negative array indexes enable us to access
ninjas[-2]; //"Kuma"          array items from the back, starting with –l,
ninjas[-3]; //"Yoshi"         which accesses the last array item.
```

Now compare the code that we normally use to access the last item in the array, `ninjas[ninjas.length-1]`, with the code that we can use if our language of choice supports negative array indexes, `ninjas[-1]`. See how much more elegant this is?

Unfortunately, JavaScript doesn't offer built-in support for negative array indexes, but we can mimic this ability through proxies. To explore this concept, we'll look at a
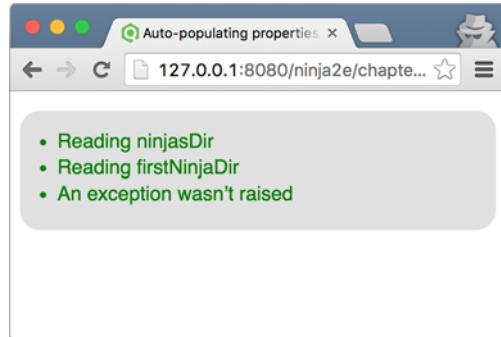
slightly simplified version of code written by Sindre Sorhus (https://github.com/sindresorhus/negative-array), as shown in the following listing.

Listing 8.12 Negative array indexes with proxies

If our target object isn't an array, throw an exception.

The get trap is activated whenever an array index is read.

Returns a new proxy that takes in the array and uses it as a proxy target

```
function createNegativeArrayProxy(array){
  if (!Array.isArray(array)) {
    throw new TypeError('Expected an array');
  }

  return new Proxy(array, {
    get: (target, index) => {
      index = +index;
      return target[index < 0 ? target.length + index : index];
    },
    set: (target, index, val) => {
      index = +index;
      return target[index < 0 ? target.length + index : index] = val;
    }
  });
}

const ninjas = ["Yoshi", "Kuma", "Hattori"];
const proxiedNinjas = createNegativeArrayProxy(ninjas);

assert(ninjas[0] === "Yoshi" && ninjas[1] === "Kuma"
    && ninjas[2] === "Hattori",
        "Array items accessed through positive indexes");

assert(proxiedNinjas[0] === "Yoshi" && proxiedNinjas[1] === "Kuma"
    && proxiedNinjas [2] === "Hattori",
        "Array items accessed through positive indexes on a proxy");

assert(typeof ninjas[-1] === "undefined"
    && typeof ninjas[-2] === "undefined"
    && typeof ninjas[-3] === "undefined",
        "Items cannot be accessed through negative indexes on an array");

assert(proxiedNinjas[-1] === "Hattori"
    && proxiedNinjas[-2] === "Kuma"
    && proxiedNinjas[-3] === "Yoshi",
        "But they can be accessed through negative indexes");

proxiedNinjas[-1] = "Hachi";
assert(proxiedNinjas[-1] === "Hachi" && ninjas[2] === "Hachi",
        "Items can be changed through negative indexes");
```

Turns the property name into a number with the unary plus operator

If the read index is a negative number, read from the back of the array, and if it's a positive number, access it normally.

The set trap is activated whenever an array index is written to.

Creates a standard array

Checks that we can access array items through the original array as well as through the proxy

Passes it into our function that will create a proxy to that array

Checks that we can't access array items through negative indexes in a standard array...

...but that we can do it through our proxy, because we've supplied a get trap that handles the case.

We can also modify array items from the back, but only if we go through the proxy.

In this example, we define a function that will create a proxy for a passed-in array. Because we don't want our proxy to work with other types of objects, we throw an exception in case the argument isn't an array:

```
if (!Array.isArray(array)) {
    throw new TypeError('Expected an array');
}
```

We continue by creating and returning a new proxy with two traps, a `get` trap that will activate whenever we try to read an array item, and a `set` trap that will activate whenever we write to an array item:

```
return new Proxy(array, {
  get: (target, index) => {
    index = +index;
    return target[index < 0 ? target.length + index : index];
  },
  set: (target, index, val) => {
    index = +index;
    return target[index < 0 ? target.length + index : index] = val;
  }
});
```

The trap bodies are similar. First, we turn the property into a number by using the unary plus operator (`index = +index`). Then, if the requested index is less than `0`, we access array items from the back by anchoring to the length of the array, and if it's greater than or equal to `0`, we access the array item in a standard fashion.

Finally, we perform various tests to check that on normal arrays we can only access array items through positive array indexes, and that, if we use a proxy, we can both access and modify array items through negative indexes.

Now that you've seen how to use proxies to achieve some interesting features such as autopopulating object properties and accessing negative array indexes, which are outright impossible without proxies, let's explore the most significant downside to proxies: performance issues.

8.2.5 *Performance costs of proxies*

As we already know, a proxy is a surrogate object through which we control access to another object. A proxy can define traps, functions that will be executed whenever a certain operation is performed on a proxy. And, as you've also seen, we can use these traps to implement useful functionalities such as logging, performance measurements, autopopulating properties, negative array indexes, and so on. Unfortunately, there's also a downside. The fact that all our operations have to pass in through the proxy adds a layer of indirection that enables us to implement all these cool features, but at the same time it introduces a significant amount of additional processing that impacts performance.

To test these performance issues, we can build on the negative array indexes example from listing 8.12 and compare the execution time when accessing items in a normal array versus accessing items through a proxy, as shown in the following listing.

Listing 8.13   Checking the performance limitations of proxies

```
function measure(items){
  const startTime = new Date().getTime();
  for(let i = 0; i < 500000; i++){
    items[0] === "Yoshi";
    items[1] === "Kuma";
    items[2] === "Hattori";
  }
  return new Date().getTime() - startTime;
}

const ninjas = ["Yoshi", "Kuma", "Hattori"];
const proxiedNinjas = createNegativeArrayProxy(ninjas);

console.log("Proxies are around",
            Math.round(measure(proxiedNinjas)/ measure(ninjas)),
            "times slower");
```

Accesses the items in our collection in a long-running loop

Gets the current time before running a long-running operation

Measures the time it took for the long-running code to execute

Compares the running time when accessing the standard array versus when accessing through a proxy

Creates a standard array and a proxy for that array

Because a single operation of the code happens much too quickly to measure reliably, the code has to be executed many times to get a measurable value. Frequently, this count can be in the tens of thousands, or even millions, depending on the nature of the code being measured. A little trial and error lets us choose a reasonable value: in this case 500,000.

We also need to bracket the execution of the code with two `new Date().getTime()` timestamps: one before we start executing the target code, and one after. Their difference tells us how long the code took to perform. Finally, we can compare the results by calling the `measure` function on both the proxied array and the standard array.

On our machine, the results don't fare well for proxies. It turns out that in Chrome, proxies are around 50 times slower; in Firefox, they're about 20 times slower.

For now, we recommend that you be careful when using proxies. Although they allow you to be creative with controlling access to objects, that amount of control comes with performance issues. You can use proxies with code that's not performance sensitive, but be careful when using them in code that's executed a lot. As always, we recommend that you thoroughly test the performance of your code.

8.3   Summary

- We can monitor our objects with getters, setters, and proxies.
- By using accessor methods (getters and setters), we can control access to object properties.

- Accessor properties can be defined by using the built-in `Object.define-Property` method or with a special `get` and `set` syntax as parts of object literals or ES6 classes.
  - A `get` method is implicitly called whenever we try to read, whereas a `set` method is called whenever we assign a value to the matching object's property.
  - Getter methods can be used to define computed properties, properties whose value is calculated on a per request basis, whereas setter methods can be used to achieve data validation and logging.
- Proxies are an ES6 addition to JavaScript and are used to control other objects.
  - Proxies enable us to define custom actions that will be executed when an object is interacted with (for example, when a property is read or a function is called).
  - All interactions have to go through the proxy, which has traps that are triggered when a specific action occurs.
- Use proxies to achieve elegant
  - Logging
  - Performance measurements
  - Data validation
  - Autopopulating object properties (thereby avoiding pesky null exceptions)
  - Negative array indexes
- Proxies aren't fast, so be careful when using them in code that's executed a lot. We recommend that you do performance testing.

8.4   Exercises

1 After running the following code, which of the following expressions will throw an exception and why?

```
const ninja = {
   get name() {
      return "Akiyama";
   }
}

aninja.name();
bconst name = ninja.name;
```

2 In the following code, which mechanism allows getters to access a private object variable?

```
function Samurai() {
  const _weapon = "katana";
  Object.defineProperty(this, "weapon", {
    get: () => _weapon
  });
}
const samurai = new Samurai();
assert(samurai.weapon === "katana", "A katana wielding samurai");
```

3 Which of the following assertions will pass?

```
const daimyo = { name: "Matsu", clan: "Takasu"};
const proxy = new Proxy(daimyo, {
  get: (target, key) => {
    if(key === "clan"){
      return "Tokugawa";
    }
  }
});

assert(daimyo.clan === "Takasu", "Matsu of clan Takasu");
assert(proxy.clan === "Tokugawa", "Matsu of clan Tokugawa?");

proxy.clan = "Tokugawa";

assert(daimyo.clan === "Takasu", "Matsu of clan Takasu");
assert(proxy.clan === "Tokugawa", "Matsu of clan Tokugawa?");
```

4 Which of the following assertions will pass?

```
const daimyo = { name: "Matsu", clan: "Takasu", armySize: 10000};
const proxy = new Proxy(daimyo, {
  set: (target, key, value) => {
    if(key === "armySize") {
      const number = Number.parseInt(value);
      if(!Number.isNaN(number)){
        target[key] = number;
      }
    } else {
      target[key] = value;
    }
  },
});

assert(daimyo.armySize === 10000, "Matsu has 10 000 men at arms");
assert(proxy.armySize === 10000, "Matsu has 10 000 men at arms");

proxy.armySize = "large";
assert(daimyo.armySize === "large", "Matsu has a large army");

daimyo.armySize = "large";
assert(daimyo.armySize === "large", "Matsu has a large army");
```

Dealing with collections

9

> ### *This chapter covers*
> - Creating and modifying arrays
> - Using and reusing array functions
> - Creating dictionaries with maps
> - Creating collections of unique objects with sets

Now that we've spent some time wrangling the particularities of object-orientation in JavaScript, we'll move on to a closely related topic: collections of items. We'll start with arrays, the most basic type of collection in JavaScript, and look at some array peculiarities you may not expect if your programming background is in another programming language. We'll continue by exploring some of the built-in array methods that will help you write more elegant array-handling code.

Next, we'll discuss two new ES6 collections: maps and sets. Using maps, you can create dictionaries of a sort that carry mappings between keys and values—a collection that's extremely useful in certain programming tasks. Sets, on the other hand, are collections of unique items in which each item can't occur more than once. Let's begin our exploration with the simplest and most common of all collections: arrays.

....

....

9.1 Arrays

Arrays are one of the most common data types. Using them, you can handle collections of items. If your programming background is in a strongly typed language such as C, you probably think of arrays as sequential chunks of memory that house items of the same type, where each chunk of memory is of fixed size and has an associated index through which you can easily access it.

But as with many things in JavaScript, arrays come with a twist: They're just objects. Although this leads to some unfortunate side effects, primarily in terms of performance, it also has some benefits. For example, arrays can access methods, like other objects—methods that will make our lives a lot easier.

In this section, we'll first look at ways to create arrays. Then we'll explore how to add items to and remove items from different positions in an array. Finally, we'll examine the built-in array methods that will make our array-handling code much more elegant.

9.1.1 Creating arrays

There are two fundamental ways to create new arrays:

- Using the built-in `Array` constructor
- Using array literals `[]`

Let's start with a simple example in which we create an array of ninjas and an array of samurai.

Listing 9.1 Creating arrays

To create an array, we can use an array literal [] …

… or the built-in Array constructor.

We access array items with index notation: The first item is indexed with 0, and the last with array.length – l.

The length property tells us the size of the array.

Reading items outside the array bounds results in undefined.

```
const ninjas = ["Kuma", "Hattori", "Yagyu"];
const samurai = new Array("Oda", "Tomoe");

assert(ninjas.length === 3, "There are three ninjas");
assert(samurai.length === 2, "And only two samurai");

assert(ninjas[0] === "Kuma", "Kuma is the first ninja");
assert(samurai[samurai.length-1] === "Tomoe",
       "Tomoe is the last samurai");

assert(ninjas[4] === undefined,
       "We get undefined if we try to access an out of bounds index");
```

```
ninjas[4] = "Ishi";
assert(ninjas.length === 5,
        "Arrays are automatically expanded");

ninjas.length = 2;
assert(ninjas.length === 2, "There are only two ninjas now");
assert(ninjas[0] === "Kuma" && ninjas[1] === "Hattori",
        "Kuma and Hattori");
assert(ninjas[2] === undefined, "But we've lost Yagyu");
```

Writing to indexes outside the array bounds extends the array.

Manually overriding the length property with a lower value deletes the excess items.

In listing 9.1, we start by creating two arrays. The ninjas array is created with a simple array literal:

```
const ninjas = ["Kuma", "Hattori", "Yagyu"];
```

It's immediately prefilled with three ninjas: Kuma, Hattori, and Yagyu. The samurai array is created using the built-in Array constructor:

```
const samurai = new Array("Oda", "Tomoe");
```

> **Array literals versus the Array constructor**
>
> Using array literals to create arrays is preferred over creating arrays with the Array constructor. The primary reason is simplicity: [] versus new Array() (2 characters versus 11 characters—hardly a fair contest). In addition, because JavaScript is highly dynamic, nothing stops someone from overriding the built-in Array constructor, which means calling new Array() doesn't necessarily have to create an array. Thus we recommend that you generally stick to array literals.

Regardless of how we create it, each array has a length property that specifies the size of the array. For example, the length of the ninjas array is 3, and it contains 3 ninjas. We can test this with the following assertions:

```
assert(ninjas.length === 3, "There are three ninjas");
assert(samurai.length === 2, "And only two samurai");
```

As you probably know, you access array items by using index notation, where the first item is positioned at index 0 and the last item at index array.length – 1. But if we try to access an index outside those bounds—for example, with ninjas[4] (remember, we have only three ninjas!), we won't get the scary "Array index out of bounds" exception that we receive in most other programming languages. Instead, undefined is returned, signaling that there's nothing there:

```
assert(ninjas[4] === undefined,
      "We get undefined if we try to access an out of bounds index");
```

This behavior is a consequence of the fact that JavaScript arrays are objects. Just as we'd get `undefined` if we tried to access a nonexistent object property, we get `unde-fined` when accessing a nonexistent array index.

On the other hand, if we try to write to a position outside of array bounds, as in

```
ninjas[4] = "Ishi";
```

the array will expand to accommodate the new situation. For example, see figure 9.1: We've essentially created a hole in the array, and the item at index 3 is `undefined`. This also changes the value of the `length` property, which now reports a value of 5, even though one array item is `undefined`.

```
var ninjas = ["Kuma", "Hattori", "Yagyu"]
```

"Kuma"	"Hattori"	"Yagyu"
0	1	2	length: 3

```
ninjas[4] = "Ishi";
```

"Kuma"	"Hattori"	"Yagyu"	undefined	"Ishi"
0	1	2	3	4	length: 5

Figure 9.1 Writing to an array index outside of array bounds expands the array.

Unlike in most other languages, in JavaScript, arrays also exhibit a peculiar feature related to the `length` property: Nothing stops us from manually changing its value. Setting a value higher than the current length will expand the array with `undefined` items, whereas setting the value to a lower value will trim the array, as in `ninjas.length = 2;`.

Now that we've gone through the basics of array creation, let's go through some of the most common array methods.

9.1.2 *Adding and removing items at either end of an array*

Let's start with the following simple methods we can use to add items to and remove items from an array:

- `push` adds an item to the end of the array.
- `unshift` adds an item to the beginning of the array.
- `pop` removes an item from the end of the array.
- `shift` removes an item from the beginning of the array.

You've probably already used these methods, but just in case, let's make sure we're on the same page by exploring the following listing.

Listing 9.2   Adding and removing array items

```
const ninjas = [];
assert(ninjas.length === 0, "An array starts empty");
```
Creates a new, empty array

```
ninjas.push("Kuma");
assert(ninjas[0] === "Kuma",
       "Kuma is the first item in the array");
assert(ninjas.length === 1, "We have one item in the array");
```
Pushes a new item to the end of the array

```
ninjas.push("Hattori");
assert(ninjas[0] === "Kuma",
       "Kuma is still first");
assert(ninjas[1] === "Hattori",
       "Hattori is added to the end of the array");
assert(ninjas.length === 2,
       "We have two items in the array!");
```
Pushes another item to the end of the array

```
ninjas.unshift("Yagyu");
assert(ninjas[0] === "Yagyu",
       "Now Yagyu is the first item");
assert(ninjas[1] === "Kuma",
       "Kuma moved to the second place");
assert(ninjas[2] === "Hattori",
       "And Hattori to the third place");
assert(ninjas.length === 3,
       "We have three items in the array!");
```
Uses the built-in unshift method to insert the item at the beginning of the array. Other items are adjusted accordingly.

```
const lastNinja = ninjas.pop();
assert(lastNinja === "Hattori",
        "We've removed Hattori from the end of the array");
assert(ninjas[0] === "Yagyu",
       "Now Yagyu is still the first item");
assert(ninjas[1] === "Kuma,
       "Kuma is still in second place");
assert(ninjas.length === 2,
       "Now there are two items in the array");
```
Pops the last item from the array

```
const firstNinja = ninjas.shift();
assert(firstNinja === "Yagyu",
       "We've removed Yagyu from the beginning of the array");
assert(ninjas[0] === "Kuma",
       "Kuma has shifted to the first place");
assert(ninjas.length === 1,
       "There's only one ninja in the array");
```
Removes the first item from the array. Other items are moved to the left accordingly.

In this example, we first create a new, empty `ninjas` array:

```
ninjas = [] // ninjas: []
```

In each array, we can use the built-in `push` method to append an item to the end of the array, changing its length in the process:

```
ninjas.push("Kuma"); // ninjas: ["Kuma"];
ninjas.push("Hattori"); // ninjas: ["Kuma", "Hattori"];
```

We can also add new items to the beginning of the array by using the built in `unshift` method:

```
ninjas.unshift("Yagyu");// ninjas: ["Yagyu", "Kuma", "Hattori"];
```

Notice how existing array items are adjusted. For example, before calling the `unshift` method, `"Kuma"` was at index `0`, and afterward it's at index `1`.

   We can also remove elements from either the end or the beginning of the array. Calling the built-in `pop` method removes an element from the end of the array, reducing the array's length in the process:

```
var lastNinja = ninjas.pop(); // ninjas:["Yagyu", "Kuma"]
                              // lastNinja: "Hattori"
```

We can also remove an item from the beginning of the array by using the built-in `shift` method:

```
var firstNinja = ninjas.shift(); //ninjas: ["Kuma"]
                                 //firstNinja: "Yagyu"
```

Figure 9.2 shows how `push`, `pop`, `shift`, and `unshift` modify arrays.
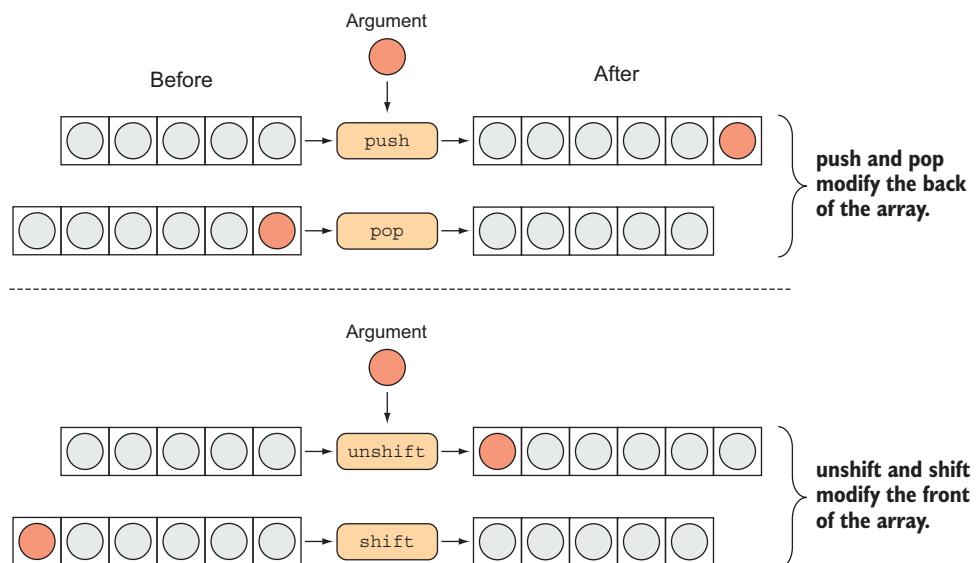
Figure 9.2   The `push` and `pop` methods modify the end of an array, whereas `shift` and `unshift` modify the array's beginning.

Performance considerations: pop and push versus shift and unshift

The `pop` and `push` methods only affect the last item in an array: `pop` by removing the last item, and `push` by inserting an item at the end of the array. On the other hand, the `shift` and `unshift` methods change the first item in the array. This means the indexes of any subsequent array items have to be adjusted. For this reason, `push` and `pop` are significantly faster operations than `shift` and `unshift`, and we recommend using them unless you have a good reason to do otherwise.

9.1.3   *Adding and removing items at any array location*

The previous example removed items from the beginning and end of the array. But this is too constraining—in general, we should be able to remove items from any array location. One straightforward approach for doing this is shown in the following listing.

Listing 9.3   Naïve way to remove an array item

```
const ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];

delete ninjas[1];                    ⟵────── Uses the delete command to delete an item

assert(ninjas.length === 4,
       "Length still reports that there are 4 items");

assert(ninjas[0] === "Yagyu", "First item is Yagyu");
assert(ninjas[1] === undefined, "We've simply created a hole");
assert(ninjas[2] === "Hattori", "Hattori is still the third item");
assert(ninjas[3] === "Fuma", "And Fuma is the last item");
```

We deleted an item, but the array still reports that it has 4 items. We've only created a hole in the array.

This approach to deleting an item from an array doesn't work. We effectively only create a hole in the array. The array still reports that it has four items, but one of them—the one we wanted to delete—is `undefined` (see figure 9.3).

```
var ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"]
```

| "Yagyu" | "Kuma" | "Hattori" | "Fuma" |

```
delete ninjas[1]
```

| "Yagyu" | undefined | "Hattori" | "Fuma" |

Figure 9.3   Deleting an item from an array creates a hole in the array.

Similarly, if we wanted to insert an item at an arbitrary position, where would we even start? As an answer to these problems, all JavaScript arrays have access to the splice method: Starting from a given index, this method removes and inserts items. Check out the following example.

Listing 9.4   Removing and adding items at arbitrary positions

Creates a new array with four items

Uses the built-in splice method to remove one element, starting at index I

```
const ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];

var removedItems = ninjas.splice(1, 1);

assert(removedItems.length === 1, "One item was removed");
assert(removedItems[0] === "Kuma");

assert(ninjas.length === 3,
       "There are now three items in the array");
assert(ninjas[0] === "Yagyu",
       "The first item is still Yagyu");
assert(ninjas[1] === "Hattori",
       "Hattori is now in the second place");
assert(ninjas[2] === "Fuma",
       "And Fuma is in the third place");

removedItems = ninjas.splice(1, 2, "Mochizuki", "Yoshi", "Momochi");
assert(removedItems.length === 2, "Now, we've removed two items");
assert(removedItems[0] === "Hattori", "Hattori was removed");
assert(removedItems[1] === "Fuma", "Fuma was removed");
assert(ninjas.length === 4, "We've inserted some new items");
assert(ninjas[0] === "Yagyu", "Yagyu is still here");
assert(ninjas[1] === "Mochizuki", "Mochizuki also");
assert(ninjas[2] === "Yoshi", "Yoshi also");
assert(ninjas[3] === "Momochi", "and Momochi");
```

splice returns an array of the removed items. In this case, we removed one item.

The ninja array no longer contains Kuma; subsequent items were automatically shifted.

We can insert an element at a position by adding arguments to the splice call.

We begin by creating a new array with four items:

```
var ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];
```

Then we call the built-in splice method:

```
var removedItems = ninjas.splice(1,1);//ninjas:["Yagyu","Hattori", "Fuma"];
                                       //removedItems: ["Kuma"]
```

In this case, splice takes two arguments: the index from which the splicing starts, and the number of elements to be removed (if we leave out this argument, all elements to the end of the array are removed). In this case, the element with index 1 is removed from the array, and all subsequent elements are shifted accordingly.

In addition, the `splice` method returns an array of items that have been removed. In this case, the result is an array with a single item: `"Kuma"`.

Using the `splice` method, we can also insert items into arbitrary positions in an array. For example, consider the following code:

```
removedItems = ninjas.splice(1, 2, "Mochizuki", "Yoshi", "Momochi");
//ninjas: ["Yagyu", "Mochizuki", "Yoshi", "Momochi"]
//removedItems: ["Hattori", "Fuma"]
```

Starting from index `1`, it first removes two items and then adds three items: `"Mochizuki"`, `"Yoshi"`, and `"Momochi"`.

Now that we've given you a refresher on how arrays work, let's continue by studying some common operations that are often performed on arrays. These will help you write more elegant array-handling code.

9.1.4   *Common operations on arrays*

In this section, we'll explore some of the most common operations on arrays:

- *Iterating* (or *traversing*) through arrays
- *Mapping* existing array items to create a new array based on them
- *Testing* array items to check whether they satisfy certain conditions
- *Finding* specific array items
- *Aggregating* arrays and computing a single value based on array items (for example, calculating the sum of an array)
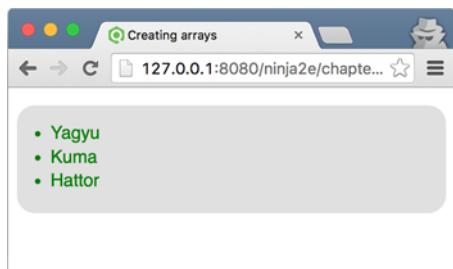
We'll start with the basics: array iterations.

ITERATING OVER ARRAYS

One of the most common operations is iterating over an array. Going back to Computer Science 101, an iteration is most often performed in the following way:

```
const ninjas = ["Yagyu", "Kuma", "Hattori"];

for(let i = 0; i < ninjas.length; i++){        ⟵  Reports the value
  assert(ninjas[i] !== null, ninjas[i]);            of each ninja
}
```

This example is as simple as it looks. It uses a `for` loop to check every item in the array; the results are shown in figure 9.4.

You've probably written something like this so many times that you don't even have to think about it anymore. But just in case, let's take a closer look at the `for` loop.

To go through an array, we have to set up a counter variable, `i`, specify the number up to which we want to count (`ninjas.length`), and

Figure 9.4   The output of checking the ninjas with a `for` loop

define how the counter will be modified (i++). That's an awful lot of bookkeeping to perform such a common action, and it can be a source of annoying little bugs. In addition, it makes our code more difficult to read. Readers have to look closely at every part of the `for` loop, just to be sure it goes through all the items and doesn't skip any.

To make life easier, all JavaScript arrays have a built-in `forEach` method we can use in such situations. Look at the following example.

Listing 9.5   Using the `forEach` method

```
const ninjas = ["Yagyu", "Kuma", "Hattori"];

ninjas.forEach(ninja => {            Uses the built-in
  assert(ninja !== null, ninja);     forEach method to
));                                   iterate over the array
```

We provide a callback (in this case, an arrow function) that's called *immediately*, for each item in the array. That's it—no more fussing about the start index, the end condition, or the exact nature of the increment. The JavaScript engine takes care of all that for us, behind the scenes. Notice how much easier to understand this code is, and how it has fewer bug-spawning points.

We'll continue by taking things up a notch and seeing how we can map arrays to other arrays.

MAPPING ARRAYS

Imagine that you have an array of `ninja` objects. Each ninja has a name and a favorite weapon, and you want to extract an array of weapons from the `ninjas` array. Armed with the knowledge of the `forEach` method, you might write something like the following listing.

Listing 9.6   Naïve extraction of a weapons array

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi", weapon: "katana"},
  {name: "Kuma", weapon: "wakizashi"}
];

const weapons = [];                  Creates a new array and uses
ninjas.forEach(ninja => {            a forEach loop over ninjas to
  weapons.push(ninja.weapon);        extract individual ninja weapons
});

assert(weapons[0] === "shuriken"
    && weapons[1] === "katana"
    && weapons[2] === "wakizashi"
    && weapons.length === 3,
    "The new array contains all weapons");
```

This isn't all that bad: We create a new, empty array, and use the `forEach` method to iterate over the `ninjas` array. Then, for each `ninja` object, we add the current weapon to the `weapons` array.

As you might imagine, creating new arrays based on the items in an existing array is surprisingly common—so common that it has a special name: *mapping* an array. The idea is that we map each item from one array to a new item of a new array. Conveniently, JavaScript has a `map` function that does exactly that, as shown in the following listing.

Listing 9.7   Mapping an array

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi", weapon: "katana"},
  {name: "Kuma", weapon: "wakizashi"}
];

const weapons = ninjas.map(ninja => ninja.weapon);     ◁──  The built-in map method
                                                             takes a function that's
assert(weapons[0] === "shuriken"                             called for each item in
    && weapons[1] === "katana"                               the array.
    && weapons[2] === "wakizashi"
    && weapons.length == 3, "The new array contains all weapons");
```

The built-in `map` method constructs a completely new array and then iterates over the input array. For each item in the input array, `map` places exactly one item in the newly constructed array, based on the result of the callback provided to `map`. The inner workings of the `map` function are shown in figure 9.5.
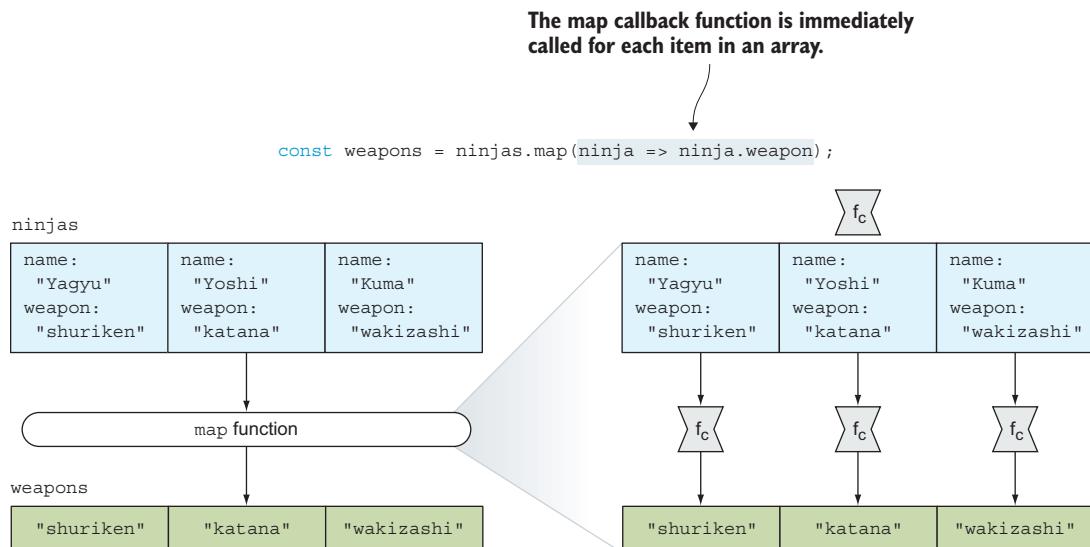
Figure 9.5   The `map` function calls the provided callback function (f_C) on each array item, and creates a new array with callback return values.

Now that we know how to map arrays, let's see how to test array items for certain conditions.

TESTING ARRAY ITEMS

When working with collections of items, we'll often run into situations where we need to know whether *all* or at least *some* of the array items satisfy certain conditions. To write this code as efficiently as possible, all JavaScript arrays have access to the built-in every and some methods, shown next.

Listing 9.8   Testing arrays with the `every` and `some` methods

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi" },
  {name: "Kuma", weapon: "wakizashi"}
];

const allNinjasAreNamed = ninjas.every(ninja => "name" in ninja);
const allNinjasAreArmed = ninjas.every(ninja => "weapon" in ninja);

assert(allNinjasAreNamed, "Every ninja has a name");
assert(!allNinjasAreArmed, "But not every ninja is armed");

const someNinjasAreArmed = ninjas.some(ninja => "weapon" in ninja);
assert(someNinjasAreArmed, "But some ninjas are armed");
```

The built-in every method takes a callback that's called for each array item. It returns true if the callback returns a true value for *all* array items, or false otherwise.

The built-in some method also takes a callback. It returns true if the callback returns a true value for at least one array item, or false otherwise.

Listing 9.8 shows an example where we have a collection of ninja objects but are unsure of their names and whether all of them are armed. To get to the root of this problem, we first take advantage of every:

```
var allNinjasAreNamed = ninjas.every(ninja => "name" in ninja);
```

The every method takes a callback that, for each ninja in the collection, checks whether we know the ninja's name. every returns true only if the passed-in callback returns true for *every* item in the array. Figure 9.6 shows how every works.

In other cases, we only care whether *some* array items satisfy a certain condition. For these situations, we can use the built-in method some:

```
const someNinjasAreArmed = ninjas.some(ninja => "weapon" in ninja);
```

Starting from the first array item, some calls the callback on each array item until an item is found for which the callback returns a true value. If such an item is found, the return value is true; if not, the return value is false.

**The every callback function is
immediately called for each item
in an array, until false is returned.**

```
const allNinjasAreNamed = ninjas.every(ninja => "name" in ninja);
```
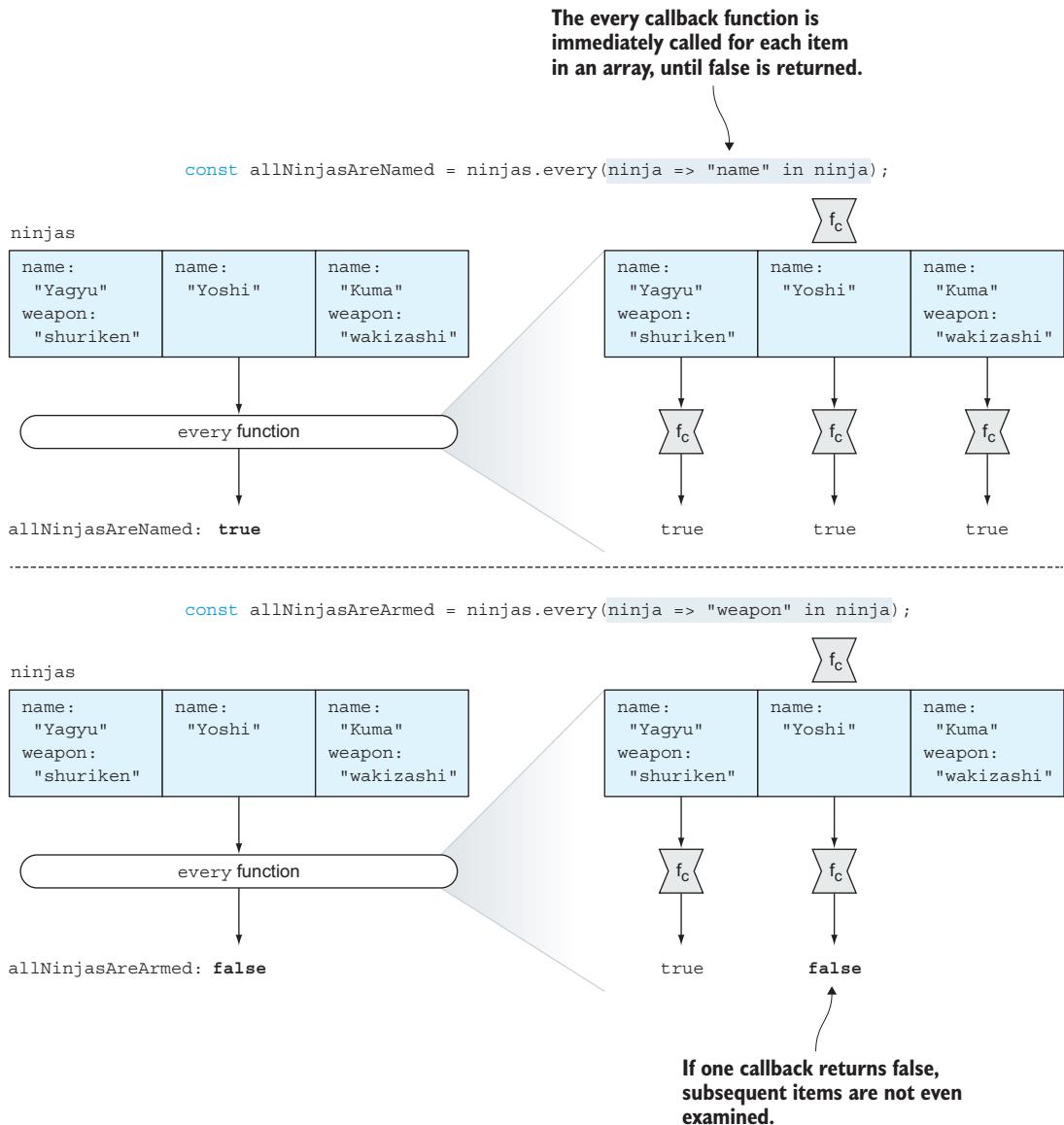

Figure 9.6 The `every` method tests whether all items in an array satisfy a certain condition represented by a callback.

Figure 9.7 shows how some works under the hood: We search an array in order to find out whether some or all of its items satisfy a certain condition. Next we'll explore how to search an array to find a particular item.

```
const someNinjasAreArmed = ninjas.some(ninja => "weapon" in ninja);
```
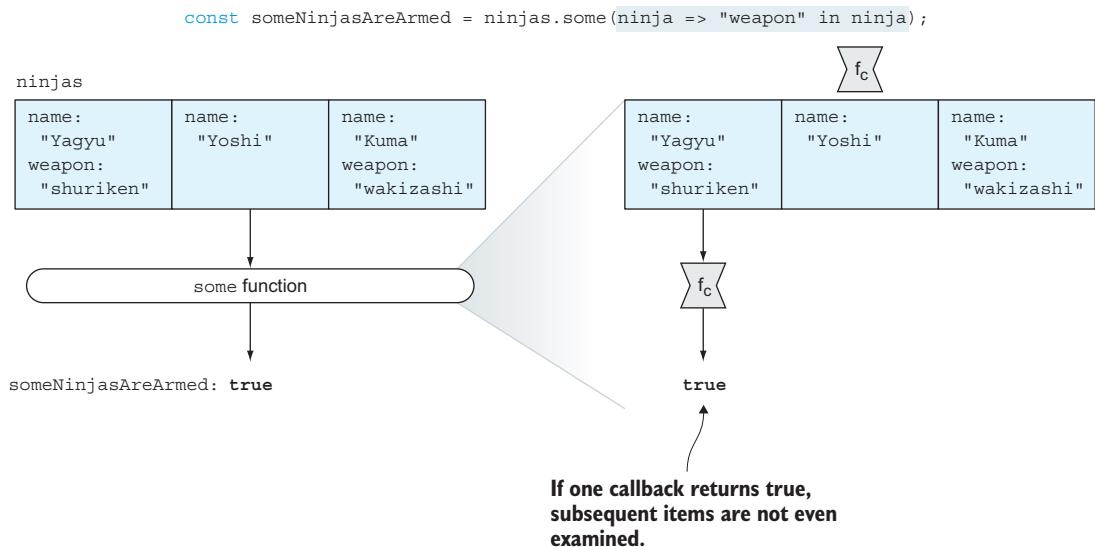

Figure 9.7 The `some` method checks whether at least one array item satisfies a condition represented by the passed-in callback.

SEARCHING ARRAYS

Another common operation that you're bound to use, sooner rather than later, is finding items in an array. Again, this task is greatly simplified with another built-in array method: `find`. Let's study the following listing.

> **NOTE** The built-in `find` method is part of the ES6 standard. For current browser compatibility, see http://mng.bz/U532.

Listing 9.9 Finding array items

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi" },
  {name: "Kuma", weapon: "wakizashi"}
];

const ninjaWithWakizashi = ninjas.find(ninja => {
  return ninja.weapon === "wakizashi";
});

assert(ninjaWithWakizashi.name === "Kuma"
    && ninjaWithWakizashi.weapon === "wakizashi",
    "Kuma is wielding a wakizashi");
```

Uses the find method to find the first array item that satisfies a certain condition, represented by a passed-in callback.

```
const ninjaWithKatana = ninjas.find(nina => {
  return ninja.weapon === "katana";
});
```

> The find method returns undefined if an item can't be found.

```
assert(ninjaWithKatana === undefined,
       "We couldn't find a ninja that wields a katana");

const armedNinjas = ninjas.filter(ninja => "weapon" in ninja);

assert(armedNinjas.length === 2, "There are two armed ninjas:");
assert(armedNinjas[0].name === "Yagyu"
    && armedNinjas[1].name === "Kuma", "Yagyu and Kuma");
```

> Use the filter method to find multiple items that all satisfy a certain condition.

It's easy to find an array item that satisfies a certain condition: We use the built-in `find` method, passing it a callback that's invoked for each item in the collection until the targeted item is found. This is indicated by the callback returning `true`. For example, the expression

```
ninjas.find(ninja => ninja.weapon === "wakizashi");
```

finds Kuma, the first ninja in the `ninjas` array that's wielding a wakizashi.

If we've gone through the entire array without a single item returning `true`, the final result of the search is `undefined`. For example, the code

```
ninjaWithKatana = ninjas.find(ninja => ninja.weapon === "katana");
```

returns `undefined`, because there isn't a katana-wielding ninja. Figure 9.8 shows the inner workings of the `find` function.
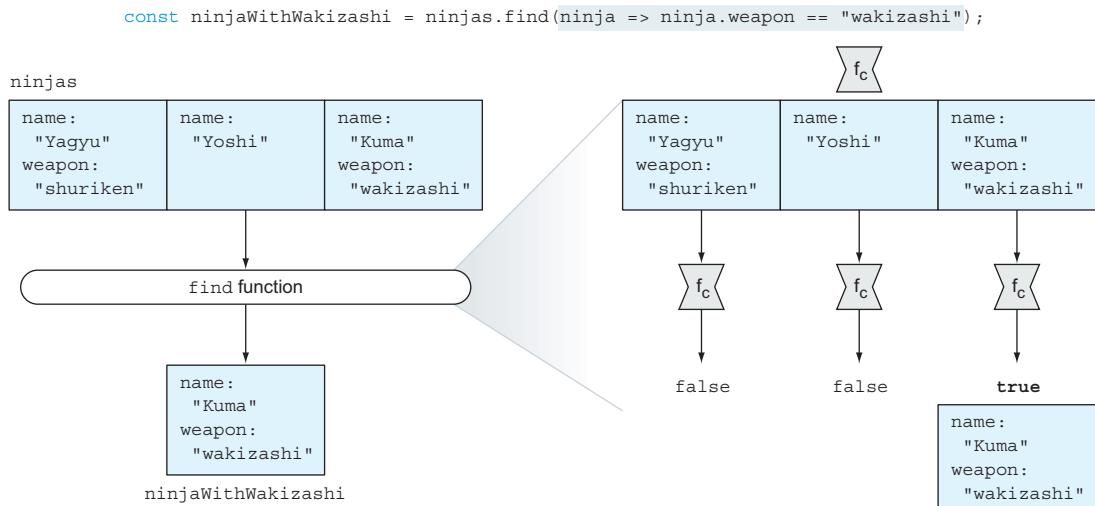
Figure 9.8 The `find` function finds one item in an array: the first item for which the `find` callback returns `true`.

If we need to find multiple items satisfying a certain criterion, we can use the `filter` method, which creates a *new* array containing all the items that satisfy that criterion. For example, the expression

```
const armedNinjas = ninjas.filter(ninja => "weapon" in ninja);
```

creates a new `armedNinjas` array that contains only `ninjas` with a weapon. In this case, poor unarmed Yoshi is left out. Figure 9.9 shows how the `filter` function works.
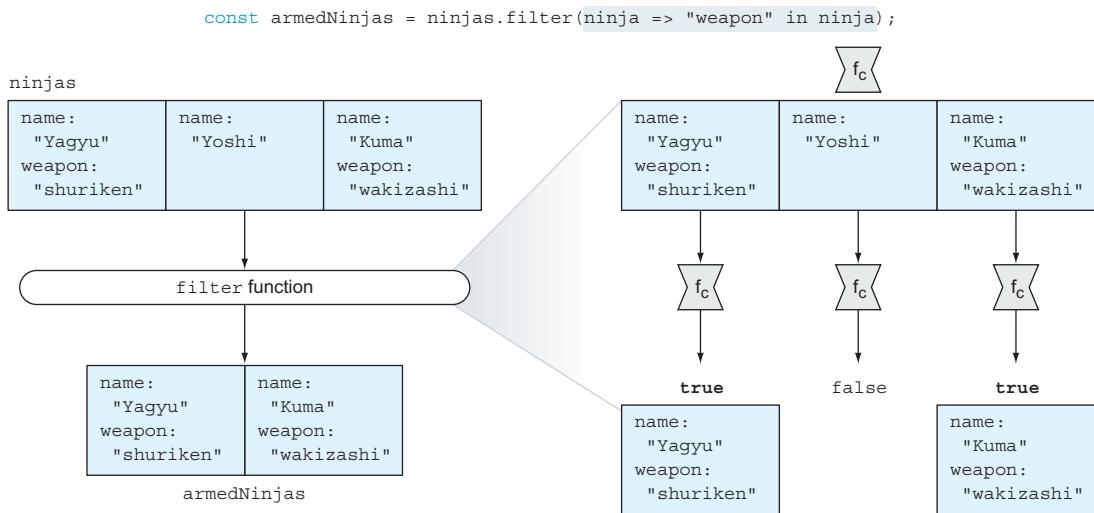
Figure 9.9 The `filter` function creates a new array that contains all items for which the callback returns `true`.

Throughout this example, you've seen how to find particular items in an array, but in many cases it might also be necessary to find the index of an item. Let's take a closer look, with the following example.

Listing 9.10   Finding array indexes

```
const ninjas = ["Yagyu", "Yoshi", "Kuma", "Yoshi"];

assert(ninjas.indexOf("Yoshi") === 1, "Yoshi is at index 1");
assert(ninjas.lastIndexOf("Yoshi") === 3, "and at index 3");

const yoshiIndex = ninjas.findIndex(ninja => ninja === "Yoshi");

assert(yoshiIndex === 1, "Yoshi is still at index 1");
```

To find the index of a particular item, we use the built-in `indexOf` method, passing it the item whose index we want to find:

```
ninjas.indexOf("Yoshi")
```

In cases where a particular item can be found multiple times in an array (as is the case with `"Yoshi"` and the `ninjas` array), we may also be interested in finding the last index where Yoshi appears. For this, we can use the `lastIndexOf` method:

```
ninjas.lastIndexOf("Yoshi")
```

Finally, in the most-general case, when we don't have a reference to the exact item whose index we want to search for, we can use the `findIndex` method:

```
const yoshiIndex = ninjas.findIndex(ninja => ninja === "Yoshi");
```

The `findIndex` method takes a callback and returns the index of the first item for which the callback returns `true`. In essence, it works a lot like the `find` method, the only difference being that `find` returns a particular item, whereas `findIndex` returns the index of that item.

SORTING ARRAYS

One of the most common array operations is *sorting*—arranging items systematically in some order. Unfortunately, correctly implementing sorting algorithms isn't the easiest of programming tasks: We have to select the best sorting algorithm for the task, implement it, and tailor it to our needs, while, as always, being careful not to introduce subtle bugs. To get this burden off our back, as you saw in chapter 3, all JavaScript arrays have access to the built-in `sort` method, whose usage looks something like this:

```
array.sort((a, b) => a - b);
```

The JavaScript engine implements the sorting algorithm. The only thing we have to provide is a callback that informs the sorting algorithm about the relationship between two array items. The possible results are as follows:

- If a callback returns a value less than `0`, then item `a` should come before item `b`.
- If a callback returns a value equal to `0`, then items `a` and `b` are on equal footing (as far as the sorting algorithm is concerned, they're equal).
- If a callback returns a value greater than `0`, then item `a` should come after item `b`.

Figure 9.10 shows the decisions made by the sorting algorithm depending on the callback return value.
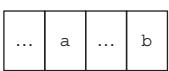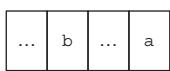
	... a ... b	... b ... a
`returnValue < 0` (a should come before b)	Leave as is	a should be moved before b
`returnValue == 0` (a and b are on equal footing)	Leave as is	Leave as is
`returnValue > 0` (b should come before a)	b should be moved before a	Leave as is

Figure 9.10   If the callback returns a value less than 0, the first item should come before the second. If the callback returns 0, both items should be left as is. And if the return value is greater than 0, the first item should come after the second item.

And that's about all you need to know about the sorting algorithm. The actual sorting is performed behind the scenes, without us having to manually move array items around. Let's look at a simple example.

Listing 9.11 Sorting an array

```
const ninjas = [{name: "Yoshi"}, {name: "Yagyu"}, {name: "Kuma"}];

ninjas.sort(function(ninja1, ninja2){
  if(ninja1.name < ninja2.name) { return -1; }          Passes a callback to the
  if(ninja1.name > ninja2.name) { return 1; }           built-in sort method to
                                                        specify a sorting order
  return 0;
});

assert(ninjas[0].name === "Kuma", "Kuma is first");
assert(ninjas[1].name === "Yagyu", "Yagyu is second");
assert(ninjas[2].name === "Yoshi", "Yoshi is third");
```

In listing 9.11 we have an array of `ninja` objects, where each ninja has a name. Our goal is to sort that array lexicographically (in alphabetical order), according to ninja names. For this, we naturally use the `sort` function:

```
ninjas.sort(function(ninja1, ninja2){
  if(ninja1.name < ninja2.name) { return -1; }
  if(ninja1.name > ninja2.name) { return 1; }

  return 0;
});
```

To the `sort` function we only need to pass a callback that's used to compare two array items. Because we want to make a lexical comparison, we state that if `ninja1`'s name is "less" than `ninja2`'s name, the callback returns `-1` (remember, this means `ninja1` should come before `ninja2`, in the final sorted order); if it's greater, the callback returns `1` (`ninja1` should come after `ninja2`); if they're equal, the callback returns `0`. Notice that we can use simple less-than (`<`) and greater-than (`>`) operators to compare two ninja names.

That's about it! The rest of the nitty-gritty details of sorting are left to the JavaScript engine, without us having to worry about them.

AGGREGATING ARRAY ITEMS

How many times have you written code like the following?

```
const numbers = [1, 2, 3, 4];
const sum = 0;

numbers.forEach(number => {
   sum += number;
});

assert(sum === 10, "The sum of first four numbers is 10");
```

This code has to visit every item in a collection and aggregate some value, in essence reducing the entire array to a single value. Don't worry—JavaScript has something to help with this situation, too: the reduce method, as shown in the following example.

Listing 9.12   Aggregating items with `reduce`

```
const numbers = [1, 2, 3, 4];

const sum = numbers.reduce((aggregated, number) =>          Uses reduce to accumulate a
                           aggregated + number, 0);          single value from an array

assert(sum === 10, "The sum of first four numbers is 10");
```

The reduce method works by taking the initial value (in this case, 0) and calling the callback function on each array item with the result of the previous callback invocation (or the initial value) and the current array item as arguments. The result of the reduce invocation is the result of the last callback, called on the last array item. Figure 9.11 sheds more light on the process.

We hope we've convinced you that JavaScript contains some useful array methods that can make our lives significantly easier and your code more elegant, without having to resort to pesky for loops. If you'd like to find out more about these and other array methods, we recommend the Mozilla Developer Network explanation at http://mng.bz/cS21.

Now we'll take things a bit further and show you how to reuse these array methods on your own, custom objects.

9.1.5   Reusing built-in array functions

There are times when we may want to create an object that contains a collection of data. If the collection was all we were worried about, we could use an array. But in certain cases, there may be more state to store than just the collection itself—perhaps we need to store some sort of metadata regarding the collected items.
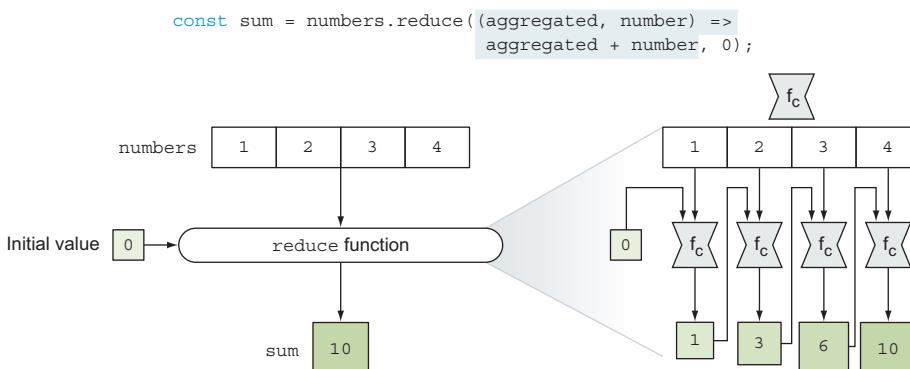
Figure 9.11   The `reduce` function applies a callback to an aggregated value and each item in an array to reduce the array to a single value.

One option may be to create a new array every time you wish to create a new version of such an object, and add the metadata properties and methods to it. Remember, we can add properties and methods to an object as we please, including arrays. Generally, however, this can be slow, not to mention tedious.

Let's examine the possibility of using a normal object and *giving* it the functionality we desire. Methods that know how to deal with collections already exist on the `Array` object; can we trick them into working on our own objects? Turns out that we can, as shown in the following listing.

Listing 9.13   Simulating array-like methods

```
<body>
  <input id="first"/>
  <input id="second"/>
  <script>
    const elems = {
      length: 0,
      add: function(elem){
        Array.prototype.push.call(this, elem);
      },
      gather: function(id){
        this.add(document.getElementById(id));
      },
      find: function(callback){
        return Array.prototype.find.call(this, callback);
      }
    };

    elems.gather("first");
    assert(elems.length === 1 && elems[0].nodeType,
          "Verify that we have an element in our stash");

    elems.gather("second");
    assert(elems.length === 2 && elems[1].nodeType,
          "Verify the other insertion");

    elems.find(elem => elem.id === "second");
    assert(found && found.id === "second",
          "We've found our element");
  </script>
</body>
```

Stores the count of elements. The array needs a place to store the number of items it's storing.

Implements the method to add elements to a collection. The prototype for Array has a method to do this, so why not use it instead of reinventing the wheel?

Implements the gather method to find elements by their id values and add them to the collection

Implements the method to find elements in the collection. Similar to the add method, it reuses the existing find method accessible to arrays.

In this example, we create a "normal" object and instrument it to mimic some of the behaviors of an array. First we define a `length` property to record the number of elements that are stored, just like an array. Then we define a method to add an element to the end of the simulated array, calling this method `add`:

```
add: function(elem){
  Array.prototype.push.call(this, elem);
}
```

Rather than write our own code, we can use a native method of JavaScript arrays: `Array.prototype.push`.

Normally, the `Array.prototype.push` method would operate on its own array via its function context. But here, we're tricking the method to use *our* object as its context by using the `call` method (remember chapter 4) and forcing our object to be the context of the `push` method. (Notice how we could've just as easily used the `apply` method.) The `push` method, which increments the `length` property (thinking that it's the `length` property of an array), adds a numbered property to the object referencing the passed element. In a way, this behavior is almost subversive (how fitting for ninjas!), but it exemplifies what we can do with mutable object contexts.

The `add` method expects an element reference to be passed for storage. Although sometimes we may have such a reference around, more often than not we won't, so we also define a convenience method, `gather`, that looks up the element by its `id` value and adds it to storage:

```
gather: function(id){
  this.add(document.getElementById(id));
}
```

Finally, we also define a `find` method that lets us find an arbitrary item in our custom object, by taking advantage of the built-in array `find` method:

```
find: function(callback){
  return Array.prototype.find.call(this, callback);
}
```

The borderline nefarious behavior we demonstrated in this section not only reveals the power that malleable function contexts give us, but also shows how we can be clever in reusing code that's already written, instead of constantly reinventing the wheel.

Now that we've spent some time with arrays, let's move on to two new types of collections introduced by ES6: maps and sets.

9.2   Maps

Imagine that you're a developer at freelanceninja.com, a site that wants to cater to a more international audience. For each piece of text on the website—for example, "Ninjas for hire"—you'd like to create a mapping to each targeted language, such as " レンタル用の忍者 " in Japanese, " 忍者出租 " in Chinese, or " 고용 닌자 " in Korean (let's hope Google Translate has done an adequate job). These collections, which map a key to a specific value, are called by different names in different programming languages, but most often they're known as *dictionaries* or *maps.*

But how do you efficiently manage this localization in JavaScript? One traditional approach is to take advantage of the fact that objects are collections of named properties and values, and create something like the following dictionary:

```
const dictionary = {
  "ja": {
    "Ninjas for hire": " レンタル用の忍者 "
  },
  "zh": {
    "Ninjas for hire": " 忍者出租 "
  },
  "ko": {
    "Ninjas for hire":" 고용 닌자 "
  }
}
assert(dictionary.ja["Ninjas for hire"] === " レンタル用の忍者 ");
```

At first glance, this may seem like a perfectly fine approach to this problem, and for this example, it isn't half bad. But unfortunately, in general, you can't rely on it.

9.2.1 *Don't use objects as maps*

Imagine that somewhere on our site we need to access the translation for the word *constructor*, so we extend the dictionary example into the following code.

Listing 9.14   Objects have access to properties that weren't explicitly defined

```
const dictionary = {
  "ja": {
    "Ninjas for hire": " レンタル用の忍者 "
  },
  "zh": {
    "Ninjas for hire": " 忍者出租 "
  },
  "ko": {
    "Ninjas for hire":" 고용 닌자 "
  }
};

assert(dictionary.ja["Ninjas for hire"] === " レンタル用の忍者 ",
      "We know how to say 'Ninjas for hire' in Japanese!");

assert(typeof dictionary.ja["constructor"] === "undefined",
      dictionary.ja["constructor"]);
```

We try to access the translation for the word *constructor*—a word that we foolishly forgot to define in our dictionary. Normally, in such a case, we'd expect the dictionary to return undefined. But that isn't the result, as you can see in figure 9.12.

As you can see, by accessing the constructor property, we obtain the following string:

```
"function Object() { [native code] }"
```
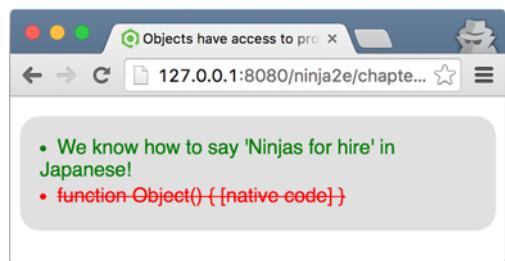

Figure 9.12   Running listing 9.14 shows that objects aren't good maps, because they have access to properties that weren't explicitly defined (through their prototypes).

What's with this? As you learned in chapter 7, all objects have prototypes; even if we define new, empty objects as our maps, they still have access to the properties of the prototype objects. One of those properties is `constructor` (recall that `constructor` is the property of the prototype object that points back to the constructor function), and it's the culprit behind the mess we now have on our hands.

In addition, with objects, keys can only be string values; if you want to create a mapping for any other value, that value will be silently converted into a string without anyone asking you anything! For example, imagine that we want to track some information about HTML nodes, as in the following listing.

Listing 9.15   Mapping values to HTML nodes with objects

Defines two HTML elements and fetches them by using the built-in document.getElementById method

Defines an object that we'll use as a map to store additional information about our HTML elements

Stores information about the first element, and checks that it was correctly stored

Stores information about the second element, and checks that it was correctly stored

The mapping for the first element is now invalid!

```
<div id="firstElement"></div>
<div id="secondElement"></div>
<script>
  const firstElement = document.getElementById("firstElement");
  const secondElement = document.getElementById("secondElement");

  const map = {};

  map[firstElement] = { data: "firstElement"};
  assert(map[firstElement].data === "firstElement",
     "The first element is correctly mapped");

  map[secondElement] = { data: "secondElement"};
  assert(map[secondElement].data === "secondElement",
     "The second element is correctly mapped");

  assert(map[firstElement].data === "firstElement",
     "But now the firstElement is overriden!");
</script>
```

In listing 9.15, we create two HTML elements, `firstElement` and `secondElement`, which we then fetch from the DOM by using the `document.getElementById` method. In order to create a mapping that will store additional information about each element, we define a plain old JavaScript object:

```
const map = {};
```

Then we use the HTML element as a key for our mapping object and associate some data with it:

```
map[firstElement] = { data: "firstElement"}
```

And we check that we can retrieve that data. Because that works as it should, we repeat the entire process for the second element:

```
map[secondElement] = { data: "secondElement"};
```

Again, everything looks hunky dory; we've successfully associated some data with our HTML element. But a problem occurs if we decide to revisit the first element:

```
map[firstElement].data
```

It would be normal to assume that we'd again obtain the information about the first element, but this isn't the case. Instead, as figure 9.13 shows, the information about the second element is returned.

This happens because with objects, keys are stored as strings. This means when we try to use any non-string value, such as an HTML element, as a property of an object, that value is silently converted to a string by calling its `toString` method. Here, this returns the string `"[object HTMLDivElement]"`, and the

Figure 9.13 **Running the code from listing 9.15 shows that objects are converted to strings if we try to use them as object properties.**

information about the first element is stored as the value of the [object HTMLDiv-Element] property.

Next, when we try to create a mapping for the second element, a similar thing happens. The second element, which is also an HTML div element, is also converted to a string, and its additional data is also assigned to the [object HTMLDivElement] property, overriding the value we set for the first element.

For these two reasons—properties inherited through prototypes and support for string-only keys—plain objects generally aren't useful as maps. Due to this limitation, the ECMAScript committee has specified a completely new type: `Map`.

> **NOTE** Maps are a part of the ES6 standard. For current browser compatibility, see: http://mng.bz/JYYM.

9.2.2 *Creating our first map*

Creating maps is easy: We use a new, built-in `Map` constructor. Look at the following example.

Listing 9.16   Creating our first map

**Uses the Map constructor
to create a map**

```
const ninjaIslandMap = new Map();          ◁
```

**Creates a mapping for the
first two ninja objects by
using the map set method**

```
const ninja1 = { name: "Yoshi"};
const ninja2 = { name: "Hattori"};      Defines three
const ninja3 = { name: "Kuma"};         ninja objects
```

```
ninjaIslandMap.set(ninja1, { homeIsland: "Honshu"});
ninjaIslandMap.set(ninja2, { homeIsland: "Hokkaido"});
```

**Gets the mapping
for the first two
ninja objects by
using the map get
method**

```
assert(ninjaIslandMap.get(ninja1).homeIsland === "Honshu",
      "The first mapping works");
assert(ninjaIslandMap.get(ninja2).homeIsland === "Hokkaido",
       "The second mapping works");
```

```
assert(ninjaIslandMap.get(ninja3) === undefined,
      "There is no mapping for the third ninja!");
```

**Checks that there's no mapping
for the third ninja**

```
assert(ninjaIslandMap.size === 2,
      "We've created two mappings");
```

**Checks that the map contains mappings for the
first two ninjas, but not for the third one!**

```
assert(ninjaIslandMap.has(ninja1)
    && ninjaIslandMap.has(ninja2),
      "We have mappings for the first two ninjas");
assert(!ninjaIslandMap.has(ninja3),
      "But not for the third ninja!");
```

**Uses the has method to
check whether a mapping
for a particular key exists**

```
ninjaIslandMap.delete(ninja1);
assert(!ninjaIslandMap.has(ninja1)
    && ninjaIslandMap.size() === 1,
     "There's no first ninja mapping anymore!");
```

**Uses the delete method
to delete a key from the
map**

```
ninjaIslandMap.clear();
assert(ninjaIslandMap.size === 0,
      "All mappings have been cleared");
```

**Uses the clear method to
completely clear the map**

In this example, we create a new map by calling the built-in Map constructor:

```
const ninjaIslandMap = new Map();
```

Next, we create three ninja objects, cleverly called ninja1, ninja2, and ninja3. We
then use the built-in map set method:

```
ninjaIslandMap.set(ninja1, { homeIsland: "Honshu"});
```

This creates a mapping between a key—in this case, the ninja1 object—and a value—
in this case, an object carrying the information about the ninja's home island. We do
this for the first two ninjas, ninja1 and ninja2.

In the next step, we obtain the mapping for the first two ninjas by using another built-in map method, get:

```
assert(ninjaIslandMap.get(ninja1).homeIsland === "Honshu",
       "The first mapping works");
```

The mapping of course exists for the first two ninjas, but it doesn't exist for the third ninja, because we haven't used the third ninja as an argument to the set method. The current state of the map is shown in figure 9.14.

In addition to get and set methods, every map also has a built-in size property and has and delete methods. The size property tells us how many mappings we've created. In this case, we've created only two mappings.

The has method, on the other hand, notifies us whether a mapping for a particular key already exists:

Figure 9.14 A map is a collection of key-value pairs, where a key can be anything—even another object.

```
ninjaIslandMap.has(ninja1); //true
ninjaIslandMap.has(ninja3); //false
```

The delete method enables us to remove items from our map:

```
ninjaIslandMap.delete(ninja1);
```

One of the fundamental concepts when dealing with maps is determining when two map keys are equal. Let's explore this concept.

KEY EQUALITY

If you come from a bit more traditional background, such as C#, Java, or Python, you may be surprised by the next example.

Listing 9.17 Key equality in maps

```
const map = new Map();
const currentLocation = location.href;        ◁   Uses the built-in location.href property
                                                   to get the current page URL

const firstLink = new URL(currentLocation);       Creates two links to
const secondLink = new URL(currentLocation);      the current page

map.set(firstLink, { description: "firstLink"});     Adds a mapping
map.set(secondLink, { description: "secondLink"});   for both links
```

```
assert(map.get(firstLink).description === "firstLink",
     "First link mapping" );
assert(map.get(secondLink).description === "secondLink",
      "Second link mapping");
assert(map.size === 2, "There are two mappings");
```

Each link gets its own mapping, even though they point to the same page.

In listing 9.17, we use the built-in location.href property to obtain the URL of the current page. Next, by using the built-in URL constructor, we create two new URL objects that link to the current page. We then associate a description object with each link. Finally, we check that the correct mappings have been created, as shown in figure 9.15.

People who have mostly worked in JavaScript may not find this result unexpected: We have two different objects for which we create two differ-

Figure 9.15   **If we run the code from listing 9.17, we can see that key equality in maps is based on object equality.**

ent mappings. But notice that the two URL objects, even though they're separate objects, still point to the same URL location: the location of the current page. We could argue that, when creating mappings, these two objects should be considered equal. But in JavaScript, we can't overload the equality operator, and the two objects, even though they have the same content, are always considered different. This isn't the case with other languages, such as Java and C#, so be careful!

9.2.3 *Iterating over maps*

So far, you've seen some of the advantages of maps: You can be sure they contain only items that you put in them, and you can use anything as a key. But there's more!

Because maps are collections, there's nothing stopping us from iterating over them with for...of loops. (Remember, we used the for...of loop to iterate over values created by generators in chapter 6.) You're also guaranteed that these values will be visited in the order in which they were inserted (something we can't rely on when iterating over objects using the for...in loop). Let's look at the following example.

Listing 9.18   Iterating over maps

```
const directory = new Map();

directory.set("Yoshi", "+81 26 6462");
directory.set("Kuma", "+81 52 2378 6462");
directory.set("Hiro", "+81 76 277 46");
```

Creates a new map, just as we've done so far

Creates a ninja directory that stores each ninja's phone number

```
for(let item of directory){
  assert(item[0] !== null, "Key:" + item[0]);
  assert(item[1] !== null, "Value:" + item[1]);
}
```

> **Iterates over each item in a dictionary using the for...of loop. Each item is a two-item array: a key and a value.**

```
for(let key of directory.keys()){
  assert(key !== null, "Key:" + key);
  assert(directory.get(key) != null,
        "Value:" + directory.get(key));
}
```

> **We can also iterate over keys using the built-in keys method…**

```
for(var value of directory.values()){
  assert(value !== null, "Value:" + value);
}
```

> **…and over values using the built-in values method.**

As the previous listing shows, once we've created a mapping, we can easily iterate over it using the `for...of` loop:

```
for(var item of directory){
  assert(item[0] !== null, "Key:" + item[0]);
  assert(item[1] !== null, "Value:" + item[1]);
}
```

In each iteration, this gives a two-item array, where the first item is a key and the second item is the value of an item from our directory map. We can also use the `keys` and `values` methods to iterate over, well, keys and values contained in a map.

Now that we've looked at maps, let's visit another newcomer to JavaScript: *sets*, which are collections of unique items.

9.3   Sets

In many real-world problems, we have to deal with collections of *distinct* items (meaning each item can't appear more than once) called *sets*. Up to ES6, this was something you had to implement yourself by mimicking sets with standard objects. For a crude example, see the next listing.

Listing 9.19   Mimicking sets with objects

```
function Set(){
  this.data = {};
  this.length = 0;
}
```

> **Uses an object to store items**

```
Set.prototype.has = function(item){
  return typeof this.data[item] !== "undefined";
};
```

> **Checks whether the item is already stored**

```
Set.prototype.add = function(item){
  if(!this.has(item)){
    this.data[item] = true;
    this.length++;
  }
};
```

> **Adds an item only if it isn't already contained in the set**

```
Set.prototype.remove = function(item){
  if(this.has(item)){
    delete this.data[item];
    this.length--;
  }
};
```
Removes an item if it's already contained in the set

```
const ninjas = new Set();
ninjas.add("Hattori");
ninjas.add("Hattori");
```
Tries to add Hattori twice

```
assert(ninjas.has("Hattori") && ninjas.length === 1,
       "Our set contains only one Hattori");
```
Checks that Hattori was added only once

```
ninjas.remove("Hattori");
assert(!ninjas.has("Hattori") && ninjas.length === 0,
       "Our set is now empty");
```
Removes Hattori and checks that he was removed from the set

Listing 9.19 shows a simple example of how sets can be mimicked with objects. We use a data-storage object, `data`, to keep track of our set items, and we expose three methods: `has`, which checks whether an item is already contained in the set; `add`, which adds an item only if the same item isn't already contained in the set; and `remove`, which removes an already-contained item from the set.

But this is a poor doppelganger. Because with maps, you can't really store objects—only strings and numbers—and there's always the risk of accessing prototype objects. For these reasons, the ECMAScript committee decided to introduce a completely new type of collection: *sets.*

> **NOTE**   Sets are a part of the ES6 standard. For current browser compatibility, see http://mng.bz/QRTS.

9.3.1   Creating our first set

The cornerstone of creating sets is the newly introduced constructor function, conveniently named `Set`. Let's look at an example.

Listing 9.20   Creating a set

The Set constructor can take an array of items with which the set is initialized.

```
const ninjas = new Set(["Kuma", "Hattori", "Yagyu", "Hattori"]);

assert(ninjas.has("Hattori"), "Hattori is in our set");
assert(ninjas.size === 3, "There are only three ninjas in our set!");
```

Discards any duplicate items

```
assert(!ninjas.has("Yoshi"), "Yoshi is not in, yet..");
ninjas.add("Yoshi");
assert(ninjas.has("Yoshi"), "Yoshi is added");
assert(ninjas.size === 4, "There are four ninjas in our set!");

assert(ninjas.has("Kuma"), "Kuma is already added");
ninjas.add("Kuma");
assert(ninjas.size === 4, "Adding Kuma again has no effect");

for(let ninja of ninjas) {
  assert(ninja !== null, ninja);
}
```

> **We can add new items that aren't already contained in the set.**

> **Adding existing items has no effect.**

> **Iterates through the set with a for...of loop**

Here we use the built-in `Set` constructor to create a new `ninjas` set that will contain distinct ninjas. If we don't pass in any arguments, an empty set is created. We can also pass in an array, such as this, which pre-fills the set:

```
new Set(["Kuma", "Hattori", "Yagyu", "Hattori"]);
```

As we already mentioned, sets are collections of unique items, and their primary purpose is to stop us from storing multiple occurrences of the same object. In this case, this means `"Hattori"`, which we tried to add twice, is added only once.

A number of methods are accessible from every set. For example, the `has` method checks whether an item is contained in the set:

```
ninjas.has("Hattori")
```

and the `add` method is used to add unique items to the set:

```
ninjas.add("Yoshi");
```

If you're curious about how many items are in a set, you can always use the `size` property.

Similar to maps and arrays, sets are collections, so we can iterate over them with a `for...of` loop. As you can see in figure 9.16, the items are always iterated over in the order in which they were inserted.

Now that we've gone through the basics of sets, let's visit some common operations on sets: unions, intersections, and differences.

Figure 9.16   Running the code from listing 9.20 shows that the items in a set are iterated over in the order in which they were inserted.

9.3.2   Union of sets

A union of two sets, `A` and `B`, creates a new set that contains all elements from both `A` and `B`. Naturally, each item can't occur more than once in the new set.

Listing 9.21   Using sets to perform a union of collections

Creates an array of ninjas and
samurai. Notice that Hattori is
both a ninja and a samurai.

```
const ninjas = ["Kuma", "Hattori", "Yagyu"];
const samurai = ["Hattori", "Oda", "Tomoe"];
```

Creates a new set of
warriors by spreading
ninjas and samurai

```
const warriors = new Set([...ninjas, ...samurai]);
```

```
assert(warriors.has("Kuma"), "Kuma is here");
assert(warriors.has("Hattori"), "And Hattori");
assert(warriors.has("Yagyu"), "And Yagyu");
assert(warriors.has("Oda"), "And Oda");
assert(warriors.has("Tomoe"), "Tomoe, last but not least");
```

All the ninjas
and samurai are
included in the
new warriors set.

```
assert(warriors.size === 5, "There are 5 warriors in total");
```

There are no duplicates in the new set. Even though Hattori is
in both the ninjas and samurai sets, he is included only once.

We first create an array of ninjas and an array of samurai. Notice that Hattori is lead-
ing a busy life: samurai by day, ninja by night. Now imagine that we need to create a
collection of people whom we can call to arms if a neighboring daimyo decides that
his province is a bit cramped. We create a new set, warriors, that includes all ninjas
and all samurai. Hattori is in both collections, but we want to include him only once—
it's not like two Hattoris will respond to our call.

In this case, a set is perfect! We don't need to manually keep track of whether an item
has been already included: The set takes care of that by itself, automatically. When cre-
ating this new set, we use the spread operator [...ninjas, ...samurai] (remember
chapter 3) to create a new array that contains all ninjas and all samurai. In case you're
wondering, Hattori is present twice in this new array. But when we finally pass that array
to the Set constructor, Hattori is included only once, as shown in figure 9.17.
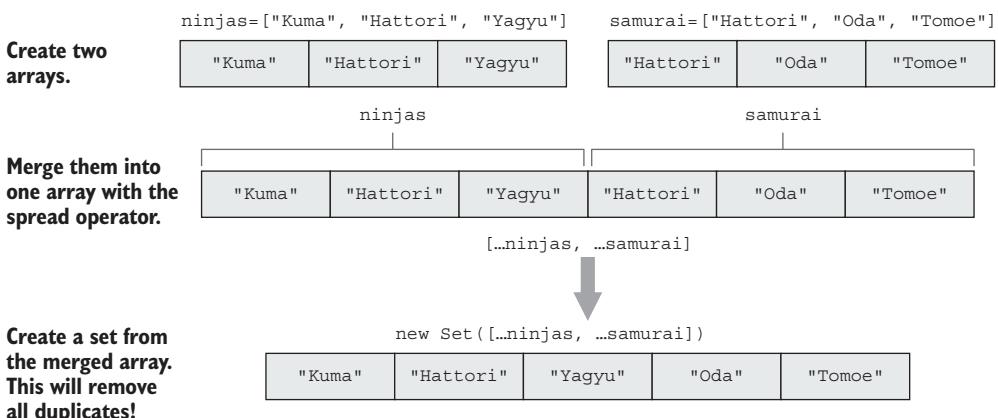
Figure 9.12   A union of two sets keeps the items from both collections (without duplicates).

9.3.3 *Intersection of sets*

The *intersection* of two sets, A and B, creates a set that contains elements of A that are also in B. For example, we can find ninjas that are also samurai, as shown next.

Listing 9.22 Intersection of sets

```
const ninjas = new Set(["Kuma", "Hattori", "Yagyu"]);
const samurai = new Set(["Hattori", "Oda", "Tomoe"]);

const ninjaSamurais = new Set(
  [...ninjas].filter(ninja => samurai.has(ninja))   ◁
);

assert(ninjaSamurais.size === 1, "There's only one ninja samurai");
assert(ninjaSamurais.has("Hattori"), "Hattori is his name");
```

> **Uses the spread operator to turn our set into an array so we can use the array's filter method to keep only ninjas that are contained in the samurai set**

The idea behind listing 9.22 is to create a new set that contains only ninjas who are also samurai. We do this by taking advantage of the array's `filter` method, which, as you'll remember, creates a new array that contains only the items that match a certain criterion. In this case, the criterion is that the ninja is also a samurai (is contained in the set of samurai). Because the `filter` method can only be used on arrays, we have to turn the `ninjas` set into an array by using the spread operator:

```
[...ninjas]
```

Finally, we check that we've found only one ninja who's also a samurai: the Jack of all trades, Hattori.

9.3.4 *Difference of sets*

The difference of two sets, A and B, contains all elements that are in set A but are *not* in set B. As you might guess, this is similar to the intersection of sets, with one small but significant difference. In the next listing, we want to find only true ninjas (not those who also moonlight as samurai).

Listing 9.23 Difference of sets

```
const ninjas = new Set(["Kuma", "Hattori", "Yagyu"]);
const samurai = new Set(["Hattori", "Oda", "Tomoe"]);

const pureNinjas = new Set(
  [...ninjas].filter(ninja => !samurai.has(ninja))   ◁
);

assert(pureNinjas.size === 2, "There's only one ninja samurai");
assert(pureNinjas.has("Kuma"), "Kuma is a true ninja");
assert(pureNinjas.has("Yagyu"), "Yagyu is a true ninja");
```

> **With set difference, we care only about ninjas who are not samurai!**

The only change is to specify that we care only about the ninjas who are *not* also samurai, by putting an exclamation mark (`!`) before the `samurai.has(ninja)` expression.

9.4   *Summary*

- Arrays are a special type of object with a `length` property and `Array.prototype` as their prototype.
- We can create new arrays using the array literal notation (`[]`) or by calling the built-in `Array` constructor.
- We can modify the contents of an array using several methods accessible from array objects:
  - The built-in `push` and `pop` methods add items to and remove items from the end of the array.
  - The built-in `shift` and `unshift` methods add items to and remove items from the beginning of the array.
  - The built-in `splice` method can be used to remove items from and add items to arbitrary array positions.
- All arrays have access to a number of useful methods:
  - The `map` method creates a new array with the results of calling a callback on every element.
  - The `every` and `some` methods determine whether all or some array items satisfy a certain criterion.
  - The `find` and `filter` methods find array items that satisfy a certain condition.
  - The `sort` method sorts an array.
  - The `reduce` method aggregates all items in an array into a single value.
- You can reuse the built-in array methods when implementing your own objects by explicitly setting the method call context with the `call` or `apply` method.
- Maps and dictionaries are objects that contain mappings between a key and a value.
- Objects in JavaScript are lousy maps because you can only use string values as keys and because there's always the risk of accessing prototype properties. Instead, use the new built-in `Map` collection.
- Maps are collections and can be iterated over using the `for...of` loop.
- Sets are collections of unique items.

9.5   *Exercises*

1 What will be the content of the `samurai` array, after running the following code?

```
const samurai = ["Oda", "Tomoe"];
samurai[3] = "Hattori";
```

2 What will be the content of the `ninjas` array, after running the following code?

```
const ninjas = [];

ninjas.push("Yoshi");
```

```
ninjas.unshift("Hattori");

ninjas.length = 3;

ninjas.pop();
```

3 What will be the content of the `samurai` array, after running the following code?

```
const samurai = [];

samurai.push("Oda");
samurai.unshift("Tomoe");
samurai.splice(1, 0, "Hattori", "Takeda");
samurai.pop();
```

4 What will be stored in variables `first`, `second`, and `third`, after running the following code?

```
const ninjas = [{name:"Yoshi", age: 18},
                {name:"Hattori", age: 19},
                {name:"Yagyu", age: 20}];

const first = persons.map(ninja => ninja.age);
const second = first.filter(age => age % 2 == 0);
const third = first.reduce((aggregate, item) =>  aggregate + item, 0);
```

5 What will be stored in variables `first` and `second`, after running the following code?

```
const ninjas = [{ name: "Yoshi", age: 18 },
                { name: "Hattor", age: 19 },
                { name: "Yagyu", age: 20 }];

const first = ninjas.some(ninja => ninja.age % 2 == 0);
const second = ninjas.every(ninja => ninja.age % 2 == 0);
```

6 Which of the following assertions will pass?

```
const samuraiClanMap = new Map();

const samurai1 = { name: "Toyotomi"};
const samurai2 = { name: "Takeda"};
const samurai3 = { name: "Akiyama"};

const oda = { clan: "Oda"};
const tokugawa = { clan: "Tokugawa"};
const takeda ={clan: "Takeda"};

samuraiClanMap.set(samurai1, oda);
samuraiClanMap.set(samurai2, tokugawa);
samuraiClanMap.set(samurai2, takeda);
```

```
assert(samuraiClanMap.size === 3, "There are three mappings");
assert(samuraiClanMap.has(samurai1), "The first samurai has a mapping");
assert(samuraiClanMap.has(samurai3), "The third samurai has a mapping");
```

7 Which of the following assertions will pass?

```
const samurai = new Set("Toyotomi", "Takeda", "Akiyama", "Akiyama");
assert(samurai.size === 4, "There are four samurai in the set");

samurai.add("Akiyama");
assert(samurai.size === 5, "There are five samurai in the set");

assert(samurai.has("Toyotomi", "Toyotomi is in!");
assert(samurai.has("Hattori", "Hattori is in!");
```

Wrangling regular expressions

10

This chapter covers

- A refresher on regular expressions
- Compiling regular expressions
- Capturing with regular expressions
- Working with frequently encountered idioms

Regular expressions are a necessity of modern development. There, we said it. Although many a web developer could go through life happily ignoring regular expressions, some problems that need to be solved in JavaScript code can't be addressed elegantly without regular expressions.

Sure, there may be other ways to solve the same problems. But frequently, something that might take a half-screen of code can be distilled down to a single statement with the proper use of regular expressions. All JavaScript ninjas need regular expressions as an essential part of their toolkits.

Regular expressions trivialize the process of tearing apart strings and looking for information. Everywhere you look in mainstream JavaScript libraries, you'll see the prevalent use of regular expressions for various spot tasks:

- Manipulating strings of HTML nodes
- Locating partial selectors within a CSS selector expression
- Determining whether an element has a specific class name

- Input validation
- And more

Let's start by looking at an example.

> **TIP**   Becoming fluent in regular expressions requires a lot of practice. You might find a site such as JS Bin (http://jsbin.com) handy for playing around with examples. A couple of sites are dedicated to regular expression testing, such as the Regular Expression Test Page for JavaScript (www.regexplanet.com/advanced/javascript/index.html) and regex101 (www.regex101.com/#javascript). regex101 is an especially useful site for beginners, because it also automatically generates explanations for the targeted regular expression.

. . . .

	When would you prefer to use a RegExp literal over a RegExp object?
Do you know?	What is sticky matching, and how do you enable it?
	How does matching differ when using a global versus a non-global regular expression?

. . . .

10.1 *Why regular expressions rock*

Let's say we want to validate that a string, perhaps entered into a form by a website user, follows the format for a nine-digit U.S. postal code. We all know that the U.S. Postal Service has little sense of humor and insists that a postal code (also known as a ZIP code) follows this specific format:

99999-9999

Each 9 represents a decimal digit, and the format is 5 decimal digits, followed by a hyphen, followed by 4 decimal digits. If you use any other format, your package or letter gets diverted into the black hole of the hand-sorting department, and good luck predicting how long it will take to emerge again.

Let's create a function that, given a string, verifies that the U.S. Postal Service will stay happy. We could resort to performing a comparison on each character, but we're a ninja and that's too inelegant a solution, resulting in a lot of needless repetition. Rather, consider the following solution.

Listing 10.1   Testing for a specific pattern in a string

```
function isThisAZipCode(candidate) {
  if (typeof candidate !== "string" ||
      candidate.length != 10) return false;
  for (let n = 0; n < candidate.length; n++) {
    let c = candidate[n];
```

Short-circuits obviously bogus candidates

```
switch (n) {                              Performs tests based
  case 0: case 1: case 2: case 3: case 4: on character index
  case 6: case 7: case 8: case 9:
    if (c < '0' || c > '9') return false;
    break;
  case 5:
    if (c != '-') return false;
    break;
}
}                                         If all succeeds,
return true;                              we're good!
}
```

This code takes advantage of the fact that we have only two checks to make, depend-ing on the position of the character within the string. We still need to perform up to nine comparisons at runtime, but we have to write each comparison only once.

Even so, would anyone consider this solution *elegant*? It's more elegant than the brute-force, noniterative approach, but it still seems like an awful lot of code for such a simple check. Now consider this approach:

```
function isThisAZipCode(candidate) {
  return /^\d{5}-\d{4}$/.test(candidate);
}
```

Except for some esoteric syntax in the body of the function, that's a lot more succinct and elegant, no? That's the power of regular expressions, and it's just the tip of the iceberg. Don't worry if that syntax looks like someone's pet iguana walked across the keyboard; we're about to recap regular expressions before you learn how to use them in ninja-like fashion on your pages.

10.2 A regular expression refresher

Much as we'd like to, we can't offer you an exhaustive tutorial on regular expressions in the space we have. After all, entire books have been dedicated to regular expres-sions. But we'll do our best to hit all the important points.

For more detail than we can offer in this chapter, the books *Mastering Regular Expressions* by Jeffrey E. F. Friedl, *Introducing Regular Expressions* by Michael Fitzgerald, and *Regular Expressions Cookbook* by Jan Goyvaerts and Steven Levithan, all from O'Reilly, are popular choices.

Let's dig in.

10.2.1 Regular expressions explained

The term *regular expression* stems from mid-century mathematics, when a mathematician named Stephen Kleene described models of computational automata as "regular sets." But that won't help us understand anything about regular expressions, so let's simplify things and say that a regular expression is a way to express a *pattern* for matching strings of text. The expression itself consists of terms and

operators that allow us to define these patterns. We'll see what those terms and operators consist of shortly.

In JavaScript, as with most other object types, we have two ways to create a regular expression:

- Via a regular expression literal
- By constructing an instance of a `RegExp` object

For example, if we want to create a mundane regular expression (or *regex*, for short) that matches the string `test` exactly, we could do so with a regex literal:

```
const pattern = /test/;
```

That might look strange, but regex literals are delimited with forward slashes in the same way that string literals are delimited with quote characters.

Alternatively, we could construct a `RegExp` instance, passing the regex as a string:

```
const pattern = new RegExp("test");
```

Both formats result in the same regex being created in the variable `pattern`.

> **TIP** The literal syntax is preferred when the regex is known at development time, and the constructor approach is used when the regex is constructed at runtime by building it up dynamically in a string.

One of the reasons that the literal syntax is preferred over expressing regexes in a string is that (as you'll soon see) the backslash character plays an important part in regular expressions. But the backslash character is *also* the escape character for string literals, so to express a backslash within a string literal, we need to use a double backslash (\\). This can make regular expressions, which already possess a cryptic syntax, even more odd-looking when expressed within strings.

In addition to the expression itself, five flags can be associated with a regex:

- `i`—Makes the regex case-insensitive, so `/test/i` matches not only *test*, but also *Test, TEST, tEsT*, and so on.
- `g`—Matches all instances of the pattern, as opposed to the default of *local*, which matches only the first occurrence. More on this later.
- `m`—Allows matches across multiple lines, as might be obtained from the value of a `textarea` element.
- `y`—Enables sticky matching. A regular expression performs sticky matching in the target string by attempting to match from the last match position.
- `u`—Enables the use of Unicode point escapes (\u{...}).

These flags are appended to the end of the literal (for example, `/test/ig`) or passed in a string as the second parameter to the `RegExp` constructor (`new RegExp("test", "ig")`).

Matching the exact string *test* (even in a case-insensitive manner) isn't interesting—after all, we can do that particular check with a simple string comparison. So let's take a look at the terms and operators that give regular expressions their immense power to match more compelling patterns.

10.2.2 Terms and operators

Regular expressions, like most other expressions we're familiar with, are made up of terms and operators that qualify those terms. In the sections that follow, you'll see how these terms and operators can be used to express patterns.

EXACT MATCHING

Any character that's not a special character or operator (which we'll introduce as we go along) must appear literally in the expression. For example, in our `/test/` regex, four terms represent characters that must appear literally in a string for it to match the expressed pattern.

Placing such characters one after the other implicitly denotes an operation that means *followed by*. So `/test/` means *t* followed by *e* followed by *s* followed by *t*.

MATCHING FROM A CLASS OF CHARACTERS

Many times, we won't want to match a specific literal character, but a character from a finite set of characters. We can specify this with the set operator (also called the *character class* operator) by placing the set of characters that we want to match in square brackets: `[abc]`.

The preceding example signifies that we want to match any of the characters a, b, or c. Note that even though this expression spans five characters (three letters and two brackets), it matches only a single character in the candidate string.

Other times, we want to match anything *but* a finite set of characters. We can specify this by placing a caret character (`^`) right after the opening bracket of the set operator:

```
[^abc]
```

This changes the meaning to any character *but* a, b, or c.

There's one more invaluable variation to the set operation: the ability to specify a range of values. For example, if we want to match any one of the lowercase characters between *a* and *m*, we could write `[abcdefghijklm]`. But we can express that much more succinctly as follows:

```
[a-m]
```

The dash indicates that all characters from a through m inclusive (and lexicographically) are included in the set.

ESCAPING

Not all characters represent their literal equivalent. Certainly all of the alphabetic and decimal digit characters represent themselves, but as you'll see, special characters such as `$` and the period (`.`) represent either matches to something other than themselves,

or operators that qualify the preceding term. In fact, you've already seen how the [,], -, and ^ characters are used to represent something other than their literal selves.

How do we specify that we want to match a literal [or $ or ^ or other special character? Within a regex, the backslash character escapes whatever character follows it, making it a literal match term. So \[specifies a literal match to the [character, rather than the opening of a character class expression. A double backslash (\\) matches a single backslash.

BEGINS AND ENDS

Frequently, we may want to ensure that a pattern matches at the beginning of a string, or perhaps at the end of a string. The caret character, when used as the first character of the regex, anchors the match at the beginning of the string, such that /^test/ matches only if the substring test appears at the beginning of the string being matched. (Note that this is an overload of the ^ character, because it's also used to negate a character class set.)

Similarly, the dollar sign ($) signifies that the pattern must appear at the end of the string:

```
/test$/
```

Using both ^ and $ indicates that the specified pattern must encompass the entire candidate string:

```
/^test$/
```

REPEATED OCCURRENCES

If we want to match a series of four a characters, we might express that with /aaaa/, but what if we want to match *any* number of the same character? Regular expressions enable us to specify several repetition options:

- To specify that a character is optional (it can appear either once or not at all), follow it with ?. For example, /t?est/ matches both *test* and *est.*
- To specify that a character should appear one or many times, use +, as in /t+est/, which matches *test, ttest,* and *tttest,* but not *est.*
- To specify that the character appears *zero, one,* or *many* times, use *, as in /t*est/, which matches *test, ttest, tttest, and est.*
- To specify a fixed number of repetitions, indicate the number of allowed repetitions between braces. For example, /a{4}/ indicates a match on four consecutive a characters.
- To specify a range for the repetition count, indicate the range with a comma separator. For example, /a{4,10}/ matches any string of 4 through 10 consecutive a characters.
- To specify an open-ended range, omit the second value in the range (but leave the comma). The regex /a{4,}/ matches any string of four or more consecutive a characters.

Any of these repetition operators can be *greedy* or *nongreedy*. By default, they're greedy: They will consume all the possible characters that make up a match. Annotating the operator with a ? character (an overload of the ? operator), as in a+?, makes the operation nongreedy: It will consume *only* enough characters to make a match.

For example, if we're matching against the string aaa, the regular expression /a+/ would match all three a characters, whereas the nongreedy expression /a+?/ would match only one a character, because a single a character is all that's needed to satisfy the a+ term.

PREDEFINED CHARACTER CLASSES

Some characters that we might want to match are impossible to specify with literal characters (for example, control characters such as a carriage return). In addition, often we might want to match character classes, such as a set of decimal digits, or a set of whitespace characters. The regular expression syntax provides predefined terms that represent these characters or commonly used classes so that we can use control-character matching in our regular expressions and don't need to resort to the character class operator for commonly used sets of characters.

Table 10.1 lists these terms and the character or set they represent. These predefined sets help keep our regular expressions from looking excessively cryptic.

Table 10.1 **Predefined character classes and character terms**

Predefined term	Matches
\t	Horizontal tab
\b	Backspace
\v	Vertical tab
\f	Form feed
\r	Carriage return
\n	Newline
\cA : \cZ	Control characters
\u0000 : \uFFFF	Unicode hexadecimal
\x00 : \xFF	ASCII hexadecimal
.	Any character, except for whitespace characters (\s)
\d	Any decimal digit; equivalent to [0-9]
\D	Any character but a decimal digit; equivalent to [^0-9]
\w	Any alphanumeric character including underscore; equivalent to [A-Za-z0-9_]
\W	Any character but alphanumeric and underscore characters; equivalent to [^A-Za-z0-9_]

Table 10.1   **Predefined character classes and character terms** *(continued)*

Predefined term	Matches
\s	Any whitespace character (space, tab, form feed, and so on)
\S	Any character but a whitespace character
\b	A word boundary
\B	Not a word boundary (inside a word)

GROUPING

So far, you've seen that operators (such as + and *) affect only the preceding term. If we want to apply the operator to a group of terms, we can use parentheses for groups, just as in a mathematical expression. For example, /(ab)+/ matches one or more consecutive occurrences of the substring ab.

When a part of a regex is grouped with parentheses, it serves double duty, also creating what's known as a *capture*. There's a lot to captures, and we discuss them in more depth in section 10.4.

ALTERNATION (OR)

Alternatives can be expressed using the pipe (|) character. For example: /a|b/ matches either the a or b character, and /(ab)+|(cd)+/ matches one or more occurrences of either ab or cd.

BACKREFERENCES

The most complex terms we can express in regular expressions are backreferences to *captures* defined in the regex. We address captures at length in section 10.4, but for now just think of them as the portions of a candidate string that are successfully matched against terms in the regular expression. The notation for such a term is the backslash followed by the number of the capture to be referenced, beginning with 1, such as \1, \2, and so on.

An example is /^([dtn])a\1/, which matches a string that starts with any of the d, t, or n characters, followed by an a, followed by whatever character matches the first capture. This latter point is important! This isn't the same as /[dtn] a[dtn]/. The character following the a can't be any of d, or t, or n, but must be whichever one of those triggers the match for the first character. As such, which character the \1 will match can't be known until evaluation time.

A good example of where this might be useful is in matching XML-type markup elements. Consider the following regex:

```
/<(\w+)>(.+)<\/\1>/
```

This allows us to match simple elements such as whatever. Without the ability to specify a backreference, this wouldn't be possible, because we'd have no way to know what closing tag would match the opening tag ahead of time.

TIP   That was kind of a whirlwind crash course on regular expressions. If they're still making you pull your hair out and you find yourself bogged down in the material that follows, we strongly recommend using one of the resources mentioned earlier in this chapter.

Now that you have a handle on regular expressions, you're ready to look at how to use them wisely in your code.

10.3   Compiling regular expressions

Regular expressions go through multiple phases of processing, and understanding what happens during each phase can help us optimize JavaScript code that uses regular expressions. The two main phases are compilation and execution.

Compilation occurs when the regular expression is first created. *Execution* occurs when we use the compiled regular expression to match patterns in a string.

During compilation, the expression is parsed by the JavaScript engine and converted into its internal representation (whatever that may be). This phase of parsing and conversion must occur every time a regular expression is created (discounting any internal optimizations performed by the browser).

Frequently, browsers *are* smart enough to determine when identical regular expressions are being used, and to cache the compilation results for that particular expression. But we can't count on this being the case in all browsers. For complex expressions, in particular, we can begin to get some noticeable speed improvements by predefining (and thus precompiling) our regular expressions for later use.

As we learned in our regular expression overview in the previous section, there are two ways of creating a compiled regular expression in JavaScript: via a literal and via a constructor. Let's look at an example in the following listing.

Listing 10.2   Two ways to create a compiled regular expression

```
const re1 = /test/i;                    ⟵——— Creates a regex via a literal
const re2 = new RegExp("test", "i");      ⟵
assert(re1.toString()=== "/test/i",             Creates a regex via the constructor
        "Verify the contents of the expression.");
assert(re1.test("TesT"), "Yes, it's case-insensitive.");
assert(re2.test("TesT"), "This one is too.");
assert(re1.toString()=== re2.toString(),
        "The regular expressions are equal.");
assert(re1 !== re2, "But they are different objects.");
```

In this example, both regular expressions are in their compiled state after creation. If we were to replace every reference to re1 with the literal /test/i, it's possible that the same regex would be compiled time and time again, so compiling a regex *once* and storing it in a variable for later reference can be an important optimization.

Note that each regex has a unique object representation: Every time a regular expression is created (and thus compiled), a new regular expression object is created.

This is unlike other primitive types (such as string, number, and so on), because the result will always be unique.

Of particular importance is the use of the constructor (`new RegExp(...)`) to create a regular expression. This technique allows us to build and compile an expression from a string that we can dynamically create at runtime. This can be immensely useful for constructing complex expressions that will be heavily reused.

For example, let's say that we want to determine which elements within a document have a particular class name, whose value we won't know until runtime. Because elements are capable of having multiple class names associated with them (inconveniently stored in a space-delimited string), this serves as an interesting example of runtime, regular-expression compilation (see the following listing).

Listing 10.3 Compiling a runtime regular expression for later use

```
<div class="samurai ninja"></div>              Creates test subjects of
<div class="ninja samurai"></div>              various elements with
<div></div>                                     various class names
<span class="samurai ninja ronin"></span>
<script>
  function findClassInElements(className, type) {
    const elems =                                    Collects elements by type
      document.getElementsByTagName(type || "*");
    const regex =                                    Compiles a regex using
      new RegExp("(^|\\s)" + className + "(\\s|$)");  the passed class name
    const results = [];        ← Stores the results
    for (let i = 0, length = elems.length; i < length; i++)
      if (regex.test(elems[i].className)) {          Tests for regex matches
        results.push(elems[i]);
      }
    return results;
  }
  assert(findClassInElements("ninja", "div").length === 2,
         "The right amount of div ninjas was found.");
  assert(findClassInElements("ninja", "span").length === 1,
         "The right amount of span ninjas was found.");
  assert(findClassInElements("ninja").length === 3,
         "The right amount of ninjas was found.");
</script>
```

We can learn several interesting things from listing 10.3. To start, we set up a number of test-subject `<div>` and `<span>` elements with various combinations of class names. Then we define our class-name checking function, which accepts as parameters the class name for which we'll check and the element type to check within.

Then we collect all the elements of the specified type by using the `getElementsBy-TagName` built-in method and set up our regular expression:

```
const regex = new RegExp("(^|\\s)" + className + "(\\s|$)");
```

Note the use of the `new RegExp()` constructor to compile a regular expression based on the class name passed to the function. This is an instance where we can't use a regex literal, as the class name for which we'll search isn't known in advance.

We construct (and hence, compile) this expression once in order to avoid frequent and unnecessary recompilation. Because the contents of the expression are dynamic (based on the incoming `className` argument), we can realize major performance savings by handling the expression in this manner.

The regex itself matches either the beginning of the string or a whitespace character, followed by the target class name, followed by either a whitespace character or the end of the string. Notice the use of a double-escape (\\) within the new regex: \\s. When creating literal regular expressions with terms including the backslash, we have to provide the backslash only once. But because we're writing these backslashes within a string, we must double-escape them. This is a nuisance, to be sure, but one that we must be aware of when constructing regular expressions in strings rather than literals.

After the regex is compiled, using it to collect the matching elements is a snap via the `test` method:

```
regex.test(elems[i].className)
```

Preconstructing and precompiling regular expressions so that they can be reused (executed) time and time again is a recommended technique that provides performance gains that can't be ignored. Virtually all complex regular expression situations can benefit from the use of this technique.

Earlier in this chapter, we mentioned that the use of parentheses in regular expressions not only serves to group terms for operator application, but also creates *captures*. Let's find out more about that.

10.4 Capturing matching segments

The height of usefulness with respect to regular expressions is realized when we *capture* the results that are found so that we can do something with them. Determining whether a string matches a pattern is an obvious first step and often all that we need, but determining *what* was matched is also useful in many situations.

10.4.1 Performing simple captures

Say we want to extract a value that's embedded in a complex string. A good example of such a string is the value of the CSS transform property, through which we can modify the visual position of an HTML element.

Listing 10.4   A simple function for capturing an embedded value

```
<div id="square" style="transform:translateY(15px);"></div>     ◁──┐  Defines the
<script>                                                              test subject
  function getTranslateY(elem){
    const transformValue = elem.style.transform;
```

```
    if(transformValue){
      const match = transformValue.match(/translateY\(([^\)]+)\)/);
      return match ? match[1] : "";
    }

    return "";
  }
```

> Extracts the translateY
> value from the string

```
  const square = document.getElementById("square");

  assert(getTranslateY(square) === "15px",
         "We've extracted the translateY value");
</script>
```

We define an element that specifies the style that will translate its position by 15 px:

```
"transform:translateY(15px);"
```

Unfortunately, the browser doesn't offer an API for easily fetching the amount by which the element is translated. So we create our own function:

```
function getTranslateY(elem){
    const transformValue = elem.style.transform;
    if(transformValue){
      const match = transformValue.match(/translateY\(([^\)]+)\)/);
      return match ? match[1] : "";
    }
     return "";
    }
```

The transform parsing code may seem confusing at first:

```
const match = transformValue.match(/translateY\(([^\)]+)\)/);
return match ? match[1] : "";
```

But it's not too bad when we break it down. To start, we need to determine whether a transform property even exists for us to parse. If not, we'll return an empty string. If the transform property is resident, we can get down to the opacity value extraction. The match method of a regular expression returns an array of captured values if a match is found, or null if no match is found.

The array returned by match includes the entire match in the first index, and then each subsequent capture following. So the zeroth entry would be the entire matched string of translateY(15px), and the entry at the next position would be 15px.

Remember that the captures are defined by parentheses in the regular expression. Thus, when we match the transform value, the value is contained in the [1] position of the array, because the only capture we specified in our regex was created by the parentheses that we embedded after the translateY portion of the regex.

This example uses a local regular expression and the match method. Things change when we use global expressions. Let's see how.

10.4.2 *Matching using global expressions*

As we saw in the previous section, using a local regular expression (one without the global flag) with the `String` object's `match` methods returns an array containing the entire matched string, along with any matched captures in the operation.

But when we supply a global regular expression (one with the g flag included), `match` returns something different. It's still an array of results, but in the case of a global regular expression, which matches all possibilities in the candidate string rather than just the first match, the array returned contains the global matches; captures *within* each match aren't returned in this case.

We can see this in action in the following code and tests.

Listing 10.5 Differences between global and local searches with `match`

```
const html = "<div class='test'><b>Hello</b> <i>world!</i></div>";
const results = html.match(/<(\/?)(\w+)([^>]*?)>/);
assert(results[0] === "<div class='test'>", "The entire match.");
assert(results[1] === "", "The (missing) slash.");
assert(results[2] === "div", "The tag name.");
assert(results[3] === " class='test'", "The attributes.");

const all = html.match(/<(\/?)(\w+)([^>]*?)>/g);
assert(all[0] === "<div class='test'>", "Opening div tag.");
assert(all[1] === "<b>", "Opening b tag.");
assert(all[2] === "</b>", "Closing b tag.");
assert(all[3] === "<i>", "Opening i tag.");
assert(all[4] === "</i>", "Closing i tag.");
assert(all[5] === "</div>", "Closing div tag.");
```

Matches using a local regex

Matches using a global regex

We can see that when we do a local match, `html.match(/<(\/?)(\w+)([^>]*?)>/)`, a single instance is matched and the captures within that match are also returned. But when we use a global match, `html.match(/<(\/?)(\w+)([^>]*?)>/g)`, what's returned is the list of matches.

If captures are important to us, we can regain this functionality while still performing a global search by using the regular expression's exec method. This method can be repeatedly called against a regular expression, causing it to return the next matched set of information every time it's called. A typical pattern for use is shown in the following listing.

Listing 10.6 Using the `exec` method to do both capturing and a global search

```
const html = "<div class='test'><b>Hello</b> <i>world!</i></div>";
const tag = /<(\/?)(\w+)([^>]*?)>/g;
let match, num = 0;
while ((match = tag.exec(html)) !== null) {
  assert(match.length === 4,
        "Every match finds each tag and 3 captures.");
  num++;
}
assert(num === 6, "3 opening and 3 closing tags found.");
```

Repeatedly calls exec

In this example, we repeatedly call the exec method:

```
while ((match = tag.exec(html)) !== null) {...}
```

This retains state from the previous invocation so that each subsequent call progresses to the next global match. Each call returns the next match *and* its captures.

By using either match or exec, we can always find the exact matches (and captures) that we're looking for. But we'll need to dig further if we want to refer to the captures themselves within the regex.

10.4.3 Referencing captures

We can refer to portions of a match that we've captured in two ways: one within the match itself, and one within a replacement string (where applicable). For example, let's revisit the match in listing 10.6 (in which we match an opening or closing HTML tag) and modify it in the following listing to also match the inner contents of the tag itself.

Listing 10.7   Using backreferences to match the contents of an HTML tag

Uses capture backreference

```
const html = "<b class='hello'>Hello</b> <i>world!</i>";
const pattern = /<(\w+)([^>]*)>(.*?)<\/\1>/g;                    ◁——  Runs the pattern
let match = pattern.exec(html);                          ◁——        on the test string
assert(match[0] === "<b class='hello'>Hello</b>",        ◁——
      "The entire tag, start to finish.");                       Tests various
assert(match[1] === "b", "The tag name.");                       captures that are
assert(match[2] === " class='hello'", "The tag attributes.");    captured by the
assert(match[3] === "Hello", "The contents of the tag.");        defined pattern

match = pattern.exec(html);
assert(match[0] === "<i>world!</i>",
      "The entire tag, start to finish.");
assert(match[1] === "i", "The tag name.");
assert(match[2] === "", "The tag attributes.");
assert(match[3] === "world!", "The contents of the tag.");
```

We use \1 to refer to the first capture within the expression, which in this case is the name of the tag. Using this information, we can match the appropriate closing tag, referring to whatever the capture matched. (This all assumes, of course, that there aren't any embedded tags of the same name within the current tag, so this is hardly an exhaustive example of tag matching.)

Additionally, we can get capture references within the replace string of a call to the replace method. Instead of using the backreference codes, as in listing 10.7, we use the syntax of $1, $2, $3, up through each capture number. Here's an example:

```
assert("fontFamily".replace(/([A-Z])/g, "-$1").toLowerCase() ===
      "font-family", "Convert the camelCase into dashed notation.");
```

In this code, the value of the first capture (in this case, the capital letter F) is refer-enced in the *replace string* (via $1). This allows us to specify a replace string without even knowing what its value will be until matching time. That's a powerful ninja-esque weapon to wield.

The ability to reference regular-expression captures helps make a lot of code that would otherwise be difficult, quite easy. The expressive nature that it provides ends up allowing for some terse statements that could otherwise be rather obtuse, convoluted, and lengthy.

Because both captures and expression grouping are specified using parentheses, there's no way for the regular-expression processor to know which sets of parentheses we added to the regex for grouping and which were intended to indicate captures. It treats all sets of parentheses as both groups and captures, which can result in the cap-ture of more information than we really intended, because we needed to specify some grouping in the regex. What can we do in such cases?

10.4.4 Noncapturing groups

As we noted, parentheses serve a double duty: They not only group terms for opera-tions, but also specify captures. This usually isn't an issue, but in regular expressions in which lots of grouping is going on, it could cause lots of needless capturing, which may make sorting through the resulting captures tedious.

Consider the following regex:

```
const pattern = /((ninja-)+)sword/;
```

Here, the intent is to create a regex that allows the prefix ninja- to appear one or more times before the word sword, and we want to capture the entire prefix. This regex requires two sets of parentheses:

- The parentheses that define the capture (everything before the string sword)
- The parentheses that group the text ninja- for the + operator

This all works fine, but it results in more than the single intended capture because of the inner set of grouping parentheses.

To indicate that a set of parentheses shouldn't result in a capture, the regular expression syntax lets us put the notation ?: immediately after the opening parenthe-sis. This is known as a *passive subexpression.*

Changing this regular expression to

```
const pattern = /((?:ninja-)+)sword/;
```

causes only the outer set of parentheses to create a capture. The inner parentheses have been converted to a passive subexpression.

To test this, take a look at the following code.

Listing 10.8   Grouping without capturing

```
const pattern = /((?:ninja-)+)sword/;                          Uses a passive
const ninjas = "ninja-ninja-sword".match(pattern);             subexpression

assert(ninjas.length === 2,"Only one capture was returned.");
assert(ninjas[1] === "ninja-ninja-",
      "Matched both words, without any extra capture.");
```

Running these tests, we can see that the passive subexpression `/((?:ninja-)+)sword/` prevents unnecessary captures.

Wherever possible in our regular expressions, we should strive to use noncapturing (passive) groups in place of capturing when the capture is unnecessary, so that the expression engine will have much less work to do in remembering and returning the captures. If we don't need captured results, there's no need to ask for them! The price that we pay is that already-complex regular expressions can become a tad more cryptic.

Now let's turn our attention to another way that regular expressions give us ninja powers: using functions with the `String` object's `replace` method.

10.5 *Replacing using functions*

The `replace` method of the `String` object is a powerful and versatile method, which we saw used briefly in our discussion of captures. When a regular expression is provided as the first parameter to `replace`, it will cause a replacement on a match (or *matches* if the regex is global) to the pattern rather than on a fixed string.

For example, let's say that we want to replace all uppercase characters in a string with X. We could write the following:

```
"ABCDEfg".replace(/[A-Z]/g,"X")
```

This results in a value of XXXXXfg. Nice.

But perhaps the most powerful feature presented by `replace` is the ability to provide a function as the replacement value rather than a fixed string.

When the replacement value (the second argument) is a function, it's invoked for each match found (remember that a global search will match all instances of the pattern in the source string) with a variable list of parameters:

- The full text of the match
- The captures of the match, one parameter for each
- The index of the match within the original string
- The source string

The value returned from the function serves as the replacement value.

This provides a tremendous amount of leeway to determine what the replacement string should be at runtime, with lots of information regarding the nature of the match at our fingertips. For example, in the following listing, we use the function to

provide a dynamic replacement value for converting a string with words separated by dashes to its camel-cased equivalent.

Listing 10.9   Converting a dashed string to camel case

```
function upper(all,letter) { return letter.toUpperCase(); }        ◁── Converts to
assert("border-bottom-width".replace(/-(\w)/g,upper)            ◁──   uppercase
       === "borderBottomWidth",
    "Camel cased a hyphenated string.");                         Matches dashed
                                                                 characters
```

Here, we provide a regex that matches any character preceded by a dash character. A capture in the global regex identifies the character that was matched (without the dash). Each time the function is called (twice in this example), it's passed the full match string as the first argument, and the capture (only one for this regex) as the second argument. We aren't interested in the rest of the arguments, so we didn't specify them.

The first time the function is called, it's passed `-b` and `b`; and the second time it's called, it's passed `-w` and `w`. In each case, the captured letter is uppercased and returned as the replacement string. We end up with `-b` replaced by `B` and with `-w` replaced by `W`.

Because a global regex will cause such a replace function to be executed for every match in a source string, this technique can even be extended beyond doing rote replacements. We can use the technique as a means of string traversal, instead of doing the `exec()`-in-a-while-loop technique that we saw earlier in this chapter.

For example, let's say that we're looking to take a query string and convert it to an alternative format that suits our purposes. We'd turn a query string such as

```
foo=1&foo=2&blah=a&blah=b&foo=3
```

into one that looks like this

```
foo=1,2,3&blah=a,b"
```

A solution using regular expressions and `replace` could result in some especially terse code, as shown in the next listing.

Listing 10.10   A technique for compressing a query string

```
function compress(source) {
  const keys = {};                              ◁──── Stores located keys
  source.replace(
    /([^=&]+)=([^&]*)/g,
    function(full, key, value) {                ◁──── Extracts key-value info
      keys[key] =
        (keys[key] ? keys[key] + "," : "") + value;
      return "";
    }
```

```
  );
  const result = [];
  for (let key in keys) {                          Collects
    result.push(key + "=" + keys[key]);            key info
  }
  return result.join("&");            ◁────────
}                                             Joins results with &

assert(compress("foo=1&foo=2&blah=a&blah=b&foo=3") ===
       "foo=1,2,3&blah=a,b",
       "Compression is OK!");
```

The most interesting aspect of this example is its use of the string `replace` method as a means of traversing a string for values, rather than as a search-and-replace mechanism. The trick is twofold: passing in a function as the replacement value argument, and instead of returning a value, using it as a means of searching.

The example code first declares a hash `key` in which we store the keys and values that we find in the source query string. Then we call the `replace` method on the source string, passing a regex that will match the key-value pairs, and capture the key and the value. We also pass a function that will be passed the full match, the key capture, and the value capture. These captured values get stored in the hash for later reference. Note that we return the empty string because we don't care what substitutions happen to the source string—we're just using the side effects rather than the result.

After `replace` returns, we declare an array in which we'll aggregate the results and iterate through the keys that we found, adding each to the array. Finally, we join each of the results we stored in the array by using `&` as the delimiter, and we return the result:

```
const result = [];
for (let key in keys) {
  result.push(key + "=" + keys[key]);
}
return result.join("&");
```

Using this technique, we can co-opt the `String` object's `replace` method as our own string-searching mechanism. The result isn't only fast, but also simple and effective. The level of power that this technique provides, especially in light of the small amount of code necessary, shouldn't be underestimated.

All of these regular expression techniques can have a huge impact on how we write script on our pages. Let's see how to apply what you've learned to solve some common problems we might encounter.

10.6 *Solving common problems with regular expressions*

In JavaScript, a few idioms tend to occur again and again, but their solutions aren't always obvious. A knowledge of regular expressions can definitely come to the rescue, and in this section we'll look at a few common problems that we can solve with a regex or two.

10.6.1 *Matching newlines*

When performing a search, it's sometimes desirable for the period (.) term, which matches any character except for newline, to also include newline characters. Regular expression implementations in other languages frequently include a flag for making this possible, but JavaScript's implementation doesn't.

Let's look at a couple of ways of getting around this omission in JavaScript, as shown in the next listing.

Listing 10.11   Matching *all* characters, including newlines

> *Defines a test subject*
> *Shows that newlines aren't matched*

```
const html = "<b>Hello</b>\n<i>world!</i>";
assert(/.*/.exec(html)[0] === "<b>Hello</b>",
       "A normal capture doesn't handle endlines.");
assert(/[\S\s]*/.exec(html)[0] ===
       "<b>Hello</b>\n<i>world!</i>",
       "Matching everything with a character set.");
assert(/(?:.|\s)*/.exec(html)[0] ===
       "<b>Hello</b>\n<i>world!</i>",
       "Using a non-capturing group to match everything.");
```

> *Matches all using whitespace matching*
> *Matches all using alteration*

This example defines a test subject string: `"<b>Hello</b>\n<i>world!</i>"`, containing a newline. Then we try various ways of matching all the characters in the string.

In the first test, `/.*/.exec(html)[0] === "<b>Hello</b>"`, we verify that newlines aren't matched by the `.` operator. Ninjas won't be denied, so in the next test we get our way with an alternative regex, `/[\S\s]*/`, in which we define a character class that matches anything that's *not* a whitespace character and anything that *is* a whitespace character. This union is the set of all characters.

Another approach is taken in the next test:

```
/[\S\s]*/.exec(html)[0] === "<b>Hello</b>\n<i>world!</i>"
```

Here we use an alternation regex, `/(?:.|\s)*/`, in which we match everything matched by `.`, which is everything but newline, and everything considered whitespace, which includes newline. The resulting union is the set of all characters including newlines. Note the use of a passive subexpression to prevent any unintended captures. Because of its simplicity (and implicit speed benefits), the solution provided by `/[\S\s]*/` is generally considered optimal.

Next, let's take a step to widen our view to a worldwide scope.

10.6.2 *Matching Unicode*

Frequently in the use of regular expressions, we want to match alphanumeric characters, such as an ID selector in a CSS selector engine implementation. But assuming that the alphabetic characters will be from only the set of English ASCII characters is shortsighted.

Expanding the set to include Unicode characters is sometimes desirable, explicitly supporting multiple languages not covered by the traditional alphanumeric character set (see the following listing).

> **Listing 10.12   Matching Unicode characters**

```
const text ="\u5FCD\u8005\u30D1\u30EF\u30FC";
const matchAll = /[\w\u0080-\uFFFF_-]+/;           ◁────── Matches all,
assert(text.match(matchAll),"Our regexp matches non-ASCII!");    including Unicode
```

This listing includes the entire range of Unicode characters in the match by creating a character class that includes the \w term, to match all the "normal" word characters, plus a range that spans the entire set of Unicode characters above U+0080. Starting at 128 gives us some high ASCII characters along with all Unicode characters in the Basic Multilingual Plane.

The astute among you might note that by adding the entire range of Unicode characters above \u0080, we match not only alphabetic characters, but also all Unicode punctuation and other special characters (arrows, for example). But that's okay, because the point of the example is to show how to match Unicode characters in general. If you have a specific range of characters that you want to match, you can use the lesson of this example to add whatever range you wish to the character class.

Before moving on from our examination of regular expressions, let's tackle one more common issue.

10.6.3   *Matching escaped characters*

It's common for page authors to use names that conform to program identifiers when assigning id values to page elements, but that's just a convention; id values can contain characters other than "word" characters, including punctuation. For example, a web developer might use the id value form:update for an element.

A library developer, when writing an implementation for, say, a CSS selector engine, would like to support escaped characters. This allows the user to specify complex names that don't conform to typical naming conventions. So let's develop a regex that will allow matching escaped characters. Consider the following code.

> **Listing 10.13   Matching escaped characters in a CSS selector**

```
const pattern = /^((\w+)|(\\.))+$/;    ◁──── This regular expression allows
const tests = [                              any sequence composed of word
  "formUpdate",                              characters, a backslash followed
  "form\\.update\\.whatever",                by any character (even a
  "form\\:update",                           backslash), or both.
  "\\f\\o\\r\\m\\u\\p\\d\\a\\t\\e",
  "form:update"
];
```

Sets up various test subjects. All should pass but the last, which fails to escape its nonword character (:).

```
for (let n = 0; n < tests.length; n++) {        ◁──┐  Runs through all
  assert(pattern.test(tests[n]),                   │  the test subjects
          tests[n] + " is a valid identifier" );   │
}
```

This particular expression works by allowing for a match of either a word character sequence or a sequence of a backslash followed by any character.

Note that more work is required to fully support all escape characters. For more details, visit https://mathiasbynens.be/notes/css-escapes.

10.7 *Summary*

- Regular expressions are a powerful tool that permeates modern JavaScript development; virtually every aspect of any sort of matching depends on their use. With a good understanding of the advanced regex concepts covered in this chapter, you should feel comfortable tackling any challenging piece of code that could benefit from regular expressions.
- We can create regular expressions with regular expression literals (/test/) and with the RegExp constructor (new RegExp("test")). Literals are preferred when the regex is known at development time, and the constructor when the regex is constructed at runtime.
- With each regular expression, we can associate five flags: i makes the regex case-insensitive, g matches all instances of the pattern, m allows matches across multiple lines, y enables sticky matching, while u enables the use of Unicode escapes. Flags are added at the end of a regex literal: /test/ig, or as a second parameter to the RegExp constructor: new RegExp("test", "i").
- Use [] (as in [abc]) to specify a set of characters that we wish to match.
- Use ^ to signify that the pattern must appear at the beginning of a string and $ to signify that the pattern must appear at the end of a string.
- Use ? to specify that a term is optional, + that a term should appear one or many times, and * to specify that a term appears zero, one, or many times.
- Use . to match any character.
- We can use backslash (\) to escape special regex characters (such as . [$ ^).
- Use parentheses () to group multiple terms together, and pipe (|) to specify alternation.
- Portions of a string that are successfully matched against terms can be back referenced with a backslash followed by the number of the capture (\1, \2, and so on).
- Every string has access to the match function, which takes in a regular expression and returns an array containing the entire matched string along with any matched captures. We can also use the replace function, which causes a replacement on pattern matches rather than on a fixed string.

10.8   Exercises

1 In JavaScript, regular expressions can be created with which of the following?

   a Regular expression literals

   b The built-in `RegExp` constructor

   c The built-in `RegularExpression` constructor

2 Which of the following is a regular expression literal?

   a `/test/`

   b `\test\`

   c `new RegExp("test");`

3 Choose the correct regular expression flags:

   a `/test/g`

   b `g/test/`

   c `new RegExp("test", "gi");`

4 The regular expression `/def/` matches which of the following strings?

   a One of the strings d, e, f

   b `def`

   c `de`

5 The regular expression `/[^abc]/` matches which of the following?

   a One of strings `a, b, c`

   b One of strings `d, e, f`

   c Matches the string `ab`

6 Which of the following regular expressions matches the string `hello`?

   a `/hello/`

   b `/hell?o/`

   c `/hel*o/`

   d `/[hello]/`

7 The regular expression `/(cd)+(de)*/` matches which of the following strings?

   a `cd`

   b `de`

   c `cdde`

   d `cdcd`

   e `ce`

   f `cdcddedede`

8 In regular expressions, we can express alternatives with which of the following?

　a `#`

　b `&`

　c `|`

9 In the regular expression `/([0-9])2/`, we can reference the first matched digit with which of the following?

　a `/0`

　b `/1`

　c `\0`

　d `\1`

10 The regular expression `/([0-5])6\1/` will match which of the following?

　a `060`

　b `16`

　c `261`

　d `565`

11 The regular expression `/(?:ninja)-(trick)?-\1/` will match which of the following?

　a `ninja-`

　b `ninja-trick-ninja`

　c `ninja-trick-trick`

12 What is the result of executing `"012675".replace(/0-5/g, "a")`?

　a `aaa67a`

　b `a12675`

　c `a1267a`

Code
modularization techniques

11

This chapter covers
- Using the module pattern
- Using current standards for writing modular code: AMD and CommonJS
- Working with ES6 modules

So far we've explored the basic primitives of JavaScript, such as functions, objects, collections, and regular expressions. We have more than a couple of tools in our belt for solving specific problems with our JavaScript code. But as our applications start to grow, another whole set of problems, related to how we structure and manage our code, starts to emerge. Time and time again, it's been proven that large, monolithic code bases are far more likely to be difficult to understand and maintain than smaller, well-organized ones. So it's only natural that one way of improving the structure and organization of our programs is to break them into smaller, relatively loosely coupled segments called *modules*.

Modules are larger units of organizing our code than objects and functions; they allow us to divide programs into clusters that belong together. When creating

modules, we should strive to form consistent abstractions and encapsulate implementation details. This makes it easier to reason about our application, because we aren't bothered with various frivolous details when using our module functionality. In addition, having modules means that we can easily reuse module functionality in different parts of our applications, and even across different applications, significantly speeding up our development process.

As you saw earlier in the book, JavaScript is big on global variables: Whenever we define a variable in mainline code, that variable is automatically considered global and can be accessed from any other part of our code. This might not be a problem for small programs, but as our applications start to grow and we include third-party code, the chance that naming clashes will occur starts to grow significantly. In most other programming languages, this problem is solved with namespaces (C++ and C#) or packages (Java), which wrap all enclosed names in another name, thereby significantly reducing potential clashes.

Up to ES6, JavaScript didn't offer a higher-level, built-in feature that allows us to group related variables in a module, namespace, or package. So in order to tackle this problem, JavaScript programmers have developed advanced code modularization techniques that take advantage of existing JavaScript constructs, such as objects, immediate functions, and closures. In this chapter, we'll explore some of these techniques.

Luckily, it's only a matter of time until we'll be able to completely let go of these work-around techniques, because ES6 finally introduces native modules. Unfortunately, the browsers haven't caught on, so we'll explore how modules should work in ES6, even though we won't have a specific native browser implementation to test them on.

Let's start with modularization techniques that we can use today.

. . . .

Do you know?

What existing mechanism do you use to approximate modules in JavaScript pre-ES6?

What is the difference between the AMD and CommonJS module specifications?

Using ES6, what two statements would you need to use the `tryThisOut()` function from a module called `test` from within another module called `guineaPig`?

. . . .

11.1 *Modularizing code in pre-ES6 JavaScript*

Pre-ES6 JavaScript has only two types of scopes: global scope and function scope. It doesn't have something in between, a namespace or a module that would allow us to group certain functionality together. To write modular code, JavaScript developers are forced to be creative with existing JavaScript language features.

When deciding which features to use, we have to keep in mind that, at a bare minimum, each module system should be able to do the following:

■ *Define an interface* through which we can access the functionality offered by the module.

■ *Hide module internals* so that the users of our modules aren't burdened with a whole host of unimportant implementation details. In addition, by hiding module internals, we protect those internals from the outside world, thereby preventing unwanted modifications that can lead to all sorts of side effects and bugs.

In this section, we'll first see how to create modules by using standard JavaScript features that we've explored so far in the book, features such as objects, closures, and immediate functions. We'll continue this modularization vein by exploring Asynchronous Module Definition (AMD) and CommonJS, the two most popular module specification standards, built on slightly different foundations. You'll learn how to define modules using these standards, as well as their pros and cons.

But let's start with something for which we've already set the stage in previous chapters.

11.1.1 Using objects, closures, and immediate functions to specify modules

Let's go back to our minimal module system requirements, *hiding implementation details* and *defining module interfaces.* Now think about which JavaScript language features we can take advantage of in order to implement these minimal requirements:

■ *Hiding module internals*—As we already know, calling a JavaScript function creates a new scope in which we can define variables that are visible only within the current function. So, one option for hiding module internals is using functions as modules. In this way, all function variables become internal module variables that are hidden from the outside world.

■ *Defining module interfaces*—Implementing module internals through function variables means that those variables are accessible from only within the module. But if our modules are to be used from other code, we have to be able to define a clean interface through which we can expose the functionality offered by the module. One way of achieving this is by taking advantage of objects and closures. The idea is that, from our function module, we return an object that represents the public interface of our module. That object should contain methods offered by the module, methods that will, through closures, keep alive our internal module variables, even after our module function has finished its execution.

Now that we've given a high-level description of how to implement modules in JavaScript, let's go through it slowly, step by step, starting with using functions for hiding module internals.

FUNCTIONS AS MODULES

Calling a function creates a new scope that we can use to define variables that won't be visible from outside the current function. Let's take a look at the following code snippet that counts the number of clicks on a web page:

```
(function countClicks(){                          Defines a local variable that
  let numClicks = 0;                              will store click counts
  document.addEventListener("click", () => {
    alert( ++numClicks );                         Whenever a user clicks, the counter
  });                                             is incremented and the current value
})();                                             reported.
```

In this example, we create a function called `countClicks` that creates a variable `num-Clicks` and registers a click event handler on the entire document. Whenever a click is made, the `numClicks` variable gets incremented and the result is displayed to the user via an alert box. There are two important things to notice here:

- The `numClicks` variable, internal to the `countClicks` function, is kept alive through the closure of the click handler function. The variable can be referenced only within the handler, and *nowhere else*! We've shielded the `numClicks` variable from the code outside the `countClicks` function. At the same time, we haven't polluted the global namespace of our program with a variable that's probably not that important for the rest of our code.
- Our `countClicks` function is called only in this one place, so instead of defining a function and then calling it in a separate statement, we've used an immediate function, or an IIFE (presented in chapter 3), to define and immediately invoke the `countClicks` function.

We can also take a look at the current application state, with respect to how our internal function (or module) variable is kept alive through closures, as shown in figure 11.1.

Now that we understand how to hide internal module details, and how closures can keep those internal details alive as long as necessary, let's move on to our second minimal requirement for modules: defining module interfaces.

THE MODULE PATTERN: AUGMENTING FUNCTIONS AS MODULES WITH OBJECTS AS INTERFACES

The module interface is usually composed of a set of variables and functions that our module provides to the outside world. The easiest way to create such an interface is to use the humble JavaScript object.
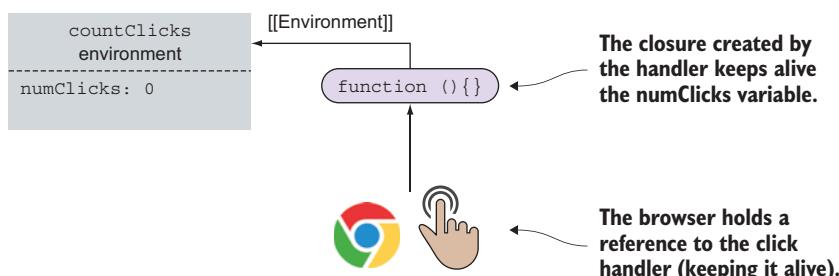
Figure 11.1 The click event handler, through closures, keeps alive the local `numClicks` variable.

For example, let's create an interface for our module that counts the clicks on our web page, as shown in the following listing.

Listing 11.1   The module pattern

Creates a "private" module variable

Creates a global module variable and assigns the result of immediately invoking a function

Creates a "private" module function

Returns an object that represents the module's interface. Through closures, we can access "private" module variables and functions.

```
const MouseCounterModule = function() {
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };

  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
}();
```

From outside, we can access the properties exposed through the interface.

```
assert(typeof MouseCounterModule.countClicks === "function",
       "We can access module functionality");
assert(typeof MouseCounterModule.numClicks === "undefined"
    && typeof MouseCounterModule.handleClick === "undefined" ,
       "We cannot access internal module details");
```

But we can't access module internals.

Here we use an immediate function to implement a module. Within the immediate function, we define our internal module implementation details: one local variable, `numClicks`, and one local function, `handleClick`, that are accessible only within the module. Next we create and immediately return an object that will serve as the module's "public interface." This interface contains a `countClicks` method that we can use from outside the module to access module functionality.

At the same time, because we've exposed a module interface, our internal module details are kept alive through closures created by the interface. For example, in this case, the `countClicks` method of the interface keeps alive internal module variables `numClicks` and `handleClick`, as shown in figure 11.2.

Finally, we store the object that represents the module interface, returned by the immediate function, into a variable named `MouseCounterModule`, through which we can easily consume module functionality, by writing the following code:

```
MouseCounterModule.countClicks()
```

And that's basically it.

By taking advantage of immediate functions, we can hide certain module implementation details. Then by adding objects and closures into the mix, we can specify a module interface that exposes the functionality provided by our module to the outside world.

```
const MouseCounterModule = function(){
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };

  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
}();
```
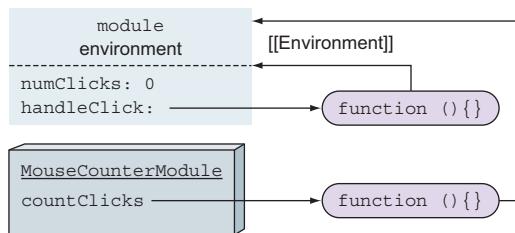
Figure 11.2 Expose the public interface of a module through a returned object. The internal module implementation ("private" variables and functions) is kept alive through closures created by public interface methods.

This pattern of using immediate functions, objects, and closures to create modules in JavaScript is called the *module pattern.* It was popularized by Douglas Crockford, and was one of the first massively popular ways of modularizing JavaScript code.

Once we have the ability to define modules, it's always nice to be able to split them across multiple files (in order to more easily manage them), or to be able to define additional functionality on existing modules, without modifying their source code.

Let's see how this can be done.

AUGMENTING MODULES

Let's augment our `MouseCounterModule` from the previous example with an additional feature of counting the number of mouse scrolls, but without modifying the original `MouseCounterModule` code. See the following listing.

Listing 11.2 Augmenting modules

```
const MouseCounterModule = function(){          ⟵  The original
  let numClicks = 0;                                MouseCounterModule
  const handleClick = () => {
    alert(++numClicks);
  };

  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
}();
                                                 Immediately invokes a function
                                                 that accepts the module we
  (function(module) {                            want to extend as an argument
```

```
        let numScrolls = 0;                          Defines new
        const handleScroll = () => {                 private variables
          alert(++numScrolls);                       and functions
        }

        module.countScrolls = () => {                           Extends the
          document.addEventListener("wheel", handleScroll);     module interface
        };
      })(MouseCounterModule);

    assert(typeof MouseCounterModule.countClicks === "function",
           "We can access initial module functionality");

    assert(typeof MouseCounterModule.countScrolls === "function",
           "We can access augmented module functionality");
```

Passes in the module as an argument (pointing to `})(MouseCounterModule);`)

When augmenting a module, we usually follow a procedure similar to creating a new module. We immediately call a function, but this time, we pass to it the module we want to extend as an argument:

```
(function(module){
  ...
  return module;
})(MouseCounterModule);
```

Within the function, we then go about our work and create all the necessary private variables and functions. In this case, we've defined a private variable and a private function for counting and reporting the number of scrolls:

```
let numScrolls = 0;
const handleScroll = () => {
  alert(++numScrolls);
}
```

Finally, we extend our module, available through the immediate function's `module` parameter, just as we would extend any other object:

```
module.countScrolls = ()=> {
  document.addEventListener("wheel", handleScroll);
};
```

After we've performed this simple operation, our `MouseCounterModule` can also `countScrolls`.

Our public module interface now has two methods, and we can use the module in the following ways:

```
MouseCounterModule.countClicks();        A method that's part of the module's
MouseCounterModule.countScrolls();       interface from the beginning

                                         A new module method that we've
                                         added by extending the module
```

As we've already mentioned, we've extended the module in a way that's similar to the creation of a new module, through an immediately invoked function that extends the module. This has some interesting side effects in terms of closures, so let's take a closer look at the application state after we've augmented the module, as shown in figure 11.3.

If you look closely, figure 11.3 also shows one of the shortcomings of the module pattern: the inability to share private module variables across module extensions. For example, the `countClicks` function keeps a closure around the `numClicks` and `handleClick` variables, and we could access these private module internals through the `countClicks` method.

```javascript
const MouseCounterModule = function() {
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };
  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
}();

(function (module){
  let numScrolls = 0;
  const handleScroll = () => {
    alert(++numScrolls);
  }

  module.countScrolls = () => {
    document.addEventListener("wheel", handleClick);
  };
})(MouseCounterModule);
```
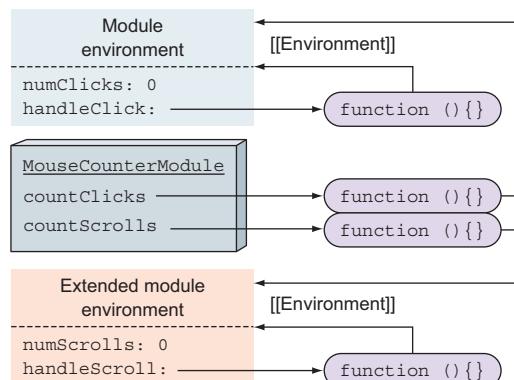

Figure 11.3 When augmenting a module, we extend its external interface with new functionality, usually by passing the module to another immediate function. In this example, we add the ability to `countScrolls` to our `MouseCounterModule`. Notice that two separate functions are defined in different environments, and they can't access each other's internal variables.

Unfortunately, our extension, the `countScrolls` function, is created in a completely separate scope, with a completely new set of private variables: `numScrolls` and `handleScroll`. The `countScrolls` function creates a closure only around `numScrolls` and `handleScroll` variables, and therefore can't access the `numClicks` and `handle-Click` variables.

> **NOTE** Module extensions, when performed through separate immediate functions, can't share private module internals, because every function invocation creates a new scope. Although this is a shortcoming, it's not a show-stopper, and we can still use the module pattern to keep our JavaScript applications modular.

Note that, in the module pattern, modules are objects just like any other, and we can extend them in any way we find appropriate. For example, we can add new functionality by extending the module object with new properties:

```
MouseCounterModule.newMethod = ()=> {...}
```

We can also use the same principle to easily create submodules:

```
MouseCounterModule.newSubmodule = () => {
  return {...};
}();
```

Notice that all of these approaches suffer from the same fundamental shortcoming of the module pattern: Subsequent extensions of the module can't access previously defined internal module details.

Unfortunately, there are more problems with the module pattern. When we start building modular applications, the modules themselves will often depend on other modules for their functionality. Unfortunately, the module pattern doesn't cover the management of these dependencies. We, as developers, have to take care of the right dependency order so that our module code has all it needs to execute. Although this isn't a problem in small and medium applications, it can introduce serious issues in large applications that use a lot of interdependent modules.

To deal with these issues, a couple of competing standards have arisen, namely Asynchronous Module Definition (AMD) and CommonJS.

11.1.2 *Modularizing JavaScript applications with AMD and CommonJS*

AMD and CommonJS are competing module specification standards that allow us to specify JavaScript modules. Besides some differences in syntax and philosophy, the main difference is that AMD was designed explicitly with the browser in mind, whereas CommonJS was designed for a general-purpose JavaScript environment (such as servers, with Node.js), without being bound to the limitations of the browser. This section provides a relatively short overview of these module specifications; setting them up and including

them in your projects is beyond the scope of this book. For more information, we recommend *JavaScript Application Design* by Nicolas G. Bevacqua (Manning, 2015).

AMD

AMD grew out of the Dojo toolkit (https://dojotoolkit.org/), one of the popular JavaScript toolkits for building client-side web applications. AMD allows us to easily specify modules and their dependencies. At the same time, it was built from the ground up for the browser. Currently, the most popular AMD implementation is RequireJS (http://requirejs.org/).

Let's see an example of defining a small module that has a dependency to jQuery.

Listing 11.3   Using AMD to specify a module dependent on jQuery

```
define('MouseCounterModule',['jQuery'], $ => {      ⊲─┐ Uses the define function to specify
  let numClicks = 0;                                   a module, its dependencies, and
  const handleClick = () => {                          the module factory function that
    alert(++numClicks);                                will create the module
  };

  return {                               ⊲────── The public interface of our module
    countClicks: () => {
      $(document).on("click", handleClick);
    }
  };
});
```

AMD provides a function called `define` that accepts the following:

- The ID of the newly created module. This ID can later be used to require the module from other parts of our system.
- A list of module IDs on which our current module depends (the required modules).
- A factory function that will initialize the module and that accepts the required modules as arguments.

In this example, we use AMD's `define` function to create a module with an ID `MouseCounterModule` that depends on jQuery. Because of this dependency, AMD first requests the jQuery module, which can take some time if the file has to be requested from a server. This action is performed asynchronously, in order to avoid blocking. After all dependencies have been downloaded and evaluated, the module factory function is called with one argument for each requested module. In this case, there will be one argument, because our new module requires only jQuery. Within the factory function, we create our module just as we would with the standard module pattern: by returning an object that exposes the public interface of the module.

As you can see, AMD offers several interesting benefits, such as these:

- Automatic resolving of dependencies, so that we don't have to think about the order in which we include our modules.
- Modules can by asynchronously loaded, thereby avoiding blocking.
- Multiple modules can be defined within one file.

Now that you have the basic idea of how AMD works, let's take a look at another, massively popular module definition standard.

COMMONJS

Whereas AMD was built explicitly for the browser, CommonJS is a module specification designed for a general-purpose JavaScript environment. Currently it has the biggest following in the Node.js community.

CommonJS uses file-based modules, so we can specify one module per file. To each module, CommonJS exposes a variable, `module`, with a property, `exports`, which we can easily extend with additional properties. In the end, the content of `module.exports` is exposed as the module's public interface.

If we want to use a module from other parts of the application, we can require it. The file will be synchronously loaded, and we'll have access to its public interface. This is the reason that CommonJS is much more popular on the server, where module fetching is relatively quick because it requires only a file-system read, than on the client, where the module has to be downloaded from a remote server, and where synchronous loading usually means blocking.

Let's look at an example that defines our reoccurring `MouseCounterModule`, this time in CommonJS.

Listing 11.4   Using CommonJS to define a module

```
//MouseCounterModule.js
const $ = require("jQuery");          ⟵  Synchronously requires
let numClicks = 0;                        a jQuery module
const handleClick = () => {
  alert(++numClicks);
};

module.exports = {                    ⟵  Modifies the module.exports property to
  countClicks: () => {                    specify the public interface of a module
    $(document).on("click", handleClick);
  }
};
```

To include our module within a different file, we can write this:

```
const MouseCounterModule = require("MouseCounterModule.js");
MouseCounterModule.countClicks();
```

See how simple this is?

Because the philosophy of CommonJS dictates one module per file, any code that we put in a file module will be a part of that module. Therefore, there's no need for wrapping variables up in immediate functions. All variables defined within a module are safely contained within the scope of the current module and don't leak out to the global scope. For example, all three of our module variables (`$`, `numClicks`, and `handleClick`) are module scoped, even though they're defined in top-level code (outside all functions and blocks), which would technically make them global variables in standard JavaScript files.

Once again, it's important to note that only variables and functions exposed through the `module.exports` object are available from outside the module. The procedure is similar to the module pattern, only instead of returning a completely new object, the environment already provides one that we can extend with our interface methods and properties.

CommonJS has a couple of advantages:

- It has simple syntax. We need to specify only the `module.exports` properties, while the rest of the module code stays pretty much the same as if we were writing standard JavaScript. Requiring modules is also simple; we just use the `require` function.

- CommonJS is the default module format for Node.js, so we have access to thousands of packages that are available through npm, node's package manager.

CommonJS's biggest disadvantage is that it wasn't explicitly built with the browser in mind. Within JavaScript in the browser, there's no support for the `module` variable and the `export` property; we have to package our CommonJS modules into a browser-readable format. We can achieve this with either Browserify (http://browserify.org/) or RequireJS (http://requirejs.org/docs/commonjs.html).

Having two competing standards for specifying modules, AMD and CommonJS, has led to one of those situations in which people tend to divide themselves into two, sometimes even opposing, camps. If you work on relatively closed projects, this might not be an issue; you choose the standard that suits you better. Problems can arise, however, when we need to reuse code from the opposing camp and are forced to jump through all sorts of hoops. One solution is to use Universal Module Definition, or UMD (https://github.com/umdjs/umd), a pattern with a somewhat convoluted syntax that allows the same file to be used by both AMD and CommonJS. This is beyond the scope of this book, but if you're interested, many quality resources are available online.

Luckily, the ECMAScript committee has recognized the need for a unified module syntax supported in all JavaScript environments, so ES6 defines a new module standard that should finally put these differences to rest.

11.2 ES6 modules

ES6 modules are designed to marry the advantages of CommonJS and AMD:

- Similar to CommonJS, ES6 modules have a relatively simple syntax, and ES6 modules are file based (one module per file).
- Similar to AMD, ES6 modules provide support for asynchronous module loading.

> **NOTE**   Built-in modules are a part of the ES6 standard. As you'll soon see, the ES6 module syntax includes additional semantics and keywords (such as the `export` and `import` keywords) that aren't supported by current browsers. If we want to use modules today, we have to transpile our module code with Traceur (https://github.com/google/traceur-compiler), Babel (http://babeljs.io/), or TypeScript (www.typescriptlang.org/). We also can use the SystemJS library (https://github.com/systemjs/systemjs), which provides support for loading all currently available module standards: AMD, CommonJS, and even ES6 modules. You can find instructions on how to use SystemJS in the project's repository (https://github.com/systemjs/systemjs).

The main idea behind ES6 modules is that only the identifiers explicitly exported from a module are accessible from outside that module. All other identifiers, even the ones defined in top-level scope (what would be global scope in standard JavaScript), are accessible only from within the module. This was inspired by CommonJS.

To provide this functionality, ES6 introduces two new keywords:

- `export`—For making certain identifiers available from outside the module
- `import`—For importing exported module identifiers

The syntax for exporting and importing module functionality is simple, but it has a lot of subtle nuances that we'll explore slowly, step by step.

11.2.1 Exporting and importing functionality

Let's start with a simple example that shows how to export functionality from one module and import it into another.

Listing 11.5   Exporting from a Ninja.js module

```
const ninja = "Yoshi";                    ⟵——— Defines a top-level variable in a module
export const message = "Hello";

                                          Defines a variable and a function, and exports
                                          them from the module with the export keyword
export function sayHiToNinja() {
  return message + " " + ninja;           ⟵
}                                               Accesses an inner module variable
                                                from the module's public API
```

We first define a variable, `ninja`, a module variable that will be accessible only within this module, even though it's placed in top-level code (which would make it a global variable in pre-ES6 code).

Next, we define another top-level variable, `message`, which we make accessible from outside the module by using the new `export` keyword. Finally, we also create and export the `sayHiToNinja` function.

And that's it! This is the minimum syntax we need to know for defining our own modules. We don't have to use immediate functions or remember any esoteric syntax in order to export functionality from a module. We write our code as we would write standard JavaScript code, with the only difference that we prefix some of the identifiers (such as variables, functions, or classes) with an `export` keyword.

Before learning how to import this exported functionality, we'll take a look at an alternative way to export identifiers: We list everything we want to export at the end of the module, as shown in the following listing.

Listing 11.6   Exporting at the end of a module

```
const ninja = "Yoshi";                  Specifies all module identifiers
const message = "Hello";

function sayHiToNinja() {
  return message + " " + ninja;         Exports some of the
}                                        module identifiers

export { message, sayHiToNinja };
```

This way of exporting module identifiers bears some resemblance to the module pattern, as an immediate function returns an object that represents the public interface of our module, and especially to CommonJS, as we expand the `module.exports` object with the public module interface.

Regardless of how we've exported identifiers of a certain module, if we need to import them into another module, we have to use the `import` keyword, as in the following example.

Listing 11.7   Importing from the Ninja.js module

```
                                          Uses the import keyword to import
                                          an identifier binding from a module
import { message, sayHiToNinja} from "Ninja.js";  ◁┘

assert(message === "Hello",
      "We can access the imported variable");     We can now access the
assert(sayHiToNinja() === "Hello Yoshi",          imported variable and call
      "We can say hi to Yoshi from outside the module");  the imported function.

assert(typeof ninja === "undefined",              We can't access not-exported
      "But we cannot access Yoshi directly");     module variables directly.
```

We use the new `import` keyword to import a variable, `message` and a function, `sayHi-ToNinja` from the `ninja` module:

```
import { message, sayHiToNinja} from "Ninja.js";
```

By doing this, we've gained access to these two identifiers defined in the `ninja` module. Finally, we can test that we can access the `message` variable and call the `sayHiToNinja` function:

```
assert(message === "Hello",
      "We can access the imported variable");
assert(sayHiToNinja() === "Hello Yoshi",
      "We can say hi to Yoshi from outside the module");
```

What we can't do is access the nonexported and nonimported variables. For example, we can't access the `ninja` variable because it isn't marked with `export`:

```
assert(typeof ninja === "undefined",
      "But we cannot access Yoshi directly");
```

With modules, we're finally a bit safer from the misuse of global variables. Everything that we didn't explicitly mark for export stays nicely isolated within a module.

In this example, we've used a *named export*, which enables us to export multiple identifiers from a module (as we did with `message` and `sayHiToNinja`). Because we can export a large number of identifiers, listing them all in an import statement can be tedious. Therefore, a shorthand notation enables us to bring in all exported identifiers from a module, as shown in the following listing.

Listing 11.8   Importing all named exports from the Ninja.js module

```
import * as ninjaModule from "Ninja.js";          ⟵──┐  Uses * notation to import
                                                       all exported identifiers

assert(ninjaModule.message === "Hello",
      "We can access the imported variable");        Refers to the named
assert(ninjaModule.sayHiToNinja() === "Hello Yoshi", exports through
      "We can say hi to Yoshi from outside the module"); property notation

assert(typeof ninjaModule.ninja === "undefined",   We still can't access not-
      "But we cannot access Yoshi directly");      exported identifiers.
```

As listing 11.8 shows, to import all exported identifiers from a module, we use the `import` `*` notation in combination with an identifier that we'll use to refer to the whole module (in this case, the `ninjaModule` identifier). After we've done this, we can access the exported identifiers through property notation; for example, `ninjaModule.message`, `ninjaModule.sayHiToNinja`. Notice that we still can't access top-level variables that weren't exported, as is the case with the `ninja` variable.

DEFAULT EXPORTS

Often we don't want to export a set of related identifiers from a module, but instead want to represent the whole module through a single export. One fairly common situation in which this occurs is when our modules contain a single class, as in the following listing.

Listing 11.9   A default export from Ninja.js

```
export default class Ninja {                    Uses the export default
  constructor(name){                            keywords to specify the
    this.name = name;                           default module binding
  }
}

export function compareNinjas(ninja1, ninja2){  We can still use named exports
  return ninja1.name === ninja2.name;           along with the default export.
}
```

Here we've added the `default` keyword after the `export` keyword, which specifies the default binding for this module. In this case, the default binding for this module is the class named `Ninja`. Even though we've specified a default binding, we can still use named exports to export additional identifiers, as we did with the `compare-Ninjas` function.

Now, we can use simplified syntax to import functionalities from Ninja.js, as shown in the following listing.

Listing 11.10   Importing a default export

```
                                           When importing a default export, there's no
                                              need for braces, and we can use whatever
                                                                      name we want.
import ImportedNinja from "Ninja.js";      We can still import
import {compareNinjas} from "Ninja.js";    named exports.

const ninja1 = new ImportedNinja("Yoshi");
const ninja2 = new ImportedNinja("Hattori");    Creates a couple
                                                of ninjas, and
assert(ninja1 !== undefined                     tests that they
    && ninja2 !== undefined, "We can create a couple of Ninjas");  exist

assert(!compareNinjas(ninja1, ninja2),     We can also access
      "We can compare ninjas");            the named exports.
```

We start this example with importing a default export. For this, we use a less cluttered import syntax by dropping the braces that are mandatory for importing named exports. Also, notice that we can choose an arbitrary name to refer to the default export; we aren't bound to use the one we used when exporting. In this example, `ImportedNinja` refers to the `Ninja` class defined in the file Ninja.js.

We continue the example by importing a named export, as in previous examples, just to illustrate that we can have both a default export and a number of named exports within a single module. Finally, we instantiate a couple of `ninja` objects and call the `compareNinjas` function, to confirm that all imports work as they should.

In this case, both imports are made from the same file. ES6 offers a shorthand syntax:

```
import ImportedNinja, {compareNinjas} from "Ninja.js";
```

Here we use the comma operator to import both the default and the named exports from the Ninja.js file, in a single statement.

RENAMING EXPORTS AND IMPORTS

If necessary, we can also rename both exports and imports. Let's start with renaming exports, as shown in the following code (the comments indicate in which file the code is located):

```
//************* Greetings.js ************/
function sayHi(){
  return "Hello";
}

assert(typeof sayHi === "function"
    && typeof sayHello === "undefined",
      "Within the module we can access only sayHi");

export { sayHi as sayHello }

//************* main.js ***************/
import {sayHello } from "Greetings.js";

assert(typeof sayHi === "undefined"
    && typeof sayHello === "function",
      "When importing, we can only access the alias");
```

- **Defines a function called sayHi**
- **Tests that we can access only the sayHi function, but not the alias!**
- **Provides an identifier alias with the as keyword**
- **When importing, only the sayHello alias is available.**

In the previous example, we define a function called `sayHi`, and we test that we can access the function only though the `sayHi` identifier, and not through the `sayHello` alias that we provide at the end of the module through the as keyword:

```
export { sayHi as sayHello }
```

We can perform an export rename only in this export form, and not by prefixing the variable or function declaration with the `export` keyword.

Then, when we perform an import of the renamed export, we reference the import through the given alias:

```
import { sayHello } from "Greetings.js";
```

Finally, we test that we have access to the aliased identifier, but not the original one:

```
assert(typeof sayHi === "undefined"
    && typeof sayHello === "function",
      "When importing, we can only access the alias");
```

The situation is similar when renaming imports, as shown in the following code segment:

```
/************* Hello.js *************/
export function greet(){
  return "Hello";
}
```
Exports a function with the name greet from the Hello.js module

```
/************* Salute.js *************/
export function greet(){
  return "Salute";
}
```
Exports a function with the same name greet from Salute.js

```
/************* main.js *************/
import { greet as sayHello } from "Hello.js";
import { greet as salute } from "Salute.js";
```
Uses the as keyword to alias imports, thereby avoiding name clashes

```
assert(typeof greet === "undefined",
      "We cannot access greet");
```
We can't access the original function name.

```
assert(sayHello() === "Hello" && salute() === "Salute",
      "We can access aliased identifiers!");
```
But we can access the aliases.

Similarly to exporting identifiers, we can also use the as keyword to create aliases when importing identifiers from other modules. This is useful when we need to provide a better name that's more suitable to the current context, or when we want to avoid naming clashes, as is the case in this small example.

With this, we've finished our exploration of the ES6 modules' syntax, which is recapped in table 11.1.

Table 11.1   Overview of ES6 module syntax

Code	Meaning

`export const ninja = "Yoshi";`
`export function compare(){}`
`export class Ninja{}`
	Export a named variable.
	Export a named function.
	Export a named class.

`export default class Ninja{}` `export default function Ninja(){}`	Export the default class export. Export the default function export.

`const ninja = "Yoshi";`
`function compare(){};`
`export {ninja, compare};`
`export {ninja as samurai, compare};`
	Export existing variables.
	Export a variable through a new name.

`import Ninja from "Ninja.js";` `import {ninja, Ninja} from "Ninja.js";`	Import a default export. Import named exports.
`import * as Ninja from "Ninja.js";`	Import all named exports from a module.
`import {ninja as iNinja} from "Ninja.js";`	Import a named export through a new name.

11.3   Summary

- Large, monolithic code bases are far more likely to be difficult to understand and maintain than smaller, well-organized ones. One way of improving the structure and organization of our programs is to break them into smaller, relatively loosely coupled segments or modules.
- Modules are larger units of organizing code than objects and functions, and they allow us to divide programs into clusters that belong together.
- In general, modules foster understandability, ease maintenance, and improve the reusability of code.
- Pre-ES6 JavaScript has no built-in modules, and developers had to be creative with existing language features to enable code modularization. One of the most popular ways of creating modules is by combining immediately invoked functions with closures.
  - Immediate functions are used because they create a new scope for defining module variables that aren't visible from outside that scope.
  - Closures are used because they enable us to keep module variables alive.
  - The most popular pattern is the module pattern, which usually combines an immediate function with a return of a new object that represents the module's public interface.
- In addition to the module pattern, two popular module standards exist: Asynchronous Module Definition, designed to enable modules in the browser; and CommonJS, which is more popular in server-side JavaScript.
  - AMD can automatically resolve dependencies, and modules are asynchronously loaded, thereby avoiding blocking.
  - CommonJS has a simple syntax, synchronously loads modules (and is therefore more appropriate for the server), and has many packages available through node's package manager (npm).
- ES6 modules are designed to take into account the features of AMD and CommonJS. These modules have a simple syntax influenced by CommonJS, and provide asynchronous module loading as in AMD.
  - ES6 modules are file based, one module per file.
  - We export identifiers so that they can be referenced by other modules by using the new `export` keyword.
  - We import identifiers exported from other modules by using the `import` keyword.
  - A module can have a single `default` export, which we use if we want to represent that whole module through a single export.
  - Both imports and exports can be renamed with the `as` keyword.

11.4 Exercises

1 Which mechanism enables private module variables in the module pattern?

  a Prototypes

  b Closures

  c Promises

2 In the following code that uses ES6 modules, which identifiers can be accessed if the module is imported?

```
const spy = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
export const general = "Minamoto";
```

  a `spy`

  b `command`

  c `general`

3 In the following code that uses ES6 modules, which identifiers can be accessed when the module is imported?

```
const ninja = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
const general  = "Minamoto";

export {ninja as spy};
```

  a `spy`

  b `command`

  c `general`

  d `ninja`

4 Which of the following imports are allowed?

```
//File: personnel.js
const ninja = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
const general  = "Minamoto";

export {ninja as spy};
```

  a `import {ninja, spy, general} from "personnel.js"`

  b `import * as Personnel from "personnel.js"`

  c `import {spy} from "personnel.js"`

5 If we have the following module code, which statement will import the `Ninja` class?

```
//Ninja.js
export default class Ninja {
  skulk(){ return "skulking"; }
}
```

a import Ninja from "Ninja.js"
b import * as Ninja from "Ninja.js"
c import * from "Ninja.js"

Part 4

Browser reconnaissance

Now that we've explored the fundamentals of the JavaScript language, we'll move on to browsers, the environment in which most JavaScript applications are executed.

In chapter 12, we'll take a closer look at the DOM by exploring efficient techniques for modifying the DOM and achieving fast, highly dynamic web applications.

In chapter 13, you'll learn about events, with a special focus on the event loop and its influence on perceived web application performance.

Finally, the book concludes with a not-so-pleasant but necessary topic: cross-browser development. Although the state of affairs has improved greatly in recent years, we still can't assume that our code will work in the same way in every available browser. Therefore, chapter 14 presents strategies for developing cross-browser web applications.

Working the DOM

Up to now, you've been learning mostly about JavaScript the language, and although there are plenty of nuances to pure JavaScript, developing web applications definitely doesn't get any easier when we throw the browser's Document Object Model (DOM) into the mix. One of the primary means for achieving highly dynamic web applications that respond to user actions is by modifying the DOM. But if we were to open up a JavaScript library, you'd notice the length and complexity of the code behind simple DOM operations. Even presumably simple operations like `cloneNode` and `removeChild` have relatively complex implementations.

This raises two questions:

- Why is this code so complex?
- Why do you need to understand how it works if the library will take care of it for you?

The most compelling reason is *performance.* Understanding how DOM modification works in libraries can allow you to write better and faster code that uses the library or, alternatively, enable you to use those techniques in your own code.

 So we'll start this chapter by seeing how to create new parts of our pages, on demand, by injecting arbitrary HTML. We'll continue by examining all the conundrums that browsers throw at us with respect to element properties and attributes, and we'll discover why the results aren't always exactly what we might expect.

The same goes for Cascading Style Sheets (CSS) and the styling of elements. Many of the difficulties that we'll run into when constructing a dynamic web application stem from the complications of setting and getting element styling. This book can't cover all that's known about handling element styling (that's enough to fill another entire book), but the core essentials are discussed.

We'll finish the chapter by taking a look at some of the performance difficulties that can arise if you don't pay attention to the way you modify and read information from the DOM. Let's start by seeing how to inject arbitrary HTML into our pages.

. . . .

Do you know?

Why do you need to preparse self-closing elements in a page before injecting HTML into it?

What are the benefits of working with DOM fragments when inserting HTML?

How do you determine the dimensions of a hidden element on a page?

. . . .

12.1   *Injecting HTML into the DOM*

In this section, we'll look at an efficient way to insert HTML into a document at any location, given that HTML as a string. We present this particular technique because it's frequently used to create highly dynamic web pages in which the user interface is modified as a response to user actions or data incoming from the server. This is particularly useful for the following scenarios:

- Injecting arbitrary HTML into a page and manipulating and inserting client-side templates
- Retrieving and injecting HTML sent from a server

It can be technically challenging to implement this functionality correctly (especially when compared to building an object-oriented-style DOM construction API, which is certainly easier to implement but requires an extra layer of abstraction than injecting the HTML). Consider this example of creating HTML elements from an HTML string that we can use with jQuery:

```
$(document.body).append("<div><h1>Greetings</h1><p>Yoshi here</p></div>")
```

And compare that with an approach that uses only the DOM API:

```
const h1 = document.createElement("h1");
h1.textContent = "Greetings";

const p = document.createElement("p");
p.textContent = "Yoshi here";

const div = document.createElement("div");

div.appendChild(h1);
div.appendChild(p);

document.body.appendChild(div);
```

Which one would you rather use?

For these reasons, we'll implement our own way of doing clean DOM manipulation from scratch. The implementation requires the following steps:

1 Convert an arbitrary but valid HTML string into a DOM structure.
2 Inject that DOM structure into any location in the DOM as efficiently as possible.

These steps provide page authors with a smart way to inject HTML into a document. Let's get started.

12.1.1 Converting HTML to DOM

Converting an HTML string to a DOM structure doesn't involve a whole lot of magic. In fact, it uses a tool that you're most likely already familiar with: the `innerHTML` property of DOM elements.

Using it is a multistep process:

1 Make sure that the HTML string contains valid HTML code.
2 Wrap the string in any enclosing markup that's required by browser rules.
3 Insert the HTML string, using `innerHTML`, into a dummy DOM element.
4 Extract the DOM nodes back out.

The steps aren't overly complex, but the actual insertion has some gotchas that we'll need to take into account. Let's take a look at each step in detail.

PREPROCESSING THE HTML SOURCE STRING

To start, we'll need to clean up the source HTML to meet our needs. For example, let's take a look at a skeleton HTML that allows us to choose a ninja (through the `option` element) and that shows the details of the chosen ninja within a table, details that are intended to be added at a later point:

```
<option>Yoshi</option>
<option>Kuma</option>
<table/>
```

This HTML string has two problems. First, the option elements shouldn't stand on their own. If you follow proper HTML semantics, they should be contained within a `select` element. Second, even though markup languages usually allow us to self-close childless elements, such as `<table/>`, in HTML the self-closing works for only a small subset of elements (`table` not being one of them). Attempting to use that syntax in other cases is likely to cause problems in some browsers.

Let's start with solving the problem of self-closing elements. To support this feature, we can do a quick preparse on the HTML string to convert elements such as `<table/>` to `<table></table>` (which will be handled uniformly in all browsers), as shown in the following listing.

Listing 12.1   Making sure that self-closing elements are interpreted correctly

> **Uses a regular expression to match the tag name of any elements we don't need to be concerned about**

```
const tags =
➥    /^(area|base|br|col|embed|hr|img|input|keygen|link|menuitem|meta|param|
➥        source|track|wbr)$/i;
function convert(html) {
  return html.replace(
➥          /(<(\w+)[^>]*?)\/>/g, (all, front, tag) => {
      return tags.test(tag) ? all :
                           front + "></" + tag + ">";
    });
  }
  assert(convert("<a/>") === "<a></a>", "Check anchor conversion.");
  assert(convert("<hr/>") === "<hr/>", "Check hr conversion.");
```

> **A function that uses regular expressions to convert self-closing tags to "normal" form**

When we apply the `convert` function to this example HTML string, we end up with the following HTML string:

```
<option>Yoshi</option>
<option>Kuma</option>
<table></table>                    ⊲——— <table/> expanded
```

With that accomplished, we still have to solve the problem that our `option` elements aren't contained within a `select` element. Let's see how to determine whether an element needs to be wrapped.

HTML WRAPPING

According to the semantics of HTML, some HTML elements must be within certain container elements before they can be injected. For example, an `<option>` element must be contained within a `<select>`.

We can solve this problem in two ways, both of which require constructing a mapping between problematic elements and their containers:

- The string could be injected directly into a specific parent by using `innerHTML`, where the parent has been previously constructed using the built-in `document createElement`. Although this may work in some cases and in some browsers, it isn't universally guaranteed.
- The string could be wrapped with the appropriate required markup and then injected directly into any container element (such as a `<div>`). This is more foolproof, but it's also more work.

The second technique is preferred; it involves little browser-specific code, in contrast to the first approach, which requires a fair amount of mostly browser-specific code.

The set of problematic elements that need to be wrapped in specific container elements is fortunately a rather manageable seven. In table 12.1, the ellipses (...) indicates the locations where the elements need to be injected.

Table 12.1 Elements that need to be contained within other elements

Element name	Ancestor element
`<option>`, `<optgroup>`	`<select multiple>...</select>`
`<legend>`	`<fieldset>...</fieldset>`
`<thead>`, `<tbody>`, `<tfoot>`, `<colgroup>`, `<caption>`	`<table>...</table>`
`<tr>`	`<table><thead>...</thead></table>` `<table><tbody>...</tbody></table>` `<table><tfoot>...</tfoot></table>`
`<td>`, `<th>`	`<table><tbody><tr>...</tr></tbody></table>`

`<col>`	`<table>`
	`  <tbody></tbody>`
	`  <colgroup>...</colgroup>`
	`</table>`

Nearly all of these are straightforward, save for the following points, which require a bit of explanation:

- A `<select>` element with the `multiple` attribute is used (as opposed to a non-multiple select) because it won't automatically check any of the options that are placed inside it (whereas a single select will autocheck the first option).
- The `<col>` fix includes an extra `<tbody>`, without which the `<colgroup>` won't be generated properly.

With the elements properly mapped to their wrapping requirements, let's start generating.

With the information from table 12.1, we can generate the HTML that we need to insert into a DOM element, as shown in the following listing.

```
function getNodes(htmlString, doc) {
    const map = {
        "<td":[3,"<table><tbody><tr>","</tr></tbody></table>"],
        "<th":[3,"<table><tbody><tr>","</tr></tbody></table>"],
        "<tr":[2,"<table><thead>","</thead></table>"],
        "<option":[1,"<select multiple>","</select>"],
        "<optgroup":[1,"<select multiple>","</select>"],
        "<legend":[1,"<fieldset>","</fieldset>"],
        "<thead":[1,"<table>","</table>"],
        "<tbody":[1,"<table>","</table>"],
        "<tfoot":[1,"<table>","</table>"],
        "<colgroup":[1,"<table>","</table>"],
        "<caption":[1,"<table>","</table>"],
        "<col":[2,"<table><tbody></tbody><colgroup>","</colgroup></table>"],
    };
    const tagName = htmlString.match(/<\w+/);
    let mapEntry = tagName ? map[tagName[0]] : null;
    if (!mapEntry) { mapEntry = [0, " "," " ];}
    let div = (doc || document).createElement("div");
    div.innerHTML = mapEntry[1] + htmlString + mapEntry[2];
    while (mapEntry[0]--) { div = div.lastChild;}
    return div.childNodes;
}
assert(getNodes("<td>test</td><td>test2</td>").length === 2,
        "Get two nodes back from the method.");
assert(getNodes("<td>test</td>")[0].nodeName === "TD",
        "Verify that we're getting the right node.");
```

Map of element types that need special parent containers. Each entry has the depth of the new node, opening HTML for the parents, and closing HTML for the parents.

Matches the opening bracket and tag name

If it's in the map, grabs the entry; otherwise, constructs a faux entry with empty "parent" markup and a depth of zero.

Creates a <div> element in which to create the new nodes. Note that we use a passed document if it exists, or default to the current document if not.

Wraps the incoming markup with the parents from the map entry, and injects it as the inner HTML of the newly created <div>

Returns the newly created element

Walks down the just-created tree to the depth indicated by the map entry. This should be the parent of the desired node created from the markup.

We create a map of all element types that need to be placed within special parent containers, a map that contains the depth of the node, as well as the enclosing HTML. Next, we use a regular expression to match the opening bracket and the tag name of the element we want to insert:

```
const tagName = htmlString.match(/<\w+/);
```

Then we select a map entry, and in case there isn't one, we create a dummy entry with an empty parent element markup:

```
let mapEntry = tagName ? map[tagName[0]] : null;
if  (!mapEntry) { mapEntry = [0, " ", " "]; }
```

We follow this by creating a new div element, surrounding it with the mapped HTML, and inserting the newly created HTML into the previously created div element:

```
let div = (doc || document).createElement("div");
div.innerHTML = mapEntry[1] + htmlString + mapEntry[2]
```

Finally, we find the parent of the desired node created from our HTML string, and we return the newly created node:

```
while (mapEntry[0]--) { div = div.lastChild;}
return div.childNodes;
```

After all of this, we have a set of DOM nodes that we can begin to insert into the document.

If we go back to our motivating example, and apply the `getNodes` function, we'll end up with something along the following lines:

```
<select multiple>
  <option>Yoshi</option>
  <option>Kuma</option>
</select>
<table></table>
```

Option elements have been wrapped inside a select element.

12.1.2 Inserting elements into the document

After we have the DOM nodes, it's time to insert them into the document. A couple of steps are required, and we'll work through them in this section.

Because we have an array of elements that we need to insert—potentially into any number of locations within the document—we want to try to keep the number of operations performed to a minimum. We can do this by using *DOM fragments*. DOM fragments are part of the W3C DOM specification and are supported in all browsers. This useful facility gives us a container to hold a collection of DOM nodes.

This in itself is quite useful, but it also has the advantage that the fragment can be injected and cloned in a single operation instead of having to inject and clone each individual node over and over again. This has the potential to dramatically reduce the number of operations required for a page.

Before we use this mechanism in our code, let's revisit the `getNodes()` code of listing 12.2 and adjust it a tad to use DOM fragments. The changes are minor and consist of adding a `fragment` parameter to the function's parameter list, as follows.

> **Listing 12.3 Expanding the `getNodes` function with fragments**

```
function getNodes(htmlString, doc, fragment){          Adds a new
    const map = {                                       fragment
      "<td":[3,"<table><tbody><tr>","</tr></tbody></table>"],    parameter to
      "<th":[3,"<table><tbody><tr>","</tr></tbody></table>"],    the function
      "<tr":[2,"<table><thead>","</thead></table>"],
      "<option":[1,"<select multiple>","</select>"],
      "<optgroup":[1,"<select multiple>","</select>"],
      "<legend":[1,"<fieldset>","</fieldset>"],
      "<thead":[1,"<table>","</table>"],
      "<tbody":[1,"<table>","</table>"],
      "<tfoot":[1,"<table>","</table>"],
      "<colgroup":[1,"<table>","</table>"],
      "<caption":[1,"<table>","</table>"],
```

```
    "<col":[2,"<table><tbody></tbody><colgroup>","</colgroup></table>"],
  };
  const tagName = htmlString.match(/<\w+/);
  let mapEntry = tagName ? map[tagName[0]] : null;
  if  (!mapEntry) { mapEntry = [0, " "," " ];}
  let div = (doc || document).createElement("div");
  div.innerHTML = mapEntry[1] + htmlString + mapEntry[2];
  while (mapEntry[0]--) { div = div.lastChild;}
  if (fragment) {
    while (div.firstChild) {
      fragment.appendChild(div.firstChild);
    }
  }

  return div.childNodes;
}
```

> **If the fragment exists, injects the nodes into it.**

In this example, we make a couple of changes. First we modify the function signature by adding another parameter, `fragment`:

```
function getNodes(htmlString, doc, fragment) {...}
```

This parameter, if it's passed, is expected to be a DOM fragment that we want the nodes to be injected into for later use.

To do so, we add the following fragment just before the `return` statement of the function to add the nodes to the passed fragment:

```
if (fragment) {
  while (div.firstChild) {
    fragment.appendChild(div.firstChild);
  }
}
```

Now, let's see it in use. In the following listing, which assumes that the updated `get-Nodes` function is in scope, a fragment is created and passed in to that function (which, you may recall, converts the incoming HTML string into DOM elements). This DOM is now appended to the fragment.

Listing 12.4   Inserting a DOM fragment into multiple locations in the DOM

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>
<div id="test2"></div>
<script>
  document.addEventListener("DOMContentLoaded", () => {
    function insert(elems, args, callback) {
      if (elems.length) {
        const doc = elems[0].ownerDocument || elems[0],
              fragment = doc.createDocumentFragment(),
              scripts = getNodes(args, doc, fragment),
              first = fragment.firstChild;
```

> **Sets up a couple of test nodes**

> **Creates a document fragment in which we'll insert nodes**

> **Creates HTML nodes from the HTML string**

```
        if (first) {
          for (let i = 0; elems[i]; i++) {
            callback.call(root(elems[i], first),
                i > 0 ? fragment.cloneNode(true) : fragment);    ◁──┐
          }
        }                                                           ┌──────────────────┐
      }                                                             │ If we need to insert the │
    }                                                               │ nodes into more than one │
    const divs = document.querySelectorAll("div");                  │ element, we have to clone │
    insert(divs, "<b>Name:</b>", function (fragment) {              │ the fragment each time. │
      this.appendChild(fragment);                                   └──────────────────┘
    });

    insert(divs, "<span>First</span> <span>Last</span>",
        function (fragment) {
          this.parentNode.insertBefore(fragment, this);
        });
  });
</script>
```

There's another important point here: If we're inserting this element into more than one location in the document, we're going to need to clone this fragment again and again. If we weren't using a fragment, we'd have to clone each individual node every time, instead of the whole fragment at once.

With this, we've developed a way to generate and insert arbitrary DOM elements in an intuitive manner. Let's continue this exploration of the DOM by seeing the difference between DOM attributes and properties.

12.2 *Using DOM attributes and properties*

When accessing the values of element attributes, we have two options: using the traditional DOM methods of `getAttribute` and `setAttribute`, or using properties of the DOM objects that correspond to the attributes.

For example, to obtain the `id` of an element whose reference is stored in variable e, we could use either of the following:

```
e.getAttribute('id')
e.id
```

Either will give us the value of the `id`.

Let's examine the following code to better understand how attribute values and their corresponding properties behave.

Listing 12.5 Accessing attribute values via DOM methods and properties

```
<div></div>
<script>
  document.addEventListener("DOMContentLoaded", () => {                ┌──────────────────┐
    const div = document.querySelector("div");          ◁──           │ Obtains an element │
                                                                      │ reference │
                                                                      └──────────────────┘
```

```
div.setAttribute("id","ninja-1");
assert(div.getAttribute('id') === "ninja-1",
       "Attribute successfully changed");
div.id = "ninja-2";
assert(div.id === "ninja-2",
       "Property successfully changed");

assert(div.getAttribute('id') === "ninja-2",
       "Attribute successfully changed via property");
div.setAttribute("id","ninja-3");
assert(div.id === "ninja-3",
       "Property successfully changed via attribute");
assert(div.getAttribute('id') === "ninja-3",
       "Attribute successfully changed");
});
</script>
```

Using setAttribute also changes the value obtained through the property.

Changes the value of the id attribute with the setAttribute method and tests that the value has changed

Changes the value of the property and tests that the value has changed

Changing the property also changes the value obtained with getAttribute.

This example shows interesting behavior with respect to element attributes and element properties. It starts by defining a simple `<div>` element that we'll use as a test subject. Within the document's `DOMContentLoaded` handler (to ensure that the DOM is fully built), we obtain a reference to the lone `<div>` element, `const div = document .querySelector("div")`, and then run a few tests.

In the first test, we set the `id` attribute to the value `ninja-1` via the `setAttribute()` method. Then we assert that `getAttribute()` returns the same value for that attribute. It should be no surprise that this test works just fine when we load the page:

```
div.setAttribute("id", "ninja-1");
assert(div.getAttribute('id') === "ninja-1",
       "Attribute successfully changed");
```

Similarly, in the next test, we set the `id` property to the value `ninja-2` and then verify that the property value has indeed changed. No problem.

```
div.id = "ninja-2";
assert(div.id === "ninja-2",
       "Property successfully changed");
```

The next test is when things get interesting. We again set the `id` property to a new value, in this case `ninja-3`, and again verify that the property value has changed. But then we also assert that not only should the property value change, but also the value of the `id` *attribute*. Both assertions pass. From this we learn that the `id` property and the `id` attribute are somehow linked together. Changing the `id` property value also changes the `id` attribute value:

```
div.id = "ninja-3";
assert(div.id === "ninja-3",
       "Property successfully changed");
assert(div.getAttribute('id') === "ninja-3",
        "Attribute successfully changed via property");
```

The next test proves that it also works the other way around: Setting an attribute value also changes the corresponding property value.

```
div.setAttribute("id","ninja-4");
assert(div.id === "ninja-4",
       "Property successfully changed via attribute");
assert(div.getAttribute('id') === "ninja-4","Attribute changed");
```

But don't let this fool you into thinking that the property and attribute are sharing the same value—they aren't. We'll see later in this chapter that the attribute and corresponding property, although linked, aren't always identical.

It's important to note that not all attributes are represented by element properties. Although it's generally true for attributes that are natively specified by the HTML DOM, *custom attributes* that we may place on the elements in our pages don't automatically become represented by element properties. To access the value of a custom attribute, we need to use the DOM methods `getAttribute()` and `setAttribute()`.

If you're not sure whether a property for an attribute exists, you can always test for it and fall back to the DOM methods if it doesn't exist. Here's an example:

```
const value = element.someValue ? element.someValue
                                : element.getAttribute('someValue');
```

> **TIP** In HTML5, use the prefix `data-` for all custom attributes to keep them
> valid in the eye of the HTML5 specification. It's a good convention that clearly
> separates custom attributes from native attributes.

12.3   *Styling attribute headaches*

As with general attributes, getting and setting styling attributes can be a headache. As with the attributes and properties in the previous section, we again have two approaches for handling `style` values: the attribute value, and the element property created from it.

The most commonly used of these is the `style` element property, which isn't a string but an object that holds properties corresponding to the style values specified in the element markup. In addition, you'll see that there's a method for accessing the computed style information of an element, where *computed style* means the style that will be applied to the element after evaluating all inherited and applied style information.

This section outlines the things you need to know when working with styles in browsers. Let's start with a look at where style information is recorded.

12.3.1   *Where are my styles?*

The style information located in the `style` property of a DOM element is initially set from the value specified for the `style` attribute in the element markup. For example, `style="color:red;"` results in that style information being placed into the style object. During page execution, the script can set or modify values in the style object, and these changes will actively affect the display of the element.

Many script authors are disappointed to find that no values from on-page `<style>` elements or external style sheets are available in the element's `style` object. But we won't stay disappointed for long—you'll soon see a way to obtain this information.

For now, let's see how the `style` property gets its values. Examine the following code.

Listing 12.6   Examining the `style` property

Declares an in-page style sheet that applies
font size and border information

This test element should
receive multiple styles from
various places, including its
own style attribute and the
style sheet.

```
<style>
  div { font-size: 1.8em; border: 0 solid gold; }
</style>
<div style="color:#000;" title="Ninja power!">
  忍者パワー
</div>
<script>
  document.addEventListener("DOMContentLoaded", () => {
    const div = document.querySelector("div");
    assert(div.style.color === 'rgb(0, 0, 0)' ||
           div.style.color === '#000',
           'color was recorded');
    assert(div.style.fontSize === '1.8em',
           'fontSize was recorded');
    assert(div.style.borderWidth === '0',
           'borderWidth was recorded');
    div.style.borderWidth = "4px";
    assert(div.style.borderWidth === '4px',
           'borderWidth was replaced');
  });
</script>
```

Tests that the
inlined color style
was recorded

Tests that the inherited font
size style was recorded

Tests that the inherited border
width style was recorded

Replaces the border
width style

Tests the border width style
change was recorded

In this example, we set up a `<style>` element to establish an internal style sheet whose values will be applied to the elements on the page. The style sheet specifies that all `<div>` elements will appear in a font size that's 1.8 times bigger than the default, with a solid gold border of `0` width. Any elements to which this is applied will possess a border, but it won't be visible because it has a width of `0`.

```
<style>
  div { font-size: 1.8em; border: 0 solid gold; }
</style>
```

Then we create a `<div>` element with an inlined style attribute that colors the text of the element black:

```
<div style="color:#000;" title="Ninja power!">
  忍者パワー
</div>
```

We then begin the testing. After obtaining a reference to the `<div>` element, we test that the `style` attribute receives a `color` property that represents the color assigned to the element. Note that even though the `color` is specified as `#000` in the inline style, it's normalized to RGB notation when set in the `style` property in most browsers (so we check both formats).

```
assert(div.style.color === 'rgb(0, 0, 0)' ||
        div.style.color === '#000',
        'color was recorded');
```

Looking ahead, in figure 12.1, we see that this test passes.

Then we naïvely test that the font-size styling and the border width specified in the inline style sheet have been recorded in the style object. But even though we can see in figure 12.1 that the font-size style has been applied to the element, the test fails. This is because the style object doesn't reflect any style information inherited from CSS style sheets:

Figure 12.1 **By running this test, we can see that inline and assigned styles are recorded, but inherited styles aren't.**

```
assert(div.style.fontSize === '1.8em',
        'fontSize was recorded');
assert(div.style.borderWidth === '0',
        'borderWidth was recorded');
```

Moving on, we use an assignment to change the value of the `borderWidth` property in the style object to 4 pixels wide and test that the change is applied. We can see in figure 12.1 that the test passes and that the previously invisible border is applied to the element. This assignment causes a `borderWidth` property to appear in the `style` property of the element, as proven by the test.

```
div.style.borderWidth = "4px";
assert(div.style.borderWidth === '4px',
      'borderWidth was replaced');
```

It should be noted that any values in an element's `style` property take precedence over anything inherited by a style sheet (even if the style sheet rule uses the `!important` annotation).

One thing that you may have noted in listing 12.6 is that CSS specifies the font size property as `font-size`, but in script you reference it as `fontSize`. Why is that?

12.3.2 *Style property naming*

CSS attributes cause relatively few cross-browser difficulties when it comes to accessing the values provided by the browser. But differences between how CSS names styles and how we access those in script do exist, and some style names differ across browsers.

CSS attributes that span more than one word separate the words with a hyphen; examples are `font-weight`, `font-size`, and `background-color`. You may recall that property names in JavaScript *can* contain a hyphen, but including a hyphen prevents the property from being accessed via the dot operator.

Consider this example:

```
const fontSize = element.style['font-size'];
```

The preceding is perfectly valid. But the following isn't:

```
const fontSize = element.style.font-size;
```

The JavaScript parser would see the hyphen as a subtraction operator, and nobody would be happy with the outcome. Rather than forcing page developers to always use the general form for property access, *multiword CSS style names are converted to camel case when used as a property name.* As a result, `font-size` becomes `fontSize`, and `background-color` becomes `backgroundColor`.

We can either remember to do this, or write a simple API to set or get styles that automatically handle the camel casing, as shown in the following listing.

Listing 12.7   A simple method for accessing styles

Defines the style function that will assign a value to a style property in case a value is provided, and in case it isn't, it will simply return the value of the style property. We can use this function for both setting and getting the value of a style property.

```
<div style="color:red;font-size:10px;background-color:#eee;"></div>
<script>
  function style(element,name,value){
    name = name.replace(/-([a-z])/ig, (all,letter) => {      ◁── Converts name
      return letter.toUpperCase();                                   to camel case
    });
    if (typeof value !== 'undefined') {      ◁──
      element.style[name] = value;                  The new value of the
    }                                               style property is set, if
    return element.style[name];                     a value is provided.
  }
  document.addEventListener("DOMContentLoaded", () => {
    const div = document.querySelector("div");
    assert(style(div,'color') === "red", style(div,'color'));
    assert(style(div,'font-size') === "10px", style(div,'font-size'));
    assert(style(div,'background-color') ===
    "rgb(238, 238, 238)",style(div,'background-color'));
  });
</script>
```

Always returns the value of the style property

The style function has two important characteristics:

- It uses a regular expression to convert the `name` parameter to camel-case notation. (If the regex-driven conversion operation has you scratching your head, you might want to review the material in chapter 10.)
- It can be used both as a setter and a getter, by inspecting its own argument list. For example, we can obtain the value of the font-size property with `style(div, 'font-size')`, and we can set a new value with `style(div, 'font-size', '5px')`.

Consider the following code:

```
function style(element,name,value){
    ...
  if (typeof value !== 'undefined') {
    element.style[name] = value;
  }

  return element.style[name];
}
```

If a `value` argument is passed to the function, the function acts as a setter, setting the passed value as the value of the attribute. If the `value` argument is omitted and only the first two arguments are passed, it acts as a getter, retrieving the value of the specified attribute. In either case, the value of the attribute is returned, which makes it easy to use the function in either of its modes in a function-call chain.

The `style` property of an element doesn't include any style information that an element inherits from style sheets in scope for the element. Many times it would be handy to know the full computed style that's been applied to an element, so let's see if there's a way to obtain that.

12.3.3 *Fetching computed styles*

At any point in time, the *computed style* of an element is a combination of all the built-in styles provided by the browser, all the styles applied to it via style sheets, the element's `style` attribute, and any manipulations of the `style` property by script. Figure 12.2 shows how browser developer tools differentiate between styles.

The standard method, implemented by all modern browsers, is the `getComputed-Style` method. This method accepts an element whose styles are to be computed and returns an interface through which property queries can be made. The returned interface provides a method named `getPropertyValue` for retrieving the computed style of a specific style property.

Unlike the properties of an element's `style` object, the `getPropertyValue` method accepts CSS property names (such as `font-size` and `background-color`) rather than the camel-cased versions of those names.
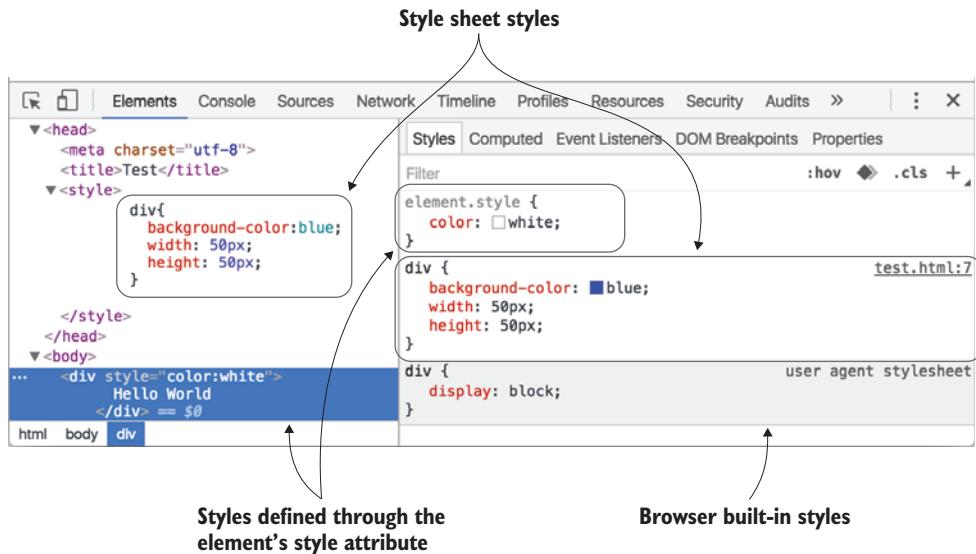
Style sheet styles

Figure 12.2   The final style associated with an element can come from many things: the browser built-in styles (user agent style sheet), the styles assigned through the style property, and styles from the CSS rules defined in CSS code.

The following listing shows a simple example.

Listing 12.8   Fetching computed style values

```
<style>
  div {
    background-color: #ffc; display: inline; font-size: 1.8em;
    border: 1px solid crimson; color: green;
  }
</style>
<div style="color:crimson;" id="testSubject" title="Ninja power!">
  忍者パワー
</div>
<script>
  function fetchComputedStyle(element,property) {
    const computedStyles = getComputedStyle(element);
    if (computedStyles) {
      property = property.replace(/([A-Z])/g,'-$1').toLowerCase();
      return computedStyles.getPropertyValue(property);
    }
  }
  document.addEventListener("DOMContentLoaded", () => {
    const div = document.querySelector("div");
    report("background-color: " +
           fetchComputedStyle(div,'background-color'));
    report("display: " +
           fetchComputedStyle(div,'display'));
```

Creates a test subject with a style attribute

Defines a function that will get the computed value of a style property

Uses the built-in getComputedStyle method to obtain a descriptor object

Replaces camel-case notation with dashes

Tests that we can obtain the values of various style properties, using different notations

```
        report("font-size: " +
            fetchComputedStyle(div,'fontSize'));
        report("color: " +
            fetchComputedStyle(div,'color'));
        report("border-top-color: " +
            fetchComputedStyle(div,'borderTopColor'));
        report("border-top-width: " +
            fetchComputedStyle(div,'border-top-width'));
    });
</script>
```

To test the function that we'll be creating, we set up an element that specifies style information in its markup and a style sheet that provides style rules to be applied to the element. We expect that the computed styles will be the result of applying both the immediate and the applied styles to the element.

We then define the new function, which accepts an element and the style property that we want to find the computed value for. And to be especially friendly (after all, we're ninjas—making things easier for those using our code is part of the job), we'll allow multiword property names to be specified in either format: dashed or camel-cased. In other words, we'll accept both `backgroundColor` and `background-color`. We'll see how to accomplish that in a little bit.

The first thing we want to do is to obtain the computed style interface, which we store in a variable, `computedStyles`, for later reference. We want to do things this way because we don't know how expensive making this call may be, and it's likely best to avoid repeating it needlessly.

```
const computedStyles = getComputedStyle(element);
if (computedStyles) {
  property = property.replace(/([A-Z])/g,'-$1').toLowerCase();
  return computedStyles.getPropertyValue(property);
}
```

If that succeeds (and we can't think of any reason why it wouldn't, but it frequently pays to be cautious), we call the `getPropertyValue()` method of the interface to get the computed style value. But first we adjust the name of the property to accommodate either the camel-cased or dashed version of the property name. The `getPropertyValue` method expects the dashed version, so we use the `String`'s `replace()` method, with a simple but clever regular expression, to insert a hyphen before every uppercase character and then lowercase the whole thing. (Bet that was easier than you thought it would be.)

To test the function, we make calls to the function, passing various style names in various formats, and display the results, as shown in figure 12.3.

Note that the styles are fetched regardless of whether they're explicitly declared on the element or inherited from the style sheet. Also note that the `color` property, specified in both the style sheet and directly on the element, returns the explicit value. Styles specified by an element's `style` attribute always take precedence over inherited styles, even if marked `!important`.

We need to be aware of one more topic when dealing with style properties: *amalgam* properties. CSS allows us to use a shortcut notation for the amalgam of properties such as the `border-` properties. Rather than forcing us to specify colors, widths, and border styles individually and for all four borders, we can use a rule such as this:

```
border: 1px solid crimson;
```

We used this exact rule in listing 12.8. This saves a lot of typing, but we need to be aware that when we retrieve the properties, we need to fetch the low-level individual properties. We can't

Figure 12.3   Computed styles include all styles specified with the element as well as those inherited from style sheets.

fetch `border`, but we can fetch styles such as `border-top-color` and `border-top-width`, just as we did in the example.

It can be a bit of a hassle, especially when all four styles are given the same values, but that's the hand we've been dealt.

12.3.4 *Converting pixel values*

An important point to consider when setting style values is the assignment of numeric values that represent pixels. When setting a numeric value for a style property, we must specify the unit in order for it to work reliably across all browsers. For example, let's say that we want to set the `height` style value of an element to 10 pixels. Either of the following is a safe way to do this across browsers:

```
element.style.height = "10px";
element.style.height = 10 + "px";
```

The following isn't safe across browsers:

```
element.style.height = 10;
```

You might think it'd be easy to add a little logic to the `style()` function of listing 12.7 to tack a px to the end of a numeric value coming into the function. But not so fast! Not all numeric values represent pixels! Some style properties take numeric values that don't represent a pixel dimension. The list includes the following:

- `z-index`
- `font-weight`
- `opacity`
- `zoom`
- `line-height`

For these (and any others you can think of), go ahead and extend the function of listing 12.6 to automatically handle nonpixel values. Also, when attempting to read a pixel value out of a style attribute, the `parseFloat` method should be used to make sure that you get the intended value under all circumstances.

Now let's take a look at a set of important style properties that can be tough to handle.

12.3.5 *Measuring heights and widths*

Style properties such as `height` and `width` pose a special problem, because their values default to `auto` when not specified, so that the element sizes itself according to its contents. As a result, we can't use the `height` and `width` style properties to get accurate values unless explicit values are provided in the attribute string.

Thankfully, the `offsetHeight` and `offsetWidth` properties provide just that: a fairly reliable means to access the height and width of an element. But be aware that the values assigned to these two properties include the padding of the element. This information is usually exactly what we want if we're attempting to position one element over another. But sometimes we may want to obtain information about the element's dimensions with and without borders and padding.

Something to watch out for, however, is that in highly interactive sites, elements will likely spend some of their time in a nondisplayed state (with the `display` style being set to `none`), and when an element isn't part of the display, it has no dimensions. Any attempt to fetch the `offsetWidth` or `offsetHeight` properties of a nondisplayed element will result in a value of `0`.

For such hidden elements, if we want to obtain the nonhidden dimensions, we can employ a trick to momentarily unhide the element, grab the values, and hide it again. Of course, we want to do so in such a way that we leave no visible clue that this is going on behind the scenes. How can we make a hidden element not hidden without making it visible?

Employing our ninja skills, we can do it! Here's how:

1 Change the `display` property to `block`.
2 Set `visibility` to `hidden`.
3 Set `position` to `absolute`.
4 Grab the dimension values.
5 Restore the changed properties.

Changing the `display` property to `block` allows us to grab the values of `offsetHeight` and `offsetWidth`, but that makes the element part of the display and therefore visible. To make the element invisible, we'll set the `visibility` property to `hidden`. But (there's always another *but*) that will leave a big hole where the element is positioned, so we also set the `position` property to `absolute` to take the element out of the normal display flow.

All that sounds more complicated than the implementation, which is shown in the following listing.

Listing 12.9   Grabbing the dimensions of hidden elements

```html
<div>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  Suspendisse congue facilisis dignissim. Fusce sodales,
  odio commodo accumsan commodo, lacus odio aliquet purus,
  <img src="../images/ninja-with-pole.png" id="withPole" alt="ninja pole"/>
  <img src="../images/ninja-with-shuriken.png"
       id="withShuriken" style="display:none" alt="ninja shuriken" />
  vel rhoncus elit sem quis libero. Cum sociis natoque
  penatibus et magnis dis parturient montes, nascetur
  ridiculus mus. In hac habitasse platea dictumst. Donec
  adipiscing urna ut nibh vestibulum vitae mattis leo
  rutrum. Etiam a lectus ut nunc mattis laoreet at
  placerat nulla. Aenean tincidunt lorem eu dolor commodo
  ornare.
</div>
<script>
  (function(){                                         ⟵——— Creates a private scope
    const PROPERTIES = {                          ⟵┐
      position: "absolute",                          │ Defines target properties
      visibility: "hidden",                          │
      display: "block"                               │
    };
    window.getDimensions = element => {        ⟵——— Creates the new function
      const previous = {};                         ⟵┐
      for (let key in PROPERTIES) {                  │ Remembers settings
        previous[key] = element.style[key];          │
        element.style[key] = PROPERTIES[key];   ⟵——— Replaces settings
      }
      const result = {                         ⟵——— Fetches dimensions
        width: element.offsetWidth,
        height: element.offsetHeight
      };
      for (let in PROPERTIES) {                 ⟵——— Restores settings
        element.style[key] = previous[key];
      }
      return result;
    };
  })();
  document.addEventListener("DOMContentLoaded", () => {
    setTimeout(() => {
      const withPole = document.getElementById('withPole'),
          withShuriken = document.getElementById('withShuriken');
      assert(withPole.offsetWidth === 41,
             "Pole image width fetched; actual: " +
             withPole.offsetWidth + ", expected: 41");   ⟵——— Tests visible element
      assert(withPole.offsetHeight === 48,
             "Pole image height fetched: actual: " +
             withPole.offsetHeight + ", expected 48");
      assert(withShuriken.offsetWidth === 36,              ⟵——— Tests hidden element
             "Shuriken image width fetched; actual: " +
             withShuriken.offsetWidth + ", expected: 36");
      assert(withShuriken.offsetHeight === 48,
             "Shuriken image height fetched: actual: " +
```

```
        withShuriken.offsetHeight + ", expected 48");
    const dimensions = getDimensions(withShuriken);    ⟵——— Uses new function
    assert(dimensions.width === 36,                     ⟵
           "Shuriken image width fetched; actual: " +    Retests hidden element
           dimensions.width + ", expected: 36");
    assert(dimensions.height === 48,
           "Shuriken image height fetched: actual: " +
           dimensions.height + ", expected 48");
  },3000);
 });
</script>
```

That's a long listing, but most of it is test code; the implementation of the new dimension-fetching function spans only a dozen or so lines of code.

Let's take a look at it piece by piece. First, we set up elements to test: a `<div>` element containing a bunch of text with two images embedded within it, left-justified by styles in an external style sheet. These image elements will be the subjects of our tests; one is visible, and one is hidden.

Prior to running any script, the elements appear as shown in figure 12.4. If the second image weren't hidden, it would appear as a second ninja just to the right of the visible one.

Then we set about defining our new function. We're going to use a hash for some important information, but we don't want to pollute the global namespace with this hash; we want it to be available to the function in its local scope, but no further than that.

We accomplish that by enclosing the hash definition and function declaration within an immediate function, which creates a local scope. The hash isn't accessible outside the immediate function, but the `getDimensions` function that we also define within the immediate function has access to the hash via its closure. Nifty, eh?

Figure 12.4 **We'll use two images—one visible, one hidden—for testing the fetching of dimensions of hidden elements.**

```
(function(){
    const PROPERTIES = {
      position: "absolute",
      visibility: "hidden",
      display: "block"
    };
    window.getDimensions = element => {
      const previous = {};
      for (let key in PROPERTIES) {
        previous[key] = element.style[key];
        element.style[key] = PROPERTIES[key];
      }
```

```
      const result = {
        width: element.offsetWidth,
        height: element.offsetHeight
      };
      for (let key in PROPERTIES) {
        element.style[key] = previous[key];
      }
      return result;
    };
  })();
```

Our new dimension-fetching function is then declared, accepting the element that's to be measured. Within that function, we first create a hash named `previous` in which we'll record the previous values of the style properties that we'll be stomping on, so that we can restore them later. Looping over the replacement properties, we then record each of their previous values and replace those values with the new ones.

That accomplished, we're ready to measure the element, which has now been made part of the display layout, invisible, and absolutely positioned. The dimensions are recorded in a hash assigned to local variable `result`.

Now that we've pilfered what we came for, we erase our tracks by restoring the original values of the style properties that we modified, and we return the results as a hash containing `width` and `height` properties.

All well and good, but does it work? Let's find out.

In a load handler, we perform the tests in a callback to a 3-second timer. Why, you ask? The load handler ensures that we don't perform the test until we know that the DOM has been built, and the timer enables us to watch the display while the test is running, to make sure no display glitches occur while we fiddle with the properties of the hidden element. After all, if the display is disturbed in any way when we run our function, it's a bust.

In the timer callback, we first get a reference to our test subjects (the two images) and assert that we can obtain the dimensions of the visible image by using the offset properties. This test passes, which we can see if we peek ahead to figure 12.5.

Then we make the same test on the hidden element, incorrectly assuming that the offset properties will work with a hidden image. Not surprisingly, because we've already acknowledged that this won't work, the test fails.

Next, we call our new function on the hidden image, and then retest with those results. Success! Our test passes, as shown in figure 12.5.

If we watch the display of the page while the test is running—remember, we delay running the test until 3 seconds after the DOM is loaded—we can see that the display isn't perturbed in any way by our behind-the-scenes adjustments of the hidden element's properties.

> **TIP** Checking the `offsetWidth` and `offsetHeight` style properties for zeroes can serve as an incredibly efficient means of determining the visibility of an element.
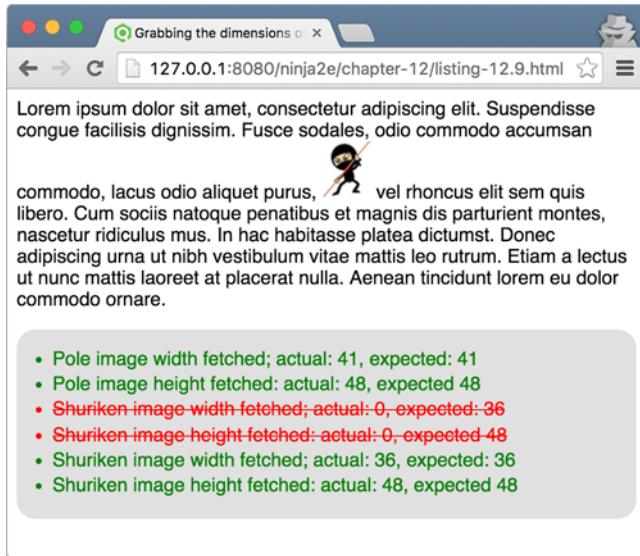
Figure 12.5   **By temporarily adjusting the style properties of hidden elements, we can successfully fetch their dimensions.**

12.4   *Minimizing layout thrashing*

So far in this chapter, you've learned how to relatively easily modify the DOM: by creating and inserting new elements, removing existing elements, or modifying their attributes. Modifying the DOM is one of the fundamental tools for achieving highly dynamic web applications.

But this tool also comes with usage asterisks, one of the most important being, be aware of *layout thrashing*. Layout thrashing occurs when we perform a series of consecutive reads and writes to DOM, in the process not allowing the browser to perform layout optimizations.

Before we delve deeper, consider that changing attributes of one element (or modifying its content) doesn't necessarily affect only that element; instead it can cause a cascade of changes. For example, setting the width of one element can lead to changes in the element's children, siblings, and parents. So whenever a change is made, the browser has to calculate the impact of those changes. In certain cases, there's nothing we can do about it; we need those changes to occur. But at the same time, there's no need to put additional weight on the shoulders of our poor browsers, causing our web application performance to dwindle.

Because recalculating layout is expensive, browsers try to be as lazy as possible, by delaying working with the layout as much as they can; they try to batch as many write operations as possible on the DOM in a queue so that these operations can be executed in one go. Then, when an operation that requires an up-to-date layout comes along, the browser grudgingly obeys, and executes all batched operations and finally updates the layout. But sometimes, the way we'll write our code doesn't give the browser enough room to perform these sorts of optimizations, and we force the

browser to perform a lot of (possibly needless) recalculations. This is what layout thrashing is all about; it occurs when our code performs a series of (often unnecessary) consecutive reads and writes to the DOM, not allowing the browser to optimize layout operations. The problem is that, whenever we modify the DOM, the browser has to recalculate the layout before any layout information is read. This action is expensive, in terms of performance. Let's take a look at an example.

Listing 12.10   Consecutive series of reads and writes causes layout thrashing

```
<div id="ninja">I'm a ninja</div>
<div id="samurai">I'm a samurai</div>          Defines a few HTML elements
<div id="ronin">I'm a ronin</div>
<script>
  const ninja = document.getElementById("ninja");
  const samurai = document.getElementById("samurai");     Fetches the elements
  const ronin = document.getElementById("ronin");         from the DOM

  const ninjaWidth = ninja.clientWidth;
  ninja.style.width = ninjaWidth/2 + "px";

  const samuraiWidth = samurai.clientWidth;       Performs a series of
  samurai.style.width = samuraiWidth/2 + "px";    consecutive reads and
                                                  writes. DOM modifications
  const roninWidth = ronin.clientWidth;           invalidate the layout.
  ronin.style.width = roninWidth/2 + "px";
</script>
```

Reading the value of the element's `clientWidth` property is one of those actions that requires the browser to have an up-to-date layout. By performing consecutive reads and writes to the width property of different elements, we don't allow the browser to be lazy in a smart way. Instead, because we read layout information after every layout modification, the browser has to recalculate the layout every time, just to be sure that we still get the correct information.

One way of minimizing layout thrashing is to write code in a way that doesn't cause needless layout recalculations. For example, we can rewrite listing 12.10 into the following.

Listing 12.11   Batch DOM reads and writes to avoid layout thrashing

```
<div id="ninja">I'm a ninja</div>
<div id="samurai">I'm a samurai</div>
<div id="ronin">I'm a ronin</div>
<script>
  const ninja = document.getElementById("ninja");
  const samurai = document.getElementById("samurai");
  const ronin = document.getElementById("ronin");

  const ninjaWidth = ninja.clientWidth;           Batches all reads to layout
  const samuraiWidth = samurai.clientWidth;       properties together
  const roninWidth = ronin.clientWidth;
```

```
  ninja.style.width = ninjaWidth/2 + "px";
  samurai.style.width = samuraiWidth/2 + "px";
  ronin.style.width = roninWidth/2 + "px";
</script>
```

Batches all writes to layout properties together

Here we batch all reads and writes, because we know that no dependencies exist between the dimensions of our elements; setting the width of the `ninja` element doesn't influence the width of the `samurai` element. This allows the browser to lazily batch operations that modify the DOM.

Layout thrashing isn't something that you'd notice in smaller, simpler pages, but it's something to keep in mind when developing complex web applications, especially on mobile devices. For this reason, it's always good to keep in mind the methods and properties that require an up-to-date layout, shown in the following table (obtained from http://ricostacruz.com/cheatsheets/layout-thrashing.html).

Table 12.2 APIs and properties that cause layout invalidation

Interface	Property name
Element	`clientHeight, clientLeft, clientTop, clientWidth, focus, getBoundingClientRect, getClientRects, innerText, offsetHeight, offsetLeft, offsetParent, offsetTop, offsetWidth, outerText, scrollByLines, scrollByPages, scrollHeight, scrollIntoView, scrollIntoViewIfNeeded, scrollLeft, scrollTop, scrollWidth`
MouseEvent	`layerX, layerY, offsetX, offsetY`
Window	`getComputedStyle, scrollBy, scrollTo, scroll, scrollY`
Frame, Document, Image	`height, width`

Several libraries that try to minimize layout thrashing have been developed. One of the more popular ones is FastDom (https://github.com/wilsonpage/fastdom). The library repository includes examples that clearly show the performance gains that can be achieved by batching DOM read/write operations (https://wilsonpage.github.io/fastdom/examples/aspect-ratio.html).

React's virtual DOM

One of the most popular client-side libraries is Facebook's React (https://facebook.github.io/react/). React achieves great performance by using a virtual DOM, a set of JavaScript objects that mimic the actual DOM. When we develop applications in React, we perform all modifications on the virtual DOM, without any regard for layout thrashing. Then, at an appropriate time, React uses the virtual DOM to figure out what changes have to be made to the actual DOM, in order to keep the UI in sync. This batching of updates increases the performance of applications.

12.5   Summary

- Converting an HTML string into DOM elements includes the following steps:
  - Making sure that the HTML string is valid HTML code
  - Wrapping it into enclosing markup, required by browser rules
  - Inserting the HTML into a dummy DOM element through the `innerHTML` property of a DOM element
  - Extracting the created DOM nodes back out
- For fast inserting of DOM nodes, use DOM fragments, because a fragment can be injected in a single operation, thereby drastically reducing the number of operations.
- DOM element attributes and properties, although linked, aren't always identical! We can read and write to DOM attributes by using the `getAttribute` and `setAttribute` methods, whereas we write to DOM properties by using object property notation.
- When working with attributes and properties, we have to be aware of *custom attributes.* Attributes that we decide to place on HTML elements in order to carry information useful to our applications aren't automatically presented as element properties.
- The `style` element property is an object that holds properties corresponding to the style values specified in the element markup. To get the computed styles, which also take into account the styles set in style sheets, use the built-in `get-ComputedStyle` method.
- For getting the dimensions of HTML elements, use `offsetWidth` and `offsetHeight` properties.
- Layout thrashing occurs when code performs a series of consecutive reads and writes to DOM, each time forcing the browser to recalculate the layout information. This leads to slower, less responsive web applications.
- Batch your DOM updates!

12.6   Exercises

1 In the following code, which of the following assertions will pass?

```
<div id="samurai"></div>
<script>
  const element = document.querySelector("#samurai");

  assert(element.id === "samurai", "property id is samurai");
  assert(element.getAttribute("id") === "samurai",
         "attribute id is samurai");

  element.id = "newSamurai";
```

```
    assert(element.id === "newSamurai", "property id is newSamurai");
    assert(element.getAttribute("id") === "newSamurai",
           "attribute id is newSamurai");
</script>
```

2 Given the following code, how can we access the element's `border-width` style property?

```
<div id="element" style="border-width: 1px;
                         border-style:solid; border-color: red">
</div>
<script>
  const element = document.querySelector("#element");
</script>
```

a `element.border-width`

b `element.getAttribute("border-width");`

c `element.style["border-width"];`

d `element.style.borderWidth;`

3 Which built-in method can get all styles applied to a certain element (styles provided by the browser, styles applied via style sheets, and properties set through the style attribute)?

a `getStyle`

b `getAllStyles`

c `getComputedStyle`

4 When does layout thrashing occur?

Surviving events

Chapter 2 included a short discussion on the JavaScript single-threaded execution model and introduced the event loop and the event queue, in which events wait for their turn to be processed. This discussion was particularly useful when presenting the steps in the lifecycle of a web page, especially when discussing the order in which certain pieces of JavaScript code get executed. At the same time, it's a simplification, so in order to get a more complete picture of how the browser works, we'll spend a significant part of this chapter exploring the nooks and crannies of the event loop. This will help us better understand some of the performance limitations inherent in JavaScript and the browser. In turn, we'll use this knowledge to develop smoother-running applications.

During this exploration, we'll put a special focus on timers, a JavaScript feature that enables us to delay the execution of a piece of code asynchronously by a certain amount of time. At first glance, this might not seem like much, but we'll show you how to use timers to break up long-running tasks that make applications slow and unresponsive into smaller tasks that don't clog the browser. This helps develop better-performing applications.

We'll continue this exploration of events by showing how events are propagated through the DOM tree, and how to use this knowledge to write simpler and less memory-intensive code. Finally, we'll finish the chapter with creating custom events, which can help reduce coupling between different parts of the application. Without further ado, let's start looping through the event loop.

. . . .

Do you know?	**Why is the timing on timer callbacks not guaranteed?**
	If a `setInterval` timer fires every 3 milliseconds while another event handler is running for 16 ms, how many times will the timer's callback function be added to the microtask queue?
	Why is the function context for an event handler some-times different from the event's target?

. . . .

13.1 *Diving into the event loop*

As you might have figured out, the event loop is more complicated than its presentation in chapter 2. For starters, instead of a single event queue, which holds only events, the event loop has at least two queues that, in addition to events, hold other actions performed by the browser. These actions are called *tasks* and are grouped into two categories: *macrotasks* (or often just called tasks) and *microtasks*.

Examples of macrotasks include creating the main document object, parsing HTML, executing mainline (or global) JavaScript code, changing the current URL, as well as various events such as page loading, input, network events, and timer events. From the browser's perspective, a macrotask represents one discrete, self-contained unit of work. After running a task, the browser can continue with other assignments such as re-rendering the UI of the page, or performing garbage collection.

Microtasks, on the other hand, are smaller tasks that update the application state and should be executed before the browser continues with other assignments such as re-rendering the UI. Examples include promise callbacks and DOM mutation changes. Microtasks should be executed as soon as possible, in an asynchronous way, but without the cost of executing a whole new macrotask. Microtasks enable us to execute certain actions *before* the UI is re-rendered, thereby avoiding unnecessary UI rendering that might show inconsistent application state.

> **NOTE**   The ECMAScript specification doesn't mention event loops. Instead, the event loop is detailed in the HTML specification (https://html.spec.whatwg.org/#event-loops), which also discusses the concept of macrotasks and microtasks. The ECMAScript specification mentions *jobs* (which are analogous to microtasks) in respect to handling promise callbacks (http://mng.bz/fOlK). Even though the event loop is defined in the HTML specification, other environments, such as Node.js, also use it.

The implementation of an event loop *should* use at least one queue for macrotasks and at least one queue for microtasks. Event loop implementations usually go beyond that, and have *several* queues for different types of macro- and microtasks. This enables the event loop to prioritize types of tasks; for example, giving priority to performance-sensitive tasks such as user input. On the other hand, because there are many browsers and JavaScript execution environments out in the wild, you shouldn't be surprised if you run into event loops with only a single queue for both types of tasks together.

The event loop is based on two fundamental principles:

- Tasks are handled one at a time.
- A task runs to completion and can't be interrupted by another task.

Let's take a look at figure 13.1, which depicts these two principles.
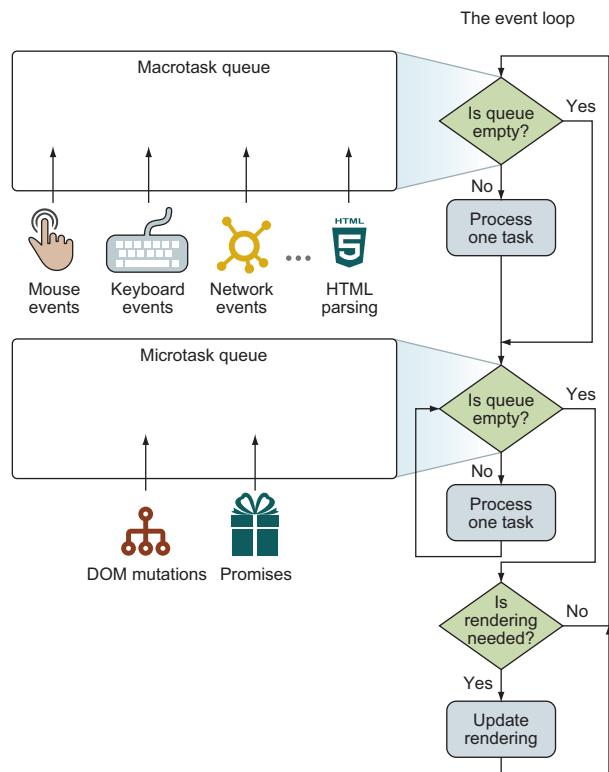
Figure 13.1   The event loop usually has access to at least two task queues: a microtask queue and a macrotask queue. Both types of tasks are handled one at a time.

On a high level, figure 13.1 shows that in a single iteration, the event loop first checks the macrotask queue, and if there's a macrotask waiting to be executed, starts its execution. Only after the task is fully processed (or if there were no tasks in the queue), the event loop moves onto processing the microtask queue. If there's a task waiting in that queue, the event loop takes it and executes it to completion. This is performed for all microtasks in the queue. Note the difference between handling the macrotask and microtask queues: In a single loop iteration, one macrotask at most is processed (others are left waiting in the queue), whereas all microtasks are processed.

When the microtask queue is finally empty, the event loop checks whether a UI render update is required, and if it is, the UI is re-rendered. This ends the current iteration of the event loop, which goes back to the beginning and checks the macrotask queue again.

Now that we have a high-level understanding of the event loop, let's check some of the interesting details shown in figure 13.1:

- *Both task queues are placed outside the event loop*, to indicate that the act of adding tasks to their matching queues happens outside the event loop. If this wasn't the case, any events that occur while JavaScript code is being executed would be ignored. Because we most definitely don't want to do this, the acts of detecting and adding tasks are done separately from the event loop.

- *Both types of tasks are executed one at a time*, because JavaScript is based on a single-threaded execution model. When a task starts executing, it's executed to its completion, without being interrupted by another task. Only the browser can stop the execution of a task; for example, if the task starts being too selfish by taking up too much time or memory.

- *All microtasks should be executed before the next rendering*, because their goal is to update the application state before rendering occurs.

- *The browser usually tries to render the page 60 times per second*, to achieve 60 frames per second (60 fps), a frame rate that's often considered ideal for smooth motion, such as animations—*meaning, the browser tries to render a frame every 16 ms.* Notice how the "Update rendering" action, shown in figure 13.1, happens inside the event loop, because the page content shouldn't be modified by another task while the page is being rendered. This all means that, if we want to achieve smooth-running applications, we don't have much time to process tasks in a single event-loop iteration. *A single task and all microtasks generated by that task should ideally complete within 16 ms.*

Now, let's consider three situations that can occur in the next event-loop iteration, after the browser has completed a page render:

- The event loop reaches the "Is rendering required?" decision point before another 16 ms has elapsed. Because updating the UI is a complex operation, if there isn't an explicit need to render the page, the browser may choose not to perform the UI rendering in this loop iteration.

- The event loop reaches the "Is rendering required?" decision point roughly around 16 ms after the last rendering. In this case, the browser updates the UI, and users will experience a smooth-running application.
- Executing the next task (and all related microtasks) takes much more than 16 ms. In this case, the browser won't be able to re-render the page at the target frame rate, and the UI won't be updated. If running the task code doesn't take up too much time (more than a couple of hundred milliseconds), this delay might not even be perceivable, especially if there isn't much motion going on in the page. On the other hand, if we take too much time, or animations are running on the page, users will probably perceive the web page as slow and nonresponsive. In a worst-case scenario, in which a task gets executed for more than a couple of seconds, the user's browser shows the dreaded "Unresponsive script" message. (Don't worry, later in the chapter we'll show you a technique for breaking complex tasks into smaller ones that won't clog the event loop.)

> **NOTE** Be careful about which events you decide to handle, how often they occur, and how much processing time an event handler takes. For example, you should be extra careful when handling mouse-move events. Moving the mouse around causes a large number of events to be queued, so performing any complex operation in that mouse-move handler is a sure path to building a slow and jerky web application.

Now that we've described how the event loop works, you're ready to explore a couple of examples in detail.

13.1.1 An example with only macrotasks

The unavoidable result of JavaScript's single-threaded execution model is that only one task can be executed at a time. This in turn means that all created tasks have to wait in a queue until their turn for execution comes.

Let's focus our attention on a simple web page that contains the following:

- Nontrivial mainline (global) JavaScript code
- Two buttons and two nontrivial click handlers, one for each button

The following listing shows the sample code.

Listing 13.1   Pseudocode for our event loop demo with one task queue

```
<button id="firstButton"></button>
<button id="secondButton"></button>
<script>
  const firstButton = document.getElementById("firstButton");
  const secondButton = document.getElementById("secondButton");
  firstButton.addEventListener("click", function firstHandler(){
   /*Some click handle code that runs for 8 ms*/
  });
```

Registers an event handler for a button-click event on the first button

```
secondButton.addEventListener("click", function secondHandler(){
  /*Click handle code that runs for 5ms*/
});
/*Code that runs for 15ms*/
</script>
```

Registers another click event handler, this time for the second button

This example requires some imagination, so instead of cluttering the code fragment with unnecessary code, we ask you to imagine the following:

- Our mainline JavaScript code takes 15 ms to execute.
- The first click event handler runs for 8 ms.
- The second click event handler runs for 5 ms.

Now let's continue to be imaginative, and say that we have a super quick user who clicks the first button 5 ms after our script starts executing, and the second button 12 ms after. Figure 13.2 depicts this situation.

There's a lot of information to digest here, but understanding it completely will give you a better idea of how the event loop works. In the top part of the figure, time
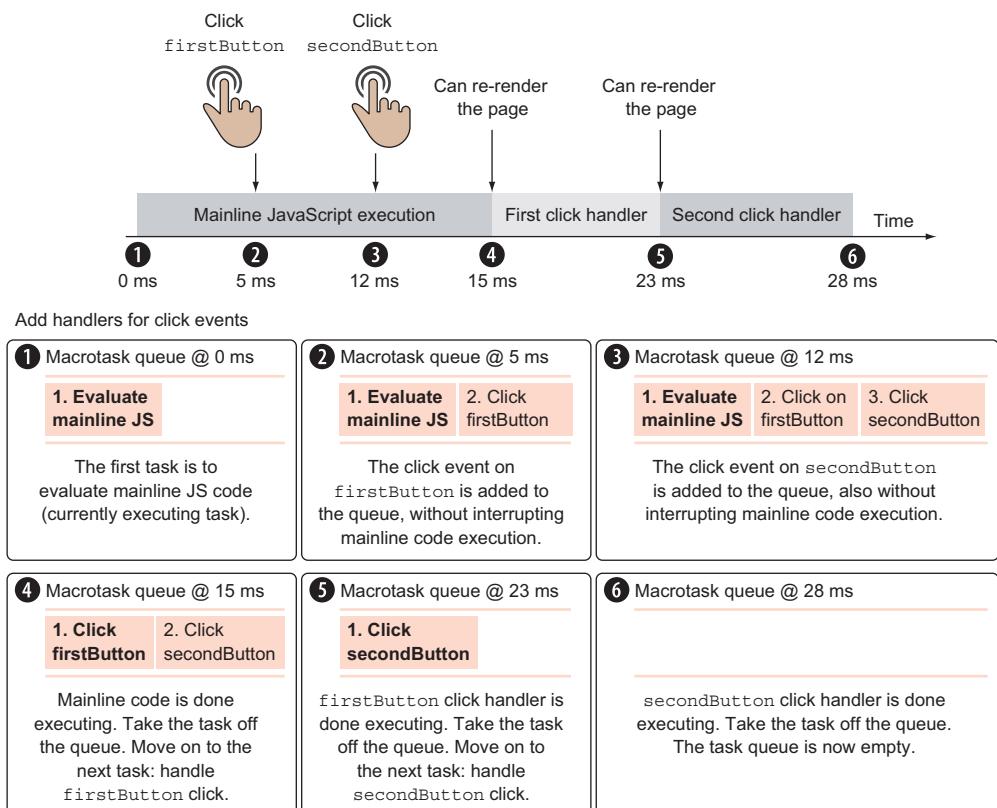
Figure 13.2 This timing diagram shows how events are added to the task queue as they occur. When a task is done executing, the event loop takes it off the queue, and continues by executing the next task.

(in milliseconds) is running from left to right along the x-axis. The rectangles underneath that timeline represent portions of JavaScript code under execution, extending for the amount of time they're running. For example, the first block of mainline JavaScript code executes for approximately 15 ms, the first click handler for approximately 8 ms, and the second click handler for approximately 5 ms. The timing diagram also shows when certain events occur; for example, the first button click occurs 5 ms into application execution, and the second button click at 12 ms. The bottom part of the figure shows the state of the macrotask queue at various points of application execution.

The program starts by executing mainline JavaScript code. Immediately, two elements, `firstButton` and `secondButton`, are fetched from the DOM, and two functions, `firstHandler` and `secondHandler`, are registered as click event handlers:

```
firstButton.addEventListener("click", function firstHandler(){...});
secondButton.addEventListener("click", function secondHandler(){...});
```

This is followed by code that executes for another 15 ms. During this execution, our quick user clicks `firstButton` 5 ms after the program starts executing, and clicks `secondButton` 12 ms after.

Because JavaScript is based on a single-threaded execution model, clicking `firstButton` doesn't mean that the click handler is immediately executed. (Remember, if a task is already being executed, it can't be interrupted by another task.) Instead, the click event related to `firstButton` is placed in the task queue, where it patiently waits for its turn to be executed. The same thing happens when a click of `secondButton` occurs: A matching event is placed in the task queue, and waits for execution. Note that it's important that the event detection and addition to the task queue happens outside the event loop; the tasks are added to the task queue even while mainline JavaScript code is being executed.

If we take a snapshot of our task queue 12 ms into the execution of our script, we'll see the following three tasks:

1. Evaluate mainline JavaScript code—the currently executing task.
2. Click `firstButton`—the event created when `firstButton` is clicked.
3. Click `secondButton`—the event created when `secondButton` is clicked.

These tasks are also shown in figure 13.3.

The next interesting point in the application execution happens at 15 ms, when the mainline JavaScript code finishes its
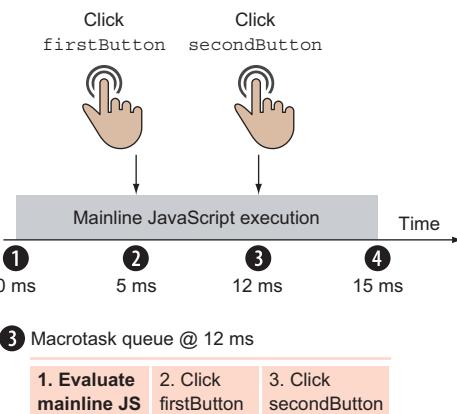
Figure 13.3 **12 ms into the application execution, the task queue has three tasks: one for evaluating mainline JavaScript code (the currently executing task), and one for each button-click event.**
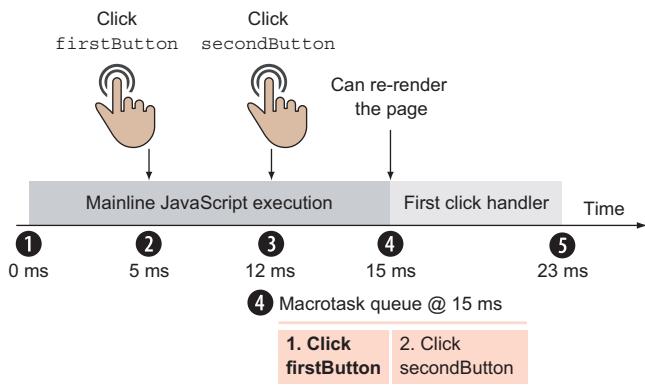
Figure 13.4 The task queue 15 ms into application execution contains two tasks for click events. The first task is currently being executed.

execution. As shown in figure 13.1, after a task has finished the execution, the event loop moves on to processing the microtask queue. Because in this situation we don't have any microtasks (we don't even show the microtask queue in the diagram, because it's always empty), we skip this step and move on to updating the UI. In this example, even though the update happens and takes some time, for simplicity sake, we keep it out of our discussion. With this, the event loop finishes the first iteration and starts the second iteration, by moving onto the following task in the queue.

Next, the firstButton click task starts its execution. Figure 13.4 illustrates the task queue 15 ms into the application execution. The execution of firstHandler, associated with the firstButton click, takes around 8 ms, and the handler is executed to its completion, without interruption, while the click event related to secondButton is waiting in the queue.

Next, at 23 ms, the firstButton click event is fully processed, and the matching task is removed from the task queue. Again, the browser checks the microtask queue, which is still empty, and re-renders the page, if necessary.

Finally, in the third loop iteration, the secondButton click event is being handled, as shown in figure 13.5. The secondHandler takes around 5 ms to execute, and after this is performed, the task queue is finally empty, at 28 ms.

This example emphasizes that an event has to wait its turn to be processed, if other tasks are already being handled. For example, even though the secondButton click has happened 12 ms into the application execution, the matching handler is called somewhere around 23 ms into the application execution.

Now let's extend this code to include microtasks.

13.1.2 An example with both macro- and microtasks

Now that you've seen how the event loop works against one task queue, we'll extend our example to also include a microtask queue. The cleanest way to do this is to include a promise in the first button-click handler and the code that handles the promise after it resolves. As you'll recall from chapter 6, *a promise is a placeholder for a value that we don't have yet but will have later;* it's a guarantee that we'll eventually know
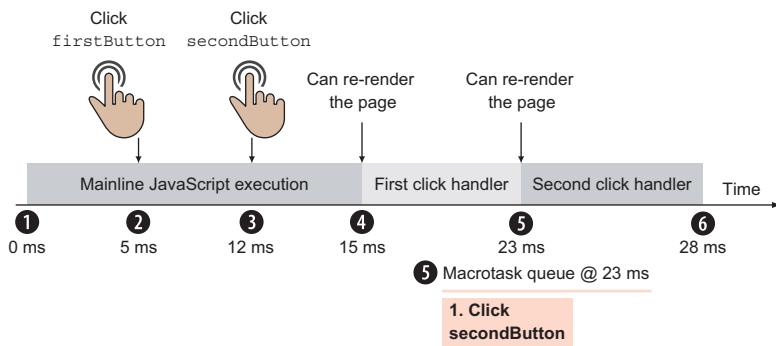
Figure 13.5   23 ms after the application starts executing, only one task, handling the `secondButton` click event, remains to be executed.

the result of an asynchronous computation. For this reason, promise handlers, the callbacks we attach through the promise's `then` method, are always called asynchronously, even if we attach them to already resolved promises.

The following listing shows the modified code for this two-queue example.

Listing 13.2   Pseudocode for our event loop demo with two queues

```
<button id="firstButton"></button>
<button id="secondButton"></button>
<script>
  const firstButton = document.getElementById("firstButton");
  const secondButton = document.getElementById("secondButton");
  firstButton.addEventListener("click", function firstHandler(){
    Promise.resolve().then(() => {
      /*Some promise handling code that runs for 4 ms*/
    });

    /*Some click handle code that runs for 8 ms*/
  });

  secondButton.addEventListener("click", function secondHandler(){
    /*Click handle code that runs for 5ms*/
  });
/*Code that runs for 15ms*/
</script>
```

> Immediately resolves a promise and passes in a callback to the then method

In this example, we assume that the same actions occur as in the first example:

- `firstButton` is clicked after 5 ms.
- `secondButton` is clicked after 12 ms.
- `firstHandler` handles the click event of `firstButton` and runs for 8 ms.
- `secondHandler` handles the click event of `secondButton` and runs for 5 ms.

The only difference is that this time, within the `firstHandler` code, we also create an immediately resolved promise to which we pass a callback that will run for 4 ms. Because a promise represents a future value that we usually don't know immediately, promise handlers are always handled asynchronously.

To be honest, in this case, where we've created an immediately resolved promise, the JavaScript engine could immediately invoke the callback, because we already know that the promise is successfully resolved. But, for consistency sake, the JavaScript engine doesn't do this, and instead calls all promise callbacks asynchronously, after the rest of the `firstHandler` code (which runs for 8 ms) is done executing. It does this by creating a new microtask and pushing it onto the microtask queue. Let's explore the timing diagram of this execution in figure 13.6.
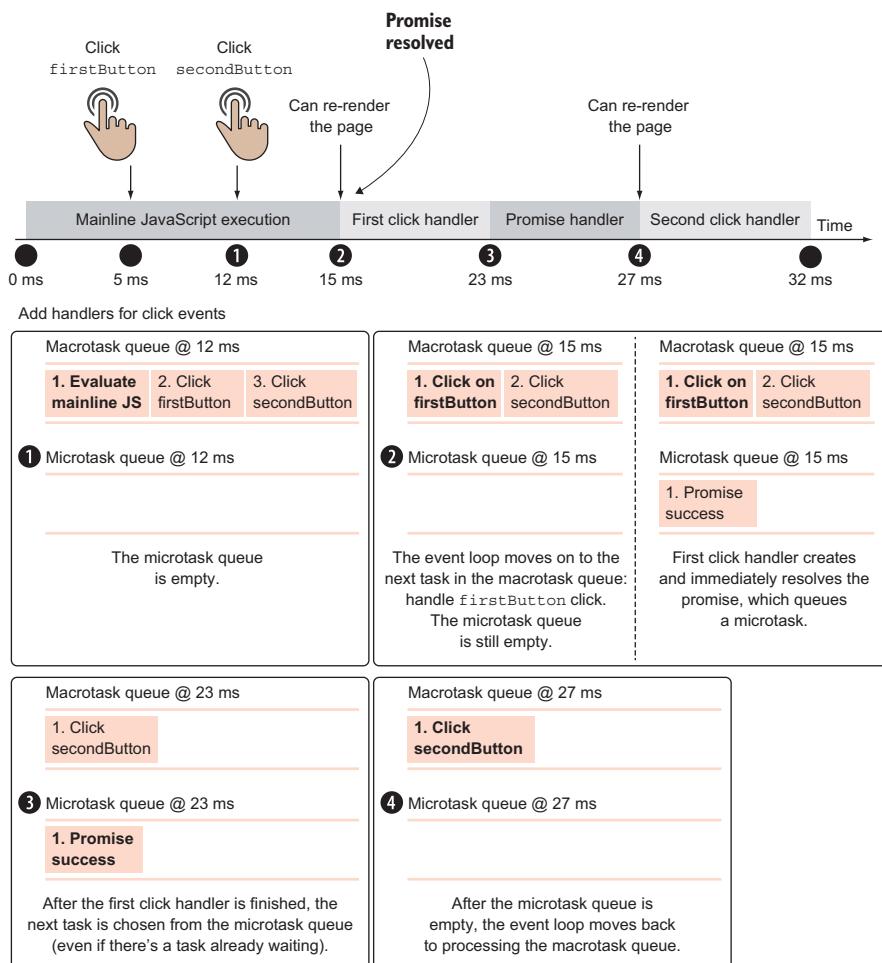
Figure 13.6 **If a microtask is queued in the microtask queue, it gets priority and is processed even if an older task is already waiting in the queue. In this case, the promise success microtask gets priority over the `secondButton` click event task.**

This timing diagram is similar to the diagram of the previous example. If we take a snapshot of the task queue 12 ms into the application execution, we'll see the exact same tasks in the queue: The mainline JavaScript code is being processed while the tasks for handling the firstButton click and the secondButton click are waiting for their turns (just as in figure 13.3). But in addition to the task queue, in this example we'll also focus on the microtask queue, which is still empty 12 ms into application execution.
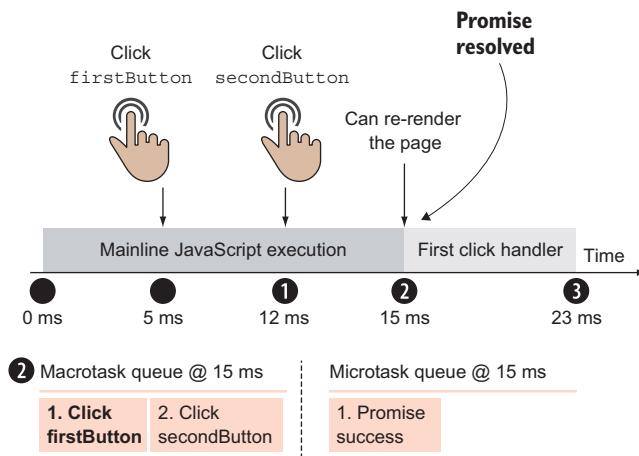
The next interesting point in application execution happens at 15 ms, when mainline JavaScript execution ends. Because a task has finished executing, the event loop checks the microtask queue, which is empty, and moves onto page rendering, if necessary. Again, for simplicity sake, we don't include a rendering fragment in our timing diagram.

In the next iteration of the event loop, the task associated with the firstButton click is being processed:

```
firstButton.addEventListener("click", function firstHandler(){
    Promise.resolve().then(() => {
      /*Some promise handling code that runs for 4ms*/
    });
    /*Some click handle code that runs for 8ms*/
  });
```

The firstHandler function creates an already resolved promise, by calling Promise .resolve() with a callback function that will for sure be invoked, because the promise is already resolved. This creates a new microtask to run the callback code. The microtask is placed into the microtask queue, and the click handler continues to execute for another 8 ms. The current state of the task queues is shown in figure 13.7.

We revisit the task queues again 23 ms into the application execution, after the firstButton click has been completely handled and its task taken off the task queue.

Figure 13.7   **During the execution of the first click handler, a resolved promise is created. This queues up a promise success microtask in the microtask queue that will be executed as soon as possible, but without interrupting the currently running task.**

At this point, the event loop has to choose which task to process next. We have one macrotask for handling the `secondButton` click that was placed in the task queue 12 ms into application execution, and one microtask for handling the promise success that was placed in the microtask queue somewhere around 15 ms into the application execution.

   If we consider things like this, it would be only fair that the `secondButton` click task gets handled first, but as we already mentioned, microtasks are smaller tasks that should be executed as soon as possible. Microtasks have priority, and if you look back at figure 13.1, you'll see that every time a task is processed, the event loop first checks the microtask queue, with the goal of processing all microtasks before continuing on to either rendering or other tasks.

   For this reason, the promise success task is executed *immediately* after the first-Button click, even with the "older" `secondButton` click task still waiting in the task queue, as shown in figure 13.8.
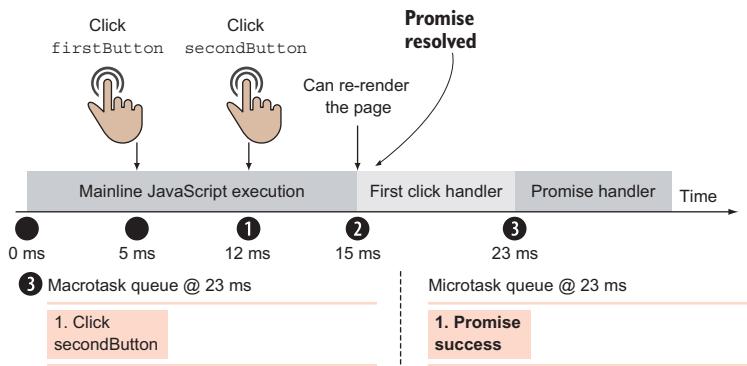
Figure 13.8   **After a task gets executed, the event loop processes all tasks in the microtask queue. In this case, before moving to the `secondButton` click task, the promise success task is handled.**

There's one important point that we need to emphasize. After a macrotask gets executed, the event loop immediately moves onto handling the microtask queue, without allowing rendering until the microtask queue is empty. Just take a look at the timing diagram in figure 13.9.
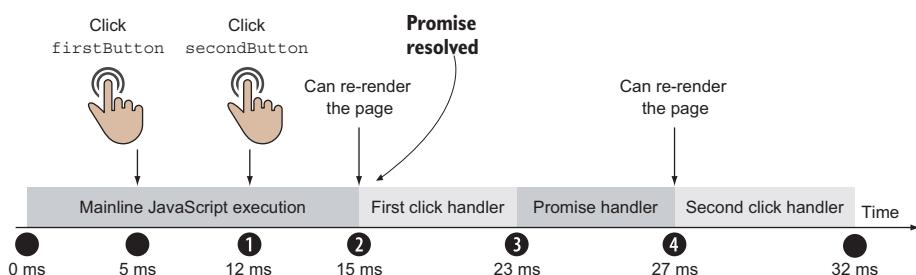
Figure 13.9   **A page can be re-rendered between two macrotasks (mainline JavaScript execution and first click handler), while it *can't* be rendered before a microtask is executed (before the promise handler).**

Figure 13.9 shows that a re-render can occur between two macrotasks, only if there are no microtasks in between. In our case, the page can be rendered between the main-line JavaScript execution and the first click handler, but it can't be rendered immediately after the first click handler, because microtasks, such as promise handlers, have priority over rendering.

A render can also occur after a microtask, but only if no other microtasks are waiting in the microtask queue. In our example, after the promise handler occurs, but before the event loop moves onto the second click handler, the browser can re-render the page.

Note that there's nothing stopping the promise success microtask from queuing other microtasks, and that all of these microtasks will have priority over the second-Button click task. The event loop will re-render the page and move onto the second-Button task only when the microtask queue is empty, so be careful!

Now that you understand how the event loop works, let's take a look at a special group of events: timers.

13.2   *Taming timers: time-outs and intervals*

Timers, an often misused and poorly understood feature in JavaScript, can enhance the development of complex applications if used properly. Timers enable us to delay the execution of a piece of code by *at least* a certain number of milliseconds. We'll use this capability to break long-running tasks into smaller tasks that won't clog the event loop, thereby stopping the browser from rendering, and in the process making our web applications slow and unresponsive.

But first, we'll start by examining the functions we can use to construct and manipulate timers. The browser provides two methods for creating timers: `setTimeout` and `setInterval`. The browser also provides two corresponding methods to clear (or remove) them: `clearTimeout` and `clearInterval`. All are methods of the `window` (global context) object. Similarly to the event loop, timers aren't defined within JavaScript itself; instead they're provided by the host environment (such as the browser on the client, or Node.js on the server). Table 13.1 lists the methods for creating and clearing timers.

Table 13.1   **JavaScript's timer-manipulation methods (methods of the global `window` object)**

Method	Format	Description
setTimeout	id = setTimeout(fn,delay)	Initiates a timer that will execute the passed callback once after the specified delay has elapsed. A value that uniquely identifies the timer is returned.
clearTimeout	clearTimeout(id)	Cancels (clears) the timer identified by the passed value if the timer hasn't yet fired.

Table 13.1 JavaScript's timer-manipulation methods (methods of the global `window` object) *(continued)*

Method	Format	Description
`setInterval`	`id = setInterval(fn,delay)`	Initiates a timer that will continually try to execute the passed callback at the specified delay interval, until canceled. A value that uniquely identifies the timer is returned.
`clearInterval`	`clearInterval(id)`	Cancels (clears) the interval timer identified by the passed value.

These methods allow us to set and clear timers that either fire a single time or fire periodically at a specified interval. In practice, most browsers allow you to use both `clearTimeout` and `clearInterval` to cancel both kinds of timers, but it's recommended that the methods be used in matched pairs, if for nothing other than clarity.

> **NOTE** It's important to understand that *a timer's delay isn't guaranteed*. This has a great deal to do with the event loop, as we'll see in the next section.

13.2.1 *Timer execution within the event loop*

You've already examined exactly what happens when an event occurs. But timers are different from standard events, so let's explore an example similar to the ones you've seen so far. The following listing shows the code used for this example.

Listing 13.3   Pseudocode for our time-out and interval demo

```
<button id="myButton"></button>
<script>
  setTimeout(function timeoutHandler(){
    /*Some timeout handle code that runs for 6ms*/
  }, 10);

  setInterval(function intervalHandler(){
    /*Some interval handle code that runs for 8ms*/
  }, 10);

  const myButton = document.getElementById("myButton");
  myButton.addEventListener("click", function clickHandler(){
    /*Some click handle code that runs for 10ms*/
  });

  /*Code that runs for 18ms*/
</script>
```

Registers a time-out that expires after 10 ms

Registers an interval that expires every 10 ms

Registers an event handler for a button-click event

This time we have only one button, but we also register two timers. First, we register a time-out that expires after 10 ms:

```
setTimeout(function timeoutHandler(){
  /*Some timeout handler code that runs for 6ms*/
}, 10);
```

As a handler, that time-out has a function that takes 6 ms to execute. Next, we also register an interval that expires every 10 ms:

```
setInterval(function intervalHandler(){
  /*Some interval handler code that runs for 8ms*/
}, 10);
```

The interval has a handler that takes 8 ms to execute. We continue by registering a button-click event handler that takes 10 ms to execute:

```
const myButton = document.getElementById("myButton");
myButton.addEventListener("click", function clickHandler(){
   /*Some click handler code that runs for 10ms*/
});
```

This example ends with a code block that runs for about 18 ms (again, humor us a bit and imagine some complex code here).

Now, let's say we again have a quick and impatient user who clicks the button 6 ms into the application execution. Figure 13.10 shows a timing diagram of the first 18 ms of execution.
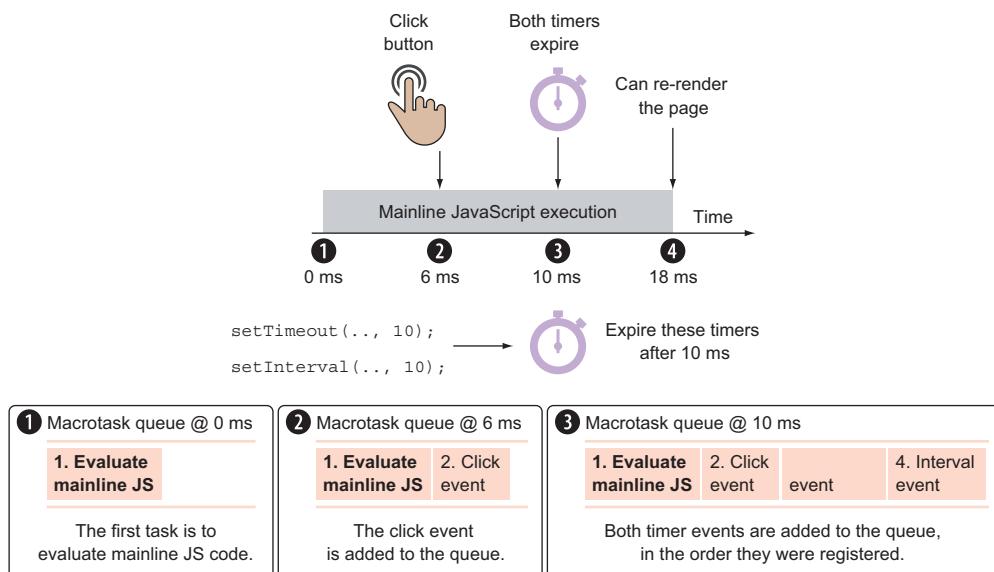
Figure 13.10   A timing diagram that shows 18 ms of execution in the example program. The first, currently running task is to evaluate mainline JavaScript code. It takes 18 ms to execute. During that execution, three events occur: a mouse click, a timer expiry, and an interval firing.

As in the previous examples, the first task in the queue is to execute mainline JavaScript code. During that execution, which roughly takes 18 ms to complete, three important things occur:

1. At 0 ms, a time-out timer is initiated with a 10 ms delay, and an interval timer is also initiated with a 10 ms delay. Their references are kept by the browser.
2. At 6 ms, the mouse is clicked.
3. At 10 ms, the time-out timer expires and the first interval fires.

As we already know from our event-loop exploration, a task always runs to completion and can't be interrupted by another task. Instead, all newly created tasks are placed in a queue, where they patiently wait their turn to be processed. When the user clicks the button 6 ms into application execution, that task is added to the task queue. A similar thing happens at around 10 ms, when the timer expires and the interval fires. Timer events, just like input events (such as mouse events), are placed in the task queue. Note that both the timer and interval are initiated with a 10 ms delay, and that after this period, their matching tasks are placed in the task queue. We'll come back to this later, but for now it's enough that you notice that the tasks are added to the queue in the order in which the handlers are registered: first the time-out handler and then the interval handler.

The initial block of code completes executing after 18 ms, and because there are no microtasks in this execution, the browser can re-render the page (again, left out from our timing discussions, due to simplicity) and move onto the second iteration of the event loop. The state of the task queue at this time is shown in figure 13.11.

Figure 13.11 Timer events are placed into the task queue as they expire.

When the initial block of code ends execution at 18 ms, three code blocks are queued up for execution: the click handler, the time-out handler, and the first invocation of the interval handler. This means that the waiting click handler (which we assume takes 10 ms to execute) begins execution. Figure 13.12 shows another timing diagram.

Unlike the `setTimeout` function, which expires only once, the `setInterval` function fires until we explicitly clear it. So, at around 20 ms, another interval fires. Normally, this would create a new task and add it to the task queue. But this time, because an instance of an interval task is already queued and awaiting execution, this invocation is dropped. *The browser won't queue up more than one instance of a specific interval handler.*

The click handler completes at 28 ms, and the browser is again allowed to re-render the page before the event loop goes into another iteration. In the next iteration of the event loop, at 28 ms, the time-out task is processed. But think back to the beginning of

Figure 13.12 If an interval event fires, and a task is already associated with that interval waiting in the queue, a new task isn't added. Instead, nothing happens, as is shown for the queues at 20 ms and 30 ms.

this example. We used the following function call to set a time-out that should expire after 10 ms:

```
setTimeout(function timeoutHandler(){
  /*Some timeout handle code that runs for 6ms*/
}, 10);
```

Because this is the first task in our application, it's not weird to expect that the time-out handler will be executed exactly after 10 ms. But as you see in figure 13.11, the time-out starts at the 28 ms mark!

That's why we were extra careful when saying that a timer provides the capability to asynchronously delay the execution of a piece of code by *at least* a certain number of milliseconds. Because of the single-threaded nature of JavaScript, we can control only when the timer task is added to the queue, and not when it's finally executed! Now that we cleared up this little conundrum, let's continue with the remainder of the application execution.

The time-out task takes 6 ms to execute, so it should be finished 34 ms into the application execution. During this time period, at 30 ms another interval fires, because we've scheduled it to be executed every 10 ms. Once more, no additional task is queued, because a matching task for interval handler execution is already waiting in the queue. At 34 ms, the time-out handler finishes, and the browser again has a chance to re-render the page and enter another iteration of the event loop.

Finally, the interval handler starts its execution at 34 ms, 24 ms *after* the 10 ms mark at which it was added to the event queue. This again emphasizes that the delay we pass in as an argument to the functions `setTimeout(fn, delay)` and `setInterval(fn, delay)` specifies only the delay after which the matching task is added to the queue, and not the exact time of execution.

The interval handler takes 8 ms to execute, so while it's executing, another interval expires at the 40 ms mark. This time, because the interval handler is being executed (and not waiting in the queue), a new interval task is finally added to the task queue, and the execution of our application continues, as shown in figure 13.13. Setting a `setInterval` delay to 10 ms doesn't mean that we'll end up with our handler executing every 10 ms. For example, because tasks are queued and the duration of a single task execution can vary, intervals can be executed one right after another, as is the case with intervals at the 42 and 50 ms marks.

Finally, after 50 ms our intervals stabilize and are executed every 10 ms. The important concept to take away is that the event loop can process only one task at a time, and that we can never be certain that timer handlers will execute exactly when we expect them to. This is especially true of interval handlers. We saw in this example that even though we scheduled an interval expected to fire at 10, 20, 30, 40, 50, 60, and 70 ms marks, the callbacks were executed at 34, 42, 50, 60, and 70 ms marks. In this case, we completely lost two of them along the way, and some weren't executed at the expected time.

As we can see, intervals have special considerations that don't apply to time-outs. Let's look at those more closely.
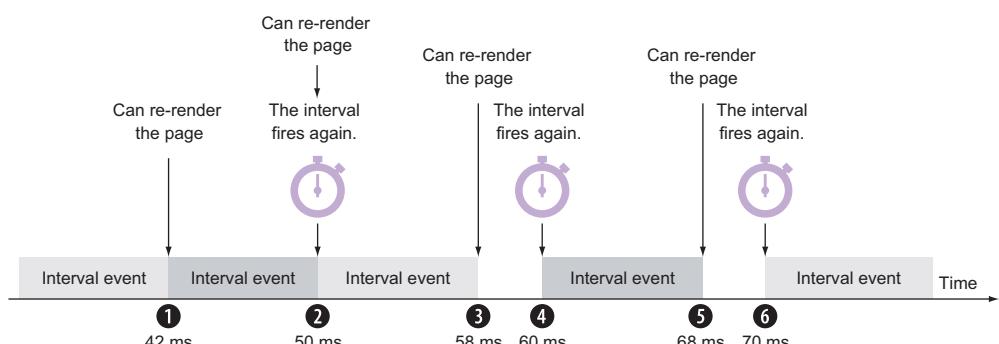
Figure 13.13 **Because of the setbacks caused by the mouse click and time-out handler, it takes some time for the interval handlers to start executing every 10 ms.**

DIFFERENCES BETWEEN TIME-OUTS AND INTERVALS

At first glance, an interval may look like a time-out that periodically repeats itself. But the differences run a little deeper. Let's look at an example to better illustrate the differences between `setTimeout` and `setInterval`:

```
setTimeout(function repeatMe(){
  /* Some long block of code... */
  setTimeout(repeatMe, 10);
}, 10);
setInterval(() => {
  /* Some long block of code... */
}, 10);
```

Sets up a time-out that reschedules itself every 10 milliseconds

Sets up an interval that triggers every 10 milliseconds

The two pieces of code may *appear* to be functionally equivalent, but they aren't. Notably, the `setTimeout` variant of the code will always have at least a 10 ms delay after the previous callback execution (depending on the state of the event queue, it may end up being more, but never less), whereas `setInterval` will attempt to execute a callback every 10 ms regardless of when the last callback was executed. And, as you saw in the example from the previous section, intervals can be fired immediately one after another, regardless of the delay.

We know that the time-out callback is never guaranteed to execute exactly when it's fired. Rather than being fired every 10 ms, as the interval is, it will reschedule itself for 10 ms after it gets around to executing.

All of this is incredibly important knowledge. Knowing how a JavaScript engine handles asynchronous code, especially with the large number of asynchronous events that typically occur in the average page, creates a great foundation for building advanced pieces of application code.

With all that under our belts, let's see how our understanding of timers and the event loop can help avoid some performance pitfalls.

13.2.2 *Dealing with computationally expensive processing*

The single-threaded nature of JavaScript is probably the largest gotcha in complex JavaScript application development. While JavaScript is busy executing, user interaction in the browser can become, at best, sluggish, and, at worst, unresponsive. The browser may stutter or seem to hang, because all updates to the rendering of a page are suspended while JavaScript is executing.

Reducing all complex operations that take any more than a few hundred milliseconds into manageable portions becomes a necessity if we want to keep the interface responsive. Additionally, most browsers will produce a dialog box warning the user that a script has become "unresponsive" if it has run nonstop for at least 5 seconds, while some other browsers will even silently kill any script running for more than 5 seconds.

You may have been to a family reunion where a garrulous uncle won't stop talking and insists on telling the same stories over and over again. If no one else gets a chance to break in and get a word in edgewise, the conversation's not going to be pleasant for

anyone (except for Uncle Bruce). Likewise, code that hogs all the processing time results in an outcome that's less than desirable; producing an unresponsive user interface is never good. But situations will almost certainly arise that require us to process a significant amount of data, situations such as manipulating a couple of thousand DOM elements, for example.

On these occasions, timers can come to the rescue and become especially useful. Because timers are capable of effectively suspending the execution of a piece of JavaScript until a later time, they can also break individual pieces of code into fragments that aren't long enough to cause the browser to hang. Taking this into account, we can convert intensive loops and operations into nonblocking operations.

Let's look at the following example of a task that's likely to take a long time.

Listing 13.4 A long-running task

Makes 20,000 rows, which should qualify as a "boatload"

Finds the tbody element that we're going to create a boatload of rows for

For each row, creates 6 cells, each with a text node

Creates an individual row

Attaches the new row to its parent

```
<table><tbody></tbody></table>
<script>
  const tbody = document.querySelector("tbody");
  for (let i = 0; i < 20000; i++) {
    const tr = document.createElement("tr");
    for (let t = 0; t < 6; t++) {
      const td = document.createElement("td");
      td.appendChild(document.createTextNode(i + "," + t));
      tr.appendChild(td);
    }
    tbody.appendChild(tr);
  }
</script>
```

In this example, we're creating a total of 240,000 DOM nodes, populating a table with 20,000 rows of 6 cells, each containing a text node. This is incredibly expensive and will likely hang the browser for a noticeable period while executing, preventing the user from performing normal interactions (much in the same way that Uncle Bruce dominates the conversation at the family get-together).

What we need to do is shut up Uncle Bruce at regular intervals so that other people can get a chance to join the conversation. In code, we can introduce timers to create just such "breaks in the conversation," as shown in the next listing.

Listing 13.5 Using a timer to break up a long-running task

```
const rowCount = 20000;          ◁──── Sets up the data
const divideInto = 4;
const chunkSize = rowCount/divideInto;
let iteration = 0;
const table = document.getElementsByTagName("tbody")[0];
setTimeout(function generateRows(){
  const base = chunkSize * iteration;    ◁──── Computes where we left off last time
```

```
      for (let i = 0; i < chunkSize; i++) {
        const tr = document.createElement("tr");
        for (let t = 0; t < 6; t++) {
          const td = document.createElement("td");
          td.appendChild(
            document.createTextNode((i + base) + "," + t +
                                    "," + iteration));
          tr.appendChild(td);
        }
        table.appendChild(tr);
      }
      iteration++;
      if (iteration < divideInto)
        setTimeout(generateRows, 0);
    },0);
```

Schedules the next phase

Sets time-out delay to 0 to indicate that the next iteration should execute "as soon as possible," but after the UI has been updated

In this modification to the example, we've broken up the lengthy operation into four smaller operations, each creating its own share of DOM nodes. These smaller operations are much less likely to interrupt the flow of the browser, as shown in figure 13.14. Note that we've set it up so that the data values controlling the operation are collected into easily tweakable variables (rowCount, divideInto, and chunkSize), should we need to break the operations into, let's say, ten parts instead of four.
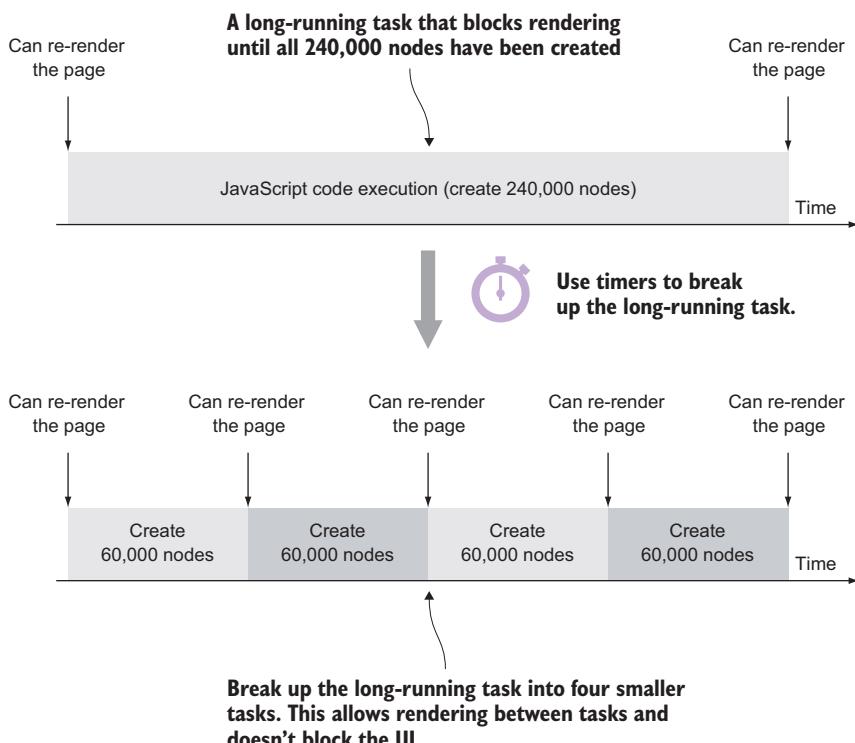
Figure 13.14 Use timers to break long-running tasks into smaller tasks that won't clog the event loop.

Also important to note is the little bit of math needed to keep track of where we left off in the previous iteration, base = chunkSize * iteration, and how we automatically schedule the next iterations until we determine that we're done:

```
if (iteration < divideInto)
    setTimeout(generateRows, 0);
```

What's impressive is just how little the code has to change in order to accommodate this new, asynchronous approach. We have to do a *little* more work to keep track of what's going on, to ensure that the operation is correctly conducted, and to schedule the execution parts. But beyond that, the core of the code looks similar to what we started with.

> **NOTE** In this case, we've used 0 for our time-out delay. If you've paid close attention to how the event loop works, you know that this doesn't mean that the callback will be executed in 0 ms. Instead, it's a way of telling the browser, please execute this callback as soon as possible; but unlike with microtasks, you're allowed to do page rendering in between. This allows the browser to update the UI and make our web applications more responsive.

The most perceptible change resulting from this technique, from the user's perspective, is that a long browser hang is replaced with four (or however many we choose) visual updates of the page. Although the browser will attempt to execute the code segments as quickly as possible, it will also render the DOM changes after each step of the timer. In the original version of the code, it needed to wait for one large bulk update.

Much of the time, these types of updates are imperceptible to the user, but it's important to remember that they do occur. We should strive to ensure that any code we introduce into the page doesn't perceptibly interrupt the normal operation of the browser.

It's often surprising just how useful this technique can be. By understanding how the event loop works, we can work around the restrictions of the single-threaded browser environment, while still providing a pleasant experience to the user.

Now that we understand the event loop and the roles timers can play in dealing with complex operations, let's take a closer look at how the events themselves work.

13.3 Working with events

When a certain event occurs, we can handle it in our code. As you've seen many times throughout this book, one common way of registering event handlers is by using the built-in addEventListener method, as shown in the following listing.

Listing 13.6 Registering event handlers

```html
<button id="myButton">Click</button>
<script>
  const myButton = document.getElementById("myButton");
  myButton.addEventListener("click", function myHandler(event){
```

Registers an event handler by using the addEventListener method

<div style="float:left; text-align:right; font-weight:bold; font-style:italic">
Accesses the
element that the
event has occurred
on through the
target property of
the passed-in
event
</div>

```
      assert(event.target === myButton,
            "The target of the event is also myButton");

      assert(this === myButton,
            "The handler is registered on myButton");
  });
</script>
```

Within the handler function, this refers to the element that has registered the handler.

In this snippet, we define a button named `myButton` and register a click event handler by using the built-in `addEventListener` method that's accessible from all elements.

After a click event occurs, the browser calls the associated handler, in this case the `myHandler` function. To this handler, the browser passes in an event object that contains properties that we can use to find out more information about the event, such as the position of the mouse or the mouse button that was clicked, in case we're dealing with a mouse-click event, or the pressed key if we're dealing with a keyboard event.

One of the properties of the passed-in event object is the `target` property, which references the element on which the event has occurred.

> **NOTE**   As within most other functions, within the event handler, we can use the `this` keyword. People often colloquially say that within an event handler, the `this` keyword refers to the object on which the event has occurred, but as we'll soon find out, this isn't exactly true. Instead, the `this` keyword refers to the element on which the event handler has been *registered*. To be honest, in most cases the element on which the event handler has been registered *is* the element on which the event has occurred, but there are exceptions. We'll explore these situations shortly.

Before exploring this concept further, let's set the stage so you can see how events can be propagated through the DOM.

13.3.1 *Propagating events through the DOM*

As we already know from chapter 2, in HTML documents, elements are organized in a tree. An element can have zero or more children, and every element (except the root `html` element) has exactly one parent. Now, suppose that we're working with a page that has an element inside another element, and both elements have a click handler, as in the following listing.

Listing 13.7   Nested elements and click handlers

```html
<html>
  <head>
    <style>
      #outerContainer {width:100px; height:100px; background-color: blue;}
      #innerContainer {width:50px; height:50px; background-color: red;}
    </style>
  </head>
  <body>
```

```
  <div id="outerContainer">                         Creates two
    <div id="innerContainer"></div>                 nested elements
  </div>
  <script>
    const outerContainer = document.getElementById("outerContainer");
    const innerContainer = document.getElementById("innerContainer");

    outerContainer.addEventListener("click", () => {    ◁─── Registers a click handler
      report("Outer container click");                       for the outer container
    });

    innerContainer.addEventListener("click", () => {    ◁─── Registers a click handler
      report("Inner container click");                       for the inner container
    });

    document.addEventListener("click", () => {      ◁─── Registers a click handler
      report("Document click");                          for the entire document
    });
  </script>
  </body>
</html>
```

Here we have two HTML elements, `outerContainer` and `innerContainer`, that are, like all the other HTML elements, contained within our global `document`. And on all three objects, we register a click handler.

Now let's suppose that a user clicks the `innerContainer` element. Because `inner-Container` is contained within the `outerContainer` element, and both of these elements are contained within the `document`, it's obvious that this should trigger the execution of all three event handlers, outputting three messages. What's not apparent is the order in which the event handlers should be executed.

Should we follow the order in which the events were registered? Should we start with the element on which the event occurs and move upward? Or should we start from the top and move downward toward the targeted element? Back in the day, when browsers were first making these decisions, the two primary competitors, Netscape and Microsoft, made opposing choices.

In Netscape's event model, the event handling starts with the top element and trickles down to the event target element. In our case, the event handlers would be executed in the following order: `document` click handler, `outerContainer` click handler, and finally `innerContainer` click handler. This is called *event capturing*.

Microsoft chose to go the other way around: start from the targeted element and bubble up the DOM tree. In our case, the events would be executed in the following order: `innerContainer` click handler, `outerContainer` click handler, and `document` click handler. This is called *event bubbling*.

The standard set by the W3 Consortium (www.w3.org/TR/DOM-Level-3-Events/), which is implemented by all modern browsers, embraces both approaches. An event is handled in two phases:

1   *Capturing phase*—An event is first captured at the top element and trickled down to the target element.

2   *Bubbling phase*—After the target element has been reached in the capturing phase, the event handling switches to bubbling, and the event bubbles up again from the target element to the top element.

These two phases are shown in figure 13.15.
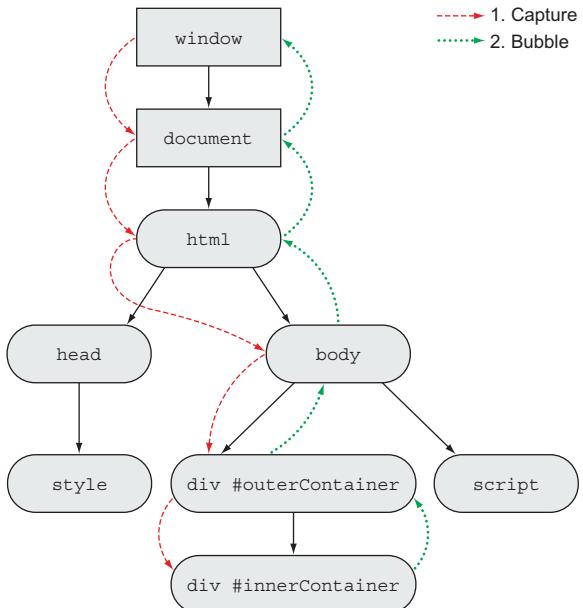
Figure 13.15   **With capturing, the event trickles down to the target element. With bubbling, the event bubbles up from the target element.**

We can easily decide which event-handling order we want to use, by adding another Boolean argument to the addEventListener method. If we use true as the third argument, the event will be captured, whereas if we use false (or leave out the value), the event will bubble. So in a sense, the W3C standard prefers event bubbling slightly more to event capturing, because bubbling has been made the default option.

Now let's go back to listing 13.7 and look closely at the way we've registered events:

```
outerContainer.addEventListener("click", () => {
  report("Outer container click");
});

innerContainer.addEventListener("click", () => {
  report("Inner container click");
});

document.addEventListener("click", () => {
  report("Document click");
});
```

As you can see, in all three cases, we've called the `addEventListener` method with only two arguments, which means that the default method, *bubbling*, is chosen. So in this case, if we click the `innerContainer` element, the event handlers would be executed in this order: `innerContainer` click handler, `outerContainer` click handler, `document` click handler.

Let's modify the code in listing 13.7 in the following way.

Listing 13.8   Capturing versus bubbling

```
const outerContainer = document.getElementById("outerContainer");
const innerContainer = document.getElementById("innerContainer");

document.addEventListener("click", () => {
  report("Document click");
});

outerContainer.addEventListener("click", () => {
  report("Outer container click");
}, true);

innerContainer.addEventListener("click", () => {
  report("Inner container click");
}, false);
```

By not specifying the third argument, the default, bubbling mode, is enabled.

Passing in true as the third argument enables capturing.

Passing in false enables bubbling.

This time, we set the event handler of the `outerContainer` to capturing mode (by passing in `true` as the third argument), and the event handlers of `innerContainer` (by passing in `false` as the third argument) and `document` to bubbling mode (leaving out the third argument chooses the default, bubbling mode).

As you know, a single event can trigger the execution of multiple event handlers, where each handler can be in either capturing or bubbling mode. For this reason, the event first goes through capturing, starting from the top element and trickling down to the event target element. When the target element is reached, bubbling mode is activated, and the event bubbles from the target element all the way back to the top.

In our case, capturing starts from the top, `window` object and trickles down to the `innerContainer` element, with the goal of finding all elements that have an event handler for this click event in capturing mode. Only one element, `outerContainer`, is found, and its matching click handler is executed as the first event handler.

The event continues traveling down the capturing path, but no more event handlers with capturing are found. After the event reaches the event target element, the `innerContainer` element, the event moves on to the bubbling phase, where it goes from the target element all the way back to the top, executing all bubbling event handlers on that path.

In our case, the `innerContainer` click handler will be executed as the second event handler, and the `document` click handler as the third. The output generated by clicking the `innerContainer` element, as well as the taken path, is shown in figure 13.16.

⑪ Bubbling
```
document.addEventListener("click", () => {
  report("Document click");
});
```

⑤ Capturing
```
outerContainer.addEventListener("click", () => {
  report("Outer container click");
}, true);
```

⑦ Bubbling
```
innerContainer.addEventListener("click", () => {
  report("Inner container click");
}, false);
```

Click the inner container

The element on which the event occurred
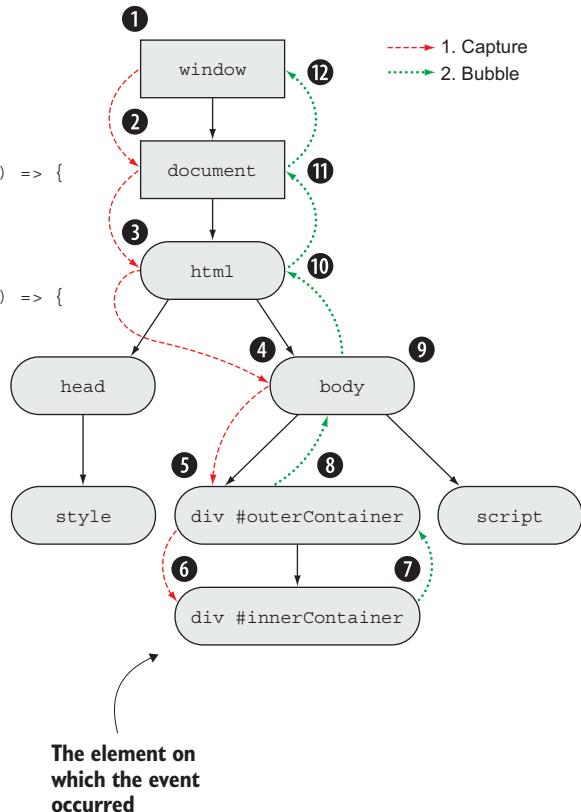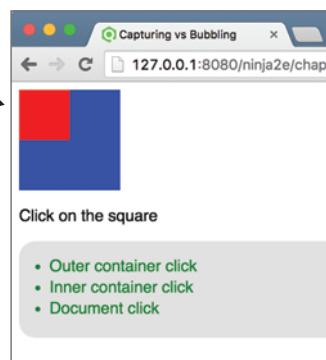
Figure 13.16 First the event trickles down from the top, executing all event handlers in capturing mode. When the target element is reached, the event bubbles up to the top, executing all event handlers in bubbling mode.

One of the things this example shows is that the element on which the event is handled doesn't have to be the element on which the event occurs. For example, in our case, the event occurs on the innerContainer element, but we can handle it on elements higher up in the DOM hierarchy, such as on the outerContainer or the document element.

This takes us back to the this keyword in event handlers, and why we explicitly stated that the this keyword refers to the element on which the event handler is registered, and not necessarily to the element on which the event occurs.

Again, let's modify our running example, as shown in the following listing.

Listing 13.9 The difference between `this` and `event.target` in event handlers

```
const outerContainer = document.getElementById("outerContainer");
const innerContainer = document.getElementById("innerContainer");

innerContainer.addEventListener("click", function(event){
```

```
  report("innerContainer handler");
  assert(this === innerContainer,
         "This referes to the innerContainer");
  assert(event.target === innerContainer,
         "event.target refers to the innerContainer");
});
```

Within the innerContainer handler, both this and event.target point to the innerContainer element.

```
outerContainer.addEventListener("click", function(event){
  report("innerContainer handler");
  assert(this === outerContainer,
         "This refers to the outerContainer");
  assert(event.target === innerContainer,
         "event.target refers to the innerContainer");
});
```

Within the outerContainer handler, if we're handling the event originating on the innerContainer, this will refer to the outerContainer and event.target to innerContainer.

Again, let's look at the application execution when a click occurs on innerContainer. Because both event handlers use event bubbling (there's no third argument set to true in the addEventListener methods), first the innerContainer click handler is invoked. Within the body of the handler, we check that both the this keyword and the event.target property refer to the innerContainer element:

```
  assert(this === innerContainer,
         "This refers to the innerContainer");
  assert(event.target === innerContainer,
         "event.target refers to the innerContainer");
```

The this keyword points to the innerContainer element because that's the element on which the current handler has been *registered*, whereas the event.target property points to the innerContainer element because that's the element on which the event has *occurred*.

Next, the event bubbles up to the outerContainer handler. This time, the this keyword and the event.target point to different elements:

```
  assert(this === outerContainer,
         "This refers to the outerContainer");
  assert(event.target === innerContainer,
         "event.target refers to the innerContainer");
```

As expected, the this keyword refers to the outerContainer element, because this is the element on which the current handler has been registered. On the other hand, the event.target property points to the innerContainer element, because this is the element on which the event has occurred.

Now that we understand how an event is propagated through the DOM tree and how to access the element on which the event has originally occurred, let's see how to apply this knowledge to write less memory-intensive code.

DELEGATING EVENTS TO AN ANCESTOR

Let's say that we want to visually indicate whether a cell within a table has been clicked by the user, by initially displaying a white background for each cell and then changing the background color to yellow after the cell is clicked. Sounds easy enough. We can iterate through all the cells and establish a handler on each one that changes the background color property:

```
const cells = document.querySelectorAll('td');
for (let n = 0; n < cells.length; n++) {
  cells[n].addEventListener('click', function(){
    this.style.backgroundColor = 'yellow';
  });
}
```

Sure this works, but is it elegant? No. We're establishing the exact same event handler on potentially hundreds of elements, and they all do *the exact same thing*.

A much more elegant approach is to establish a single handler at a level higher than the cells that can handle all the events using event bubbling. We know that all the cells will be descendants of their enclosing table, and we know that we can get a reference to the element that was clicked via event.target. It's much more suave to *delegate* the event handling to the table, as follows:

```
const table = document.getElementById('someTable');
table.addEventListener('click', function(event){
  if (event.target.tagName.toLowerCase() === 'td')
    event.target.style.backgroundColor = 'yellow';
});
```

Performs an action only if the click happens on a cell element (and not on a random descendant)

Here, we establish one handler that easily handles the work of changing the background color for all cells in the table that were clicked. This is much more efficient and elegant.

With event delegation, we have to make sure that it's only applied to elements that are ancestors of the elements that are the event targets. That way, we're sure that the events will eventually bubble up to the element to which the handler has been delegated.

So far, we've been dealing with events that are provided by the browser, but haven't you ever fervently desired the ability to trigger your own *custom* events?

13.3.2 *Custom events*

Imagine a scenario in which you want to perform an action, but you want to trigger it under a variety of conditions from different pieces of code, perhaps even from code that's in shared script files. A novice would repeat the code everywhere it's needed. A journeyman would create a global function and call it from everywhere it's needed. A ninja would use custom events. But why?

LOOSE COUPLING

Say we're doing operations from shared code, and we want to let page code know when it's time to react to a particular condition. If we use the journeyman's global function approach, we introduce the disadvantage that our shared code needs to define a fixed name for the function, and all pages that use the shared code need to use such a function.

Moreover, what if there are multiple things to do when the triggering condition occurs? Making allowances for multiple notifications would be arduous and necessarily messy. These disadvantages are a result of *close coupling*, in which the code that detects the conditions has to know the details of the code that will react to that condition.

Loose coupling, on the other hand, occurs when the code that triggers the condition doesn't know anything about the code that will react to the condition, or even if there's anything that will react to it at all. One of the advantages of event handlers is that we can establish as many as we want, and these handlers are completely independent of each other. So event handling is a good example of loose coupling. When a button-click event is triggered, the code triggering the event has no knowledge of what handlers we've established on the page, or even if there are any. Rather, the click event is pushed onto the task queue by the browser, and whatever caused the event to trigger could care less what happens after that. If handlers have been established for the click event, they'll eventually be individually invoked in a completely independent fashion.

There's much to be said for loose coupling. In our scenario, the shared code, when it detects an interesting condition, triggers a signal of some sort that says, "This interesting thing has happened; anyone interested can deal with it," and it couldn't give a darn whether anyone's interested. Let's examine a concrete example.

AN AJAX-Y EXAMPLE

Let's pretend that we've written some shared code that will be performing an Ajax request. The pages that this code will be used on want to be notified when an Ajax request begins and when it ends; each page has its own things that it needs to do when these "events" occur.

For example, on one page using this package, we want to display a spinning pinwheel when an Ajax request starts, and we want to hide it when the request completes, in order to give the user some visual feedback that a request is being processed. If we imagine the start condition as an event named `ajax-start`, and the stop condition as `ajax-complete`, wouldn't it be grand if we could establish event handlers on the page for these events that show and hide the image as appropriate?

Consider this:

```
document.addEventListener('ajax-start', e => {
  document.getElementById('whirlyThing').style.display = 'inline-block';
});
document.addEventListener('ajax-complete', e => {
  document.getElementById('whirlyThing').style.display = 'none';
});
```

Sadly, these events don't exist, but there's nothing stopping us from bringing them into existence.

CREATING CUSTOM EVENTS

Custom events are a way of simulating (for the user of our shared code) the experience of a real event, but an event that has business sense within the context of our application. The following listing shows an example of triggering a custom event.

Listing 13.10 Using custom events

```
<style>
  #whirlyThing { display: none; }
</style>
<button type="button" id="clickMe">Start</button>     ◁   A button that we'll click to
<img id="whirlyThing" src="whirly-thing.gif" />            simulate an Ajax request

                                                          Our spinner image,
                                                          which indicates
                                                          loading, if shown

<script>
  function triggerEvent(target, eventType, eventDetail){   Uses the CustomEvent
    const event = new CustomEvent(eventType, {        ◁   constructor to create a
      detail: eventDetail                                 new event
    });
    target.dispatchEvent(event);                     ◁   Uses the built-in dispatchEvent
  }                                                       method to dispatch the event
                                                          to the specified element
  function performAjaxOperation() {
    triggerEvent(document, 'ajax-start', { url: 'my-url'});
    setTimeout(() => {
      triggerEvent(document, 'ajax-complete');
    },5000);                                              Mimics our Ajax request with a
  }                                                       timer. At the start of execution,
                                                          triggers the ajax-start event. After
  const button = document.getElementById('clickMe');      enough time elapses, triggers the
  button.addEventListener('click', () => {                ajax-complete event. Passes in a
    performAjaxOperation();                                URL as additional event data
  });
                                                          Checks that we can access
  document.addEventListener('ajax-start', e => {            additional event data
    document.getElementById('whirlyThing').style.display = 'inline-block';
    assert(e.detail.url === 'my-url', 'We can pass in event data');  ◁
  });

  document.addEventListener('ajax-complete', e => {    ◁
      document.getElementById('whirlyThing').style.display = 'none';
  });                                                 Handles the ajax-complete event
</script>                                             by hiding our whirly thing
```

Passes in information to the event object through the detail property

When a button is clicked, the Ajax operation is started.

Handles the ajax-start event by showing our whirly thing

In this example, we explore custom events by establishing the scenario described in the previous section: An animated pinwheel image is displayed while an Ajax operation is underway. The operation is triggered by the click of a button.

In a completely decoupled fashion, a handler for a custom event named `ajax-start` is established, as is the one for the `ajax-complete` custom event. The handlers for these events show and hide the pinwheel image, respectively:

```
button.addEventListener('click', () => {
  performAjaxOperation();
});

document.addEventListener('ajax-start', e => {
  document.getElementById('whirlyThing').style.display = 'inline-block';
  assert(e.detail.url === 'my-url', 'We can pass in event data');
});

document.addEventListener('ajax-complete', e => {
    document.getElementById('whirlyThing').style.display = 'none';
});
```

Note that the three handlers know nothing of each other's existence. In particular, the button click handler has no responsibilities with respect to showing and hiding the image.

The Ajax operation itself is simulated with the following code:

```
function performAjaxOperation() {
  triggerEvent(document, 'ajax-start', { url: 'my-url'});
  setTimeout(() => {
    triggerEvent(document, 'ajax-complete');
  }, 5000);
}
```

The function triggers the `ajax-start` event and sends data about the event (the `url` property), pretending that an Ajax request is about to be made. The function then issues a 5-second time-out, simulating an Ajax request that spans 5 seconds. When the timer expires, we pretend that the response has been returned and trigger an `ajax-complete` event to signify that the Ajax operation has completed.

Notice the high degree of decoupling throughout this example. The shared Ajax operation code has no knowledge of what the page code is going to do when the events are triggered, or even if there's page code to trigger at all. The page code is modularized into small handlers that don't know about each other. Furthermore, the page code has no idea how the shared code is doing its thing; it just reacts to events that may or may not be triggered.

This level of decoupling helps to keep code modular, easier to write, and a lot easier to debug when something goes wrong. It also makes it easy to share portions of code and to move them around without fear of violating a coupled dependency between the code fragments. Decoupling is a fundamental advantage when using custom events in code, and it allows us to develop applications in a much more expressive and flexible manner.

13.4   Summary

- An event-loop task represents an action performed by the browser. Tasks are grouped into two categories:
    - Macrotasks are discrete, self-contained browser actions such as creating the main document object, handling various events, and making URL changes.
    - Microtasks are smaller tasks that should be executed as soon as possible. Examples include promise callbacks and DOM mutation changes.
- Because of the single-threaded execution model, tasks are processed one at a time, and after a task starts executing, it can't be interrupted by another task. The event loop usually has at least two event queues: a macrotask queue and a microtask queue.
- Timers provide the ability to asynchronously delay the execution of a piece of code by *at least* some number of milliseconds.
    - Use the `setTimeout` function to execute a callback once after the specified delay has elapsed.
    - Use the `setInterval` function to initiate a timer that will try to execute the callback at the specified delay interval, until canceled.
    - Both functions return an ID of the timer that we can use to cancel a timer through the `clearTimeout` and `clearInterval` functions.
    - Use timers to break up computationally expensive code into manageable chunks that won't clog the browser.
- The DOM is a hierarchical tree of elements, and an event that occurs on an element (the target) is usually propagated through the DOM. There are two propagation mechanisms:
    - In event capturing, the event trickles down from the top element all the way to the target element.
    - In event bubbling, the event bubbles up from the target element all the way to the top element.
- When calling event handlers, the browser also passes in an event object. Access the element on which the event has occurred through the event's `target` property. Within the handler, use the `this` keyword to refer to the element on which the handler has been registered.
- Use custom events, created through the built-in `CustomEvent` constructor, and dispatched with the `dispatchEvent` method, to reduce coupling between different parts of your application.

13.5   Exercises

1   Why is it important that adding tasks into the task queue happens outside the event loop?

2   Why is it important that each iteration of the event loop doesn't take much more than about 16 ms?

3 What's the output from running the following code for 2 seconds?

```
setTimeout(function(){
  console.log("Timeout ");
}, 1000);

setInterval(function(){
  console.log("Interval ");
}, 500);
```

a `Timeout Interval Interval Interval Interval`

b `Interval Timeout Interval Interval Interval`

c `Interval Timeout Timeout`

4 What's the output from running the following code for 2 seconds?

```
const timeoutId = setTimeout(function(){
  console.log("Timeout ");
}, 1000);

setInterval(function(){
  console.log("Interval ");
}, 500);

clearTimeout(timeoutId);
```

a `Interval Timeout Interval Interval Interval`

b `Interval`

c `Interval Interval Interval Interval`

5 What's the output from running the following code and clicking the element with the ID `inner`?

```
<body>
  <div id="outer">
    <div id="inner"></div>
  </div>
  <script>
    const innerElement = document.querySelector("#inner");
    const outerElement = document.querySelector("#outer");
    const bodyElement = document.querySelector("body");

    innerElement.addEventListener("click", function(){
        console.log("Inner");
    });

    outerElement.addEventListener("click", function(){
        console.log("Outer");
    }, true);

    bodyElement.addEventListener("click", function(){
        console.log("Body");
```

```
    })
  </script>
</body>
```

a Inner Outer Body

b Body Outer Inner

c Outer Inner Body

14 *Developing cross-browser strategies*

This chapter covers

- Developing reusable, cross-browser JavaScript code
- Analyzing cross-browser issues needing to be tackled
- Tackling those issues in a smart way

Anyone who's spent time developing on-page JavaScript code knows that a wide range of pain points exist when it comes to ensuring that the code works flawlessly across a set of supported browsers. These considerations span from providing basic development for immediate needs, to planning for future browser releases, all the way to reusing code on web pages that have yet to be created.

Coding for multiple browsers is a nontrivial task that must be balanced according to the development methodologies that you have in place, as well as the resources available to your project. As much as we'd love for our pages to work perfectly in every browser that ever existed or will ever exist, reality will rear its ugly head and we must realize that we have finite development resources. We must plan to apply those resources appropriately and carefully, getting the biggest bang for our buck.

Because of this, we start this chapter with advice on choosing which browsers to support. This is followed with a discussion of the major concerns regarding cross-browser development, as well as effective strategies for dealing with such problems. Let's jump right into ways to carefully choose supported browsers.

. . . .

Do you know?

What are some common ways of dealing with inconsistencies in behavior with different browsers?

What is the best way to make your code usable on other people's pages?

Why are polyfills useful in cross-browser scripting?

. . . .

14.1 *Cross-browser considerations*

Perfecting our JavaScript programming skills will take us far, especially now that JavaScript has escaped the confines of the browser and is being used on the server with Node.js. But when developing browser-based JavaScript applications (which is the focus of this book), sooner rather than later, we're going to run face first into *The Browsers* and their various issues and inconsistencies.

In a perfect world, all browsers would be bug-free and would support web standards consistently, but as we all know, we don't live in that world. Although the quality of browsers has improved greatly as of late, all still have some bugs, missing APIs, and browser-specific quirks that we'll need to deal with. Developing a comprehensive strategy for tackling these browser issues, and becoming intimately familiar with their differences and quirks, isn't less important than proficiency in JavaScript itself.

When writing browser applications, choosing which browsers to support is important. We'd probably like to support them all, but limitations on development and testing resources dictate otherwise. So how do we decide which to support, and to what level?

One approach that we can employ is loosely borrowed from an older Yahoo! approach, *graded browser support.* In this technique, we create a browser-support matrix that serves as a snapshot of how important a browser and its platform are to our needs. In this table, we list the target platforms on one axis, and the browsers on the other. Then, in the table cells, we give a "grade" (A through F, or any other grading system that meets our needs) to each browser/platform combination. Table 14.1 shows a hypothetical example.

Note that we haven't filled in any grades. The grades you assign to a particular combination of platform and browser are entirely dependent on the needs and requirements of your project, as well as other important factors, such as the makeup of the target audience. We can use this approach to come up with grades that measure the importance of support for that platform/browser, and combine that info with the cost of that support to try to come up with the optimal set of supported browsers.

Table 14.1 A hypothetical browser-support matrix

	Windows	OS X	Linux	iOS	Android
IE 9		N/A	N/A	N/A	N/A
IE10		N/A	N/A	N/A	N/A
IE11		N/A	N/A	N/A	N/A
Edge		N/A	N/A	N/A	N/A
Firefox				N/A	
Chrome					
Opera					
Safari			N/A		N/A

When we choose to support a browser, we're typically making the following promises:

- We'll actively test against that browser with our test suite.
- We'll fix bugs and regressions associated with that browser.
- The browser will execute our code with a reasonable level of performance.

Because it's impractical to develop against lots of platform/browser combinations, we must weigh the costs versus the benefits of supporting the various browsers. This analysis must take into account multiple considerations, and the primary ones are as follows:

- The expectations and needs of the target audience
- The market share of the browser
- The amount of effort necessary to support the browser

The first point is a subjective one that only your project can determine. Market share, on the other hand, can frequently be measured using available information. And a rough estimate of the effort involved in supporting each browser can be determined by considering the capabilities of the browsers and their adherence to modern standards.

Figure 14.1 shows a sample chart that represents information on browser use (obtained from http://gs.statcounter.com for April 2016). Any piece of reusable JavaScript code, whether it's a mass-consumption JavaScript library or our own on-page code, should be developed to work in as many environments as feasible, concentrating on the browsers and platforms that are important to the end user. For mass-consumption libraries, that's a large set; for more-targeted applications, the required set may be narrower.

But it's vitally important not to bite off more than you can chew, and quality should never be sacrificed for coverage. That's important enough to repeat; in fact, we urge you to read it out loud:

Quality should never be sacrificed for coverage.

Desktop browser statistics

Mobile browser statistics
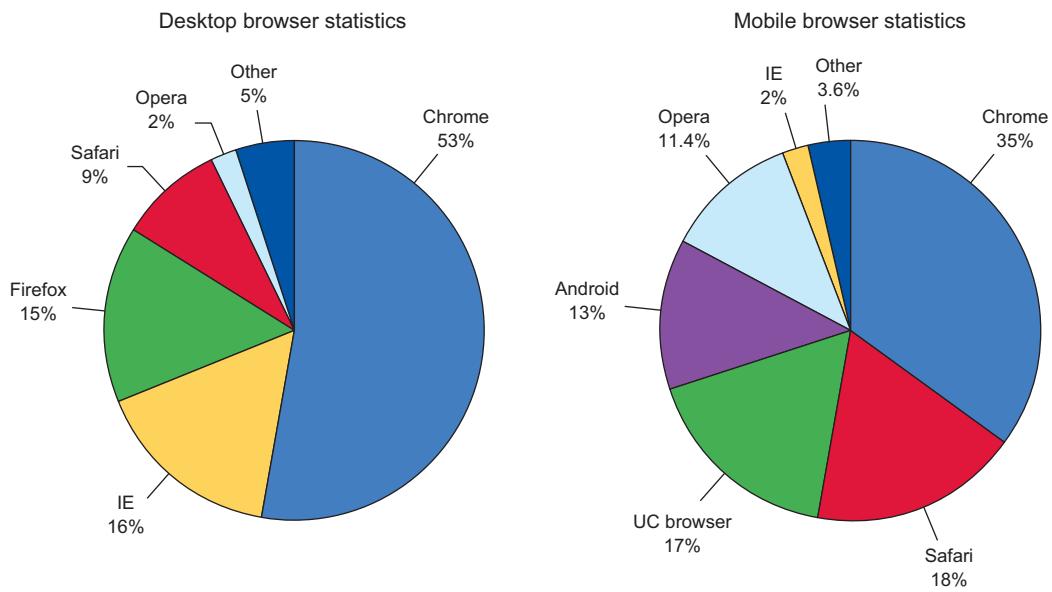
Figure 14.1   Looking at the usage statistics of browsers on desktop and mobile devices gives us an idea of which browsers to focus our attention on.

In this chapter, we'll examine the situations that JavaScript code will find itself up against with regards to cross-browser support. Then we'll examine some of the best ways to write that code with the aim of alleviating any potential problems that those situations pose. This should go a long way in helping you decide which of these techniques are worth your time to adopt, and it should help you fill out your own browser-support chart.

14.2   *The five major development concerns*

Any piece of nontrivial code carries myriad development concerns to worry about. But five major points pose the biggest challenges to our reusable JavaScript code, as illustrated in figure 14.2.

These are the five points:

- Browser bugs
- Browser bug fixes
- External code
- Browser regressions
- Missing features in the browsers

We'll want to balance the amount of time we spend on each point against the resulting benefits. Ultimately, these
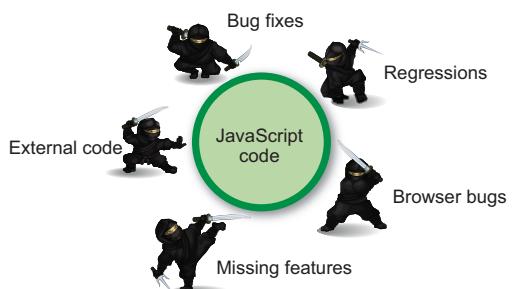
Figure 14.2   The five major points of concern for the development of reusable JavaScript

are questions that you'll have to answer, applying them to your own situation. An analysis of your intended audience, development resources, and schedule are all factors that go into your decision.

When striving to develop reusable JavaScript code, we must take all the points into consideration but pay closest attention to the most popular browsers that exist right now, because these are most likely to be used by our targeted audience. With other, less popular browsers, we should at least make sure that our code degrades gracefully. For example, if a browser doesn't support a certain API, at the very least, we should be careful that our code doesn't throw any exceptions so that the rest of the code can be executed.

In the following sections, we'll break down these various concerns to get a better understanding of the challenges we're up against and how to combat them.

14.2.1 Browser bugs and differences

One of the concerns that we'll need to deal with when developing reusable JavaScript code is handling the various bugs and API differences associated with the set of browsers we've decided to support. Even though browsers are much more uniform these days, any features that we provide in our code should be completely and *verifiably* correct in *all* browsers we choose to support.

The way to achieve this is straightforward: We need a comprehensive suite of tests to cover both the common and fringe use cases of the code. With good test coverage, we can feel safe in knowing that the code we develop will work in the supported set of browsers. And assuming that no subsequent browser changes break backward compatibility, we'll have a warm, fuzzy feeling that our code will even work in future versions of those browsers. We'll be looking at specific strategies for dealing with browser bugs and differences in section 14.3.

A tricky point in all of this is implementing fixes for current browser bugs in such a way that they're resistant to any fixes for those bugs that are implemented in future versions of the browser.

14.2.2 Browser bug fixes

Assuming that a browser will forever present a particular bug is foolhardy—most browser bugs eventually get fixed, and *counting* on the presence of the bug is a dangerous development strategy. It's best to use the techniques in section 14.3 to make sure that any bug work-arounds are future-proofed as much as possible.

When writing a piece of reusable JavaScript code, we want to make sure that it can last a long time. As with writing any aspect of a website (CSS, HTML, and so on), we don't want to have to go back and fix code that's broken by a new browser release.

Making assumptions about browser bugs causes a common form of website breakage: specific hacks put in place to work around bugs presented by a browser that break when the browser fixes the bugs in future releases.

The problem with handling browser bugs is twofold:

- Our code is liable to break when the bug fix is eventually instituted.
- We could end up training browser vendors to *not* fix bugs for fear of causing websites to break.

An interesting example of the second situation occurred just recently, with the `scrollTop` bug (https://dev.opera.com/articles/fixing-the-scrolltop-bug/).

When dealing with elements in the HTML DOM, we can use the `scrollTop` and `scrollLeft` properties to access and modify the current scroll position of the element. But if we use these properties on the root, `html` element, these properties should, according to specification, instead report (and influence) the scroll position of the viewport. IE 11 and Firefox closely follow this specification. Unfortunately, Safari, Chrome, and Opera don't. Instead, if you try to modify these properties of the root, `html` element, nothing happens. To achieve the same effect in these browsers, we have to use the `scrollTop` and `scrollLeft` properties on the `body` element.

When faced with this inconsistency, web developers have often resorted to detecting the current name of the browser (through the user agent string, more on this later), and then modifying the `scrollTop` and `scrollLeft` of the `html` element if our JavaScript code is being executed in IE or Firefox, and of the `body` element if the code is being executed in Safari, Chrome, or Opera. Unfortunately, this way of circumventing this bug has proved to be disastrous. Because many pages now explicitly encode "if this is Safari, Chrome, or Opera," modify the `body` element, these browsers can't really fix this bug, because the bug fix would, ironically, cause failures in many web pages.

This brings up another important point concerning bugs: When determining whether a piece of functionality is potentially a bug, always verify it with the specification!

A browser bug is also different from an unspecified API. It's important to refer to browser specifications, because they provide the exact standards that browsers use to develop and improve their code. In contrast, the implementation of an unspecified API could change at any point (especially if the implementation ever attempts to become standardized). In the case of inconsistencies in unspecified APIs, you should always test for your expected output. Always be aware that future changes could occur in these APIs as they become solidified.

Additionally, there's a distinction between bug fixes and API changes. Whereas bug fixes are easily foreseen—a browser will eventually fix the bugs in its implementation, even if it takes a long time—API changes are much harder to spot. Standard APIs are unlikely to change (though it's not completely unheard of); changes are much more likely to occur with unspecified APIs.

Luckily, this rarely happens in a way that will massively break most web applications. But if it does, it's effectively undetectable in advance (unless, of course, we test every single API that we ever touch—but the overhead incurred in such a process would be ludicrous). API changes of this sort should be handled like any other regression.

For our next point of concern, we know that no man is an island, and neither is our code. Let's explore the ramifications.

14.2.3 External code and markup

Any reusable code must coexist with the code that surrounds it. Whether we're expecting our code to work on pages that we write or on websites developed by others, we need to ensure that it can exist on the page with any other random code.

This is a double-edged sword: Our code not only must be able to withstand living with potentially poorly written external code, but also must take care not to have adverse effects on the code with which it lives.

Exactly how much we need to be vigilant about this point of concern depends a great deal on the environment in which we expect the code to be used. For example, if we're writing reusable code for a single or limited number of websites that we have some level of control over, it might be safe to worry less about effects of external code because we know where the code will operate, and we can, to some degree, fix any problems ourselves.

> **TIP** This is an important enough concern to warrant an entire book on the subject. If you'd like to delve more deeply, we highly recommend *Third-Party JavaScript* by Ben Vinegar and Anton Kovalyov (Manning, 2013, https://www.manning.com/books/third-party-javascript).

If we're developing code that will have a broad level of applicability in unknown (and uncontrollable) environments, we'll need to make doubly sure that our code is robust. Let's discuss some strategies to achieve that.

ENCAPSULATING OUR CODE

To keep our code from affecting other pieces of code on the pages where it's loaded, it's best to practice *encapsulation*. In general, this refers to the act of placing something in, or as if in, a capsule. A more domain-focused definition is "a language mechanism for restricting access to some of the object's components." Your Aunt Mathilda might summarize it more succinctly as "Keep your nose in your own business!"

Keeping an incredibly small global footprint when introducing our code into a page can go a long way toward making Aunt Mathilda happy. In fact, keeping our global footprint to a handful of global variables, or better yet, *one*, is fairly easy.

As you saw in chapter 12, jQuery, the most popular client-side JavaScript library, is a good example of this. It introduces one global variable (a function) named `jQuery`, and one alias for that global variable, `$`. It even has a supported means to give the `$` alias back to whatever other on-page code or other library may want to use it.

Almost all operations in jQuery are made via the `jQuery` function. And any other functions that it provides (called *utility functions*) are defined as properties of `jQuery` (remember from chapter 3 how easy it is to define functions that are properties of other functions), thus using the name `jQuery` as a *namespace* for all its definitions.

We can use the same strategy. Let's say that we're defining a set of functions for our own use, or for the use of others, that we'll group under a namespace of our own choosing—say, `ninja`.

We could, like jQuery, define a global function named `ninja()` that performs various operations based on what we pass to the function. For example:

```
var ninja = function(){ /* implementation code goes here */ }
```

Defining our own utility functions that use this function as their namespace is easy:

```
ninja.hitsuke = function(){ /* code to distract guards with fire here */ }
```

If we didn't want or need `ninja` to be a function but only to serve as a namespace, we could define it as follows:

```
var ninja = {};
```

This creates an empty object in which we can define properties and functions in order to keep from adding these names to the global namespace.

Other practices that we want to avoid, in order to keep our code encapsulated, are modifying any existing variables, function prototypes, or even DOM elements. Any aspect of the page that our code modifies, outside itself, is a potential area for collision and confusion.

The other side of the two-way street is that even if *we* follow best practices and carefully encapsulate our code, we can't be assured that code we haven't written is going to be as well-behaved.

DEALING WITH LESS-THAN-EXEMPLARY CODE

There's an old joke that's been going around since Grace Hopper removed that moth from a relay back in the Cretaceous period: "The only code that doesn't suck is the code you write yourself." This may seem cynical, but when our code coexists with code that we can't control, we should assume the worst, just to be safe.

Some code, even if well-written, might *intentionally* be doing things like modifying function prototypes, object properties, and DOM element methods. This practice, well-meant or otherwise, can lay traps for us to step into.

In these circumstances, our code could be doing something innocuous, such as using JavaScript arrays, and no one could fault us for making the simple assumption that JavaScript arrays are going to act like JavaScript arrays. But if some other on-page code modifies the way that arrays work, our code could end up not working as intended, through absolutely no fault of our own.

Unfortunately, there aren't many steadfast rules when dealing with situations of this nature, but we can take some mitigating steps. The next few sections introduce these defensive steps.

COPING WITH GREEDY IDS

Most browsers exhibit an *anti-feature* (we can't call it a *bug* because the behavior is absolutely intended) that can cause our code to trip and fall unexpectedly. This feature causes element references to be added to other elements by using the `id` or `name` attributes of the original element. And when that `id` or `name` conflicts with properties that are already part of the element, bad things can happen.

Take a look at the following HTML snippet to observe the nastiness that can ensue as a result of these *greedy IDs*:

```
<form id="form" action="/conceal">
  <input type="text" id="action"/>
  <input type="submit" id="submit"/>
</form>
```

Now, in the browsers, let's call this:

```
var what = document.getElementById('form').action;
```

Rightly, we'd expect this to be the value of the form's `action` attribute. And in most cases, it would be. But if we inspect the value of variable `what`, we find that it's instead a reference to the `input#action` element! Huh?

Let's try something else:

```
document.getElementById('form').submit();
```

This statement should cause the form to be submitted, but instead, we get a script error:

```
Uncaught TypeError: Property 'submit' of object #<HTMLFormElement> is not a function
```

What's going on?

The browsers have added properties to the `<form>` element for each of the input elements within the form that reference the element. This might seem handy at first, until we realize that the name of the added property is taken from the `id` or `name` values of the input elements. And if that value just happens to be an already-used property of the form element, such as `action` or `submit`, those original properties are replaced by the new property. This is usually referred to as *DOM clobbering*.

So, before the `input#submit` element is created, the reference `form.action` points to the value of the `action` attribute for the `<form>`. Afterward, it points to the `input#submit` element. The same thing happens to `form.submit`. Yeesh!

This is a remnant from way back, from a time when browsers didn't have a rich set of API methods for fetching elements from the DOM. Browser vendors added this feature to give easy access to form elements. Nowadays we can easily access any element in the DOM, so we're left with only the unfortunate side effects of the feature.

In any case, this particular "feature" of the browsers can cause numerous and mystifying problems in our code, and we should keep it in mind when debugging. When we encounter properties that have seemingly been inexplicably transformed into something other than what we expect, DOM clobbering is a likely culprit.

Luckily, we can avoid this problem in our own markup by avoiding simple `id` and `name` values that can conflict with standard property names, and we can encourage others to do the same. The value `submit` is especially to be avoided, as it's a common source of frustrating and perplexing buggy behavior.

LOADING ORDER OF STYLE SHEETS AND SCRIPTS

Often we'll expect CSS rules to already be available by the time our code executes. One of the best ways to ensure that CSS rules provided by style sheets are defined when our JavaScript code executes is to include the external style sheets *prior* to including the external script files.

Not doing so can cause unexpected results, because the script attempts to access the as-yet-undefined style information. Unfortunately, this issue can't easily be rectified with pure JavaScript and should instead be handled with user documentation.

These last few sections have covered some basic examples of how externalities can affect how our code works, frequently in unintentional and confounding ways. Issues with our code will often pop up when other users try to integrate it into *their* sites, at which point we should be able to diagnose the issues and build appropriate tests to handle them. At other times, we'll discover such problems when we integrate others' code into our pages, and hopefully the tips in these sections will help to identify the causes.

It's unfortunate that there are no better and deterministic solutions to handling these integration issues other than to take some smart first steps and to write our code defensively. We'll now move on to the next point of concern.

14.2.4 Regressions

Regressions are one of the hardest problems we'll encounter in creating reusable and maintainable JavaScript code. These are bugs, or non-backward-compatible API changes (mostly to unspecified APIs), that browsers have introduced and that cause code to break in unpredictable ways.

> **NOTE**   Here we're using the term *regression* in its classical definition: a feature that used to work but no longer functions as expected. This is usually unintentional, but it's sometimes caused by deliberate changes that break existing code.

ANTICIPATING CHANGES

There *are* some API changes that, with some foresight, we can proactively detect and handle, as shown in listing 14.1. For example, with Internet Explorer 9, Microsoft introduced support for DOM level 2 event handlers (bound using the `addEventListener` method), while previous versions of IE were using the IE-specific built-in `attachEvent` method. For code written prior to IE 9, simple feature detection was able to handle that change.

Listing 14.1   Anticipating an upcoming API change

```
function bindEvent(element, type, handle) {
  if (element.addEventListener) {
    element.addEventListener(type, handle, false);        ◁—  Binds using the
  }                                                            standard API
  else if (element.attachEvent) {
```

```
    element.attachEvent("on" + type, handle);
  }
}
```
Binds using a proprietary API

In this example, we future-proof our code knowing (or hoping against hope) that someday Microsoft will bring Internet Explorer into line with DOM standards. If the browser supports the standards-compliant API, we use feature detection to infer that and use the standard API, the `addEventListener` method. If not, we check to see that the IE-proprietary method `attachEvent` is available and use that. If all else fails, we do nothing.

Most future API changes, alas, aren't that easy to predict, and there's no way to predict upcoming bugs. This is but one of the important reasons that we've stressed testing throughout this book. In the face of unpredictable changes that will affect our code, the best that we can hope for is to be diligent in monitoring our tests for each browser release, and to quickly address issues that regressions may introduce.

Having a good suite of tests and keeping close track of upcoming browser releases is absolutely the best way to deal with future regressions of this nature. It doesn't have to be taxing on your normal development cycle, which should already include routine testing. Running these tests on new browser releases should always be factored into the planning of any development cycle.

You can get information on upcoming browser releases from the following locations:

- Microsoft Edge (a successor to IE): http://blogs.windows.com/msedgedev/
- Firefox: http://ftp.mozilla.org/pub/firefox/nightly/latest-trunk/
- WebKit (Safari): https://webkit.org/nightly/
- Opera: https://dev.opera.com/
- Chrome: http://chrome.blogspot.hr/

Diligence is important. Because we can never fully predict the bugs that will be introduced by a browser, it's best to make sure that we stay on top of our code and quickly avert any crises that may arise.

Thankfully, browser vendors are doing a lot to make sure that regressions of this nature don't occur, and browsers often have test suites from various JavaScript libraries integrated into their main browser test suite. This ensures that no future regressions will be introduced that affect those libraries directly. Although this won't catch all regressions (and certainly won't in all browsers), it's a great start and shows good progress by the browser vendors toward preventing as many issues as possible.

In this section, we've gone through four major points of concern for the development of reusable JavaScript: browser bugs, browser bug fixes, external code, and browser regressions. The fifth point—missing features in the browsers—deserves a special mention, so we cover it in the next section, alongside other implementation strategies relevant for cross-browser web applications.

14.3   Implementation strategies

Knowing which issues to be aware of is only half the battle. Figuring out effective solutions and using them to implement robust cross-browser code is another matter.

A wide range of strategies are available, and although not every strategy will work in every situation, the range presented in this section covers most of the concerns that we'll need to address within our robust code bases. Let's start with something that's easy and almost trouble free.

14.3.1   Safe cross-browser fixes

The simplest (and safest) classes of cross-browser fixes are those that exhibit two important traits:

- They have no negative effects or side effects on other browsers.
- They use no form of browser or feature detection.

The instances for applying these fixes may be rare, but they're a tactic that we should always strive for in our applications.

Let's look at an example. The following code snippet represents a change (plucked from jQuery) that came about when working with Internet Explorer:

```
// ignore negative width and height values
if ((key == 'width' || key == 'height') && parseFloat(value) < 0)
  value = undefined;
```

Some versions of IE throw an exception when a negative value is set on the `height` or `width` style properties. All other browsers ignore negative input. This work-around ignores all negative values in *all* browsers. This change prevents an exception from being thrown in Internet Explorer and has no effect on any other browser. This is a painless change that provides a unified API to the user (because throwing unexpected exceptions is never desired).

Another example of this type of fix (also from jQuery) appears in the attribute manipulation code. Consider this:

```
if (name == "type" &&
    elem.nodeName.toLowerCase()== "input" &&
    elem.parentNode)
  throw "type attribute can't be changed";
```

Internet Explorer doesn't allow us to manipulate the `type` attribute of input elements that are already part of the DOM; attempts to change this attribute result in a proprietary exception being thrown. jQuery came to a middle-ground solution: It disallows *all* attempts to manipulate the `type` attribute on injected input elements in all browsers, throwing a uniform informational exception.

This change to the jQuery code base requires no browser or feature detection; it unifies the API across all browsers. The action still results in an exception, but that exception is uniform across all browser types.

This particular approach could be considered controversial. It purposefully limits the features of the library in all browsers because of a bug that exists in only one. The jQuery team weighed this decision carefully and decided that it was better to have a unified API that works consistently than an API that would break unexpectedly when developing cross-browser code. You might come across situations like this when developing your own reusable code bases, and you'll need to carefully consider whether a limiting approach such as this is appropriate for your audience.

The important thing to remember for these types of code changes is that they provide a solution that works seamlessly across browsers without the need for browser or feature detection, effectively making them immune to changes going forward. You should always strive for solutions that work in this manner, even if the applicable instances are few and far between.

14.3.2 *Feature detection and polyfills*

As we've previously discussed, *feature detection* is a commonly used approach when writing cross-browser code. This approach isn't only simple but also generally effective. It works by determining whether a certain object or object property exists, and if so, assuming that it provides the implied functionality. (In the next section, we'll see what to do about cases where this assumption fails.)

Most commonly, feature detection is used to choose between multiple APIs that provide duplicate pieces of functionality. For example, chapter 10 explored the `find` method, accessible to all arrays, a method that we can use to find the first array item that satisfies a certain condition. Unfortunately, the method is accessible only in browsers that fully support ES6. So what do we do when we're stuck with browsers that still don't support this feature? In general, how do we deal with missing features in the browsers?

The answer is polyfilling! A polyfill is a browser fallback. If a browser doesn't support a particular functionality, we provide our own implementation. For example, the Mozilla Developer Network (MDN) provides polyfills for a wide range of ES6 functionality. Among other things, this includes the JavaScript implementation of the `Array.prototype.find` method (http://mng.bz/d9lU), as shown in the following listing.

> **Listing 14.2   A polyfill for the `Array.prototype.find` method**

```
if (!Array.prototype.find) {                           ◁—— Provides a polyfill only if the
  Array.prototype.find = function(predicate) {              current browser doesn't
    if (this === null) {                                    implement the method
      throw new TypeError('find called on null or undefined');
    }
    if (typeof predicate !== 'function') {
```

Specifies our own implementation

```
        throw new TypeError('predicate must be a function');
    }
    var list = Object(this);
    var length = list.length >>> 0;                        Makes sure that length is a
    var thisArg = arguments[1];                            non-negative integer
    var value;

    for (var i = 0; i < length; i++) {
      value = list[i];
      if (predicate.call(thisArg, value, i, list)) {       Finds the first array
        return value;                                      item that satisfies a
      }                                                    predicate
    }
    return undefined;
  };
}
```

In this example, we first use feature detection to check whether the current browser has built-in support for the find method:

```
if (!Array.prototype.find) {
 ...
}
```

Whenever possible, we should default to the standard way of performing any action. As mentioned before, this will help make our code as future-proof as possible. For this reason, if the browser already supports the method, we do nothing. If we're dealing with a browser that hasn't yet caught up with ES6, we provide our own implementation.

It turns out that the core of the method is straightforward. We loop through the array, calling the passed-in predicate function, which checks whether an array item satisfies our criteria. If it does, we return it.

One interesting technique is presented in this listing:

```
var length = list.length >>> 0;
```

The >>> operator is the *zero-fill right-shift operator*, which shifts the first operand the specified number of bits to the right, while discarding the excess bits. In this case, this operator is used to convert the length property to a non-negative integer. This is done because array indexes in JavaScript should be unsigned integers.

An important use of feature detection is discovering the facilities provided by the browser environment in which the code is executing. This allows us to provide features that use those facilities in our code, or to determine whether we need to provide a fallback.

The following code snippet shows a basic example of detecting the presence of a browser feature by using feature detection, to determine whether we should provide full application functionality or a reduced-experience fallback:

```
if (typeof document !== "undefined" &&
    document.addEventListener &&
    document.querySelector &&
    document.querySelectorAll) {
  // We have enough of an API to work with to build our application
}
else {
  // Provide Fallback
}
```

Here, we test whether

- The browser has a document loaded
- The browser provides a means to bind event handlers
- The browser can find elements based on a selector

Failing any of these tests causes us to resort to a fallback position. What is done in the fallback is up to the expectations of the consumers of the code, and the requirements placed on the code. A few options can be considered:

- We could perform further feature detection to figure out how to provide a reduced experience that still uses some JavaScript.
- We could opt to not execute any JavaScript, falling back to the unscripted HTML on the page.
- We could redirect the user to a plainer version of the site. Google does this with Gmail, for example.

Because feature detection has little overhead (it's just a property/object lookup) and is relatively simple in its implementation, it's a good way to provide basic levels of fallback, both at the API and application levels. It's a good choice for the first line of defense in your reusable code authoring.

14.3.3 Untestable browser issues

Unfortunately, JavaScript and the DOM have several possible problem areas that are either impossible or prohibitively expensive to test for. These situations are fortunately rare, but when we encounter them, it always pays to spend time investigating to see whether there's something we can do about it.

The following sections discuss some known issues that are impossible to test using any conventional JavaScript interactions.

EVENT HANDLER BINDINGS

One of the infuriating lapses in browsers is the inability to programmatically determine whether an event handler has been bound. The browsers don't provide any way of determining whether any functions have been bound to an event listener on an element. There's no way to remove all bound event handlers from an element unless we've maintained references to all bound handlers as we create them.

EVENT FIRING

Another aggravation is determining whether an event will fire. Although it's possible to determine whether a browser supports a means of binding an event, it's *not* possible to know whether a browser will fire an event. This becomes problematic in a couple of places.

First, if a script is loaded dynamically after the page has already loaded, the script may try to bind a listener to wait for the window to load when, in fact, that event already happened. Because there's no way to determine whether the event has already occurred, the code may wind up waiting forever to execute.

The second situation occurs if a script wants to use custom events provided by a browser as an alternative. For example, Internet Explorer provides `mouseenter` and `mouseleave` events, which simplify the process of determining when a user's mouse enters or leaves an element's boundaries. These are frequently used as alternatives to the `mouseover` and `mouseout` events, because they act slightly more intuitively than the standard events. But because there's no way of determining whether these events will fire without first binding the events and waiting for a user interaction against them, it's hard to use them in reusable code.

CSS PROPERTY EFFECTS

Yet another pain point is determining whether changing certain CSS properties affects the presentation. Some CSS properties affect only the visual representation of the display and nothing else; they don't change surrounding elements or affect other properties on the element. Examples are `color`, `backgroundColor`, and `opacity`.

There's no way to programmatically determine whether changing these style properties will generate the effects desired. The only way to verify the impact is through a visual examination of the page.

BROWSER CRASHES

Testing script that causes the browser to crash is another annoyance. Code that causes a browser to crash is especially problematic, because unlike exceptions that can be easily caught and handled, these will always cause the browser to break.

For example, in older versions of Safari (see http://bugs.jquery.com/ticket/1331), creating a regular expression that used Unicode-character ranges would always cause the browser to crash, as in the following example:

```
new RegExp("[\\\w\u0128-\uFFFF*_-]+");
```

The problem with this is that it's not possible to test whether this problem exists, because the test itself will always produce a crash in that older browser.

Additionally, bugs that cause crashes to occur forever become embroiled in difficulty, because although it may be acceptable for JavaScript to be disabled in some segment of the population using your browser, it's never acceptable to outright crash the browser of those users.

INCONGRUOUS APIS

A while back, we saw that jQuery decided to disallow the ability to change the `type` attribute in all browsers because of a bug in Internet Explorer. We *could* test this feature and disable it only in IE, but that would set up an incongruity, as the API would work differently from browser to browser. In these situations, when a bug is so bad that it causes an API to break, the only option is to work around the affected area and provide a different solution.

In addition to impossible-to-test problems, some issues are *possible* to test but are prohibitively difficult to test effectively. Let's look at some of them.

API PERFORMANCE

Sometimes specific APIs are faster or slower in different browsers. When writing reusable and robust code, it's important to try to use the APIs that provide good performance. But it's not always obvious which API that is.

Effectively conducting performance analysis of a feature usually entails throwing a large amount of data at it, and that typically takes a relatively long time. Therefore, it's not something we can do whenever our page is loaded.

Untestable features are a significant hassle that hinder writing reusable JavaScript, but frequently we can work around them with a bit of effort and cleverness. By using alternative techniques, or constructing our APIs so as to obviate these issues in the first place, we'll likely be able to build effective code despite the odds stacked against us.

14.4 Reducing assumptions

Writing cross-browser, reusable code is a battle of assumptions, but by using clever detection and authoring, we can reduce the number of assumptions that we make in our code. When we make assumptions about the code that we write, we stand to encounter problems further down the road.

For example, assuming that an issue or a bug will always exist in a specific browser is a huge and dangerous assumption. Instead, testing for the problem (as we've done throughout this chapter) proves to be much more effective. In our coding, we should always strive to reduce the number of assumptions, effectively reducing the room for error and the probability that something is going to come back and bite us in the behind.

The most common area for making assumptions in JavaScript is in user-agent detection—specifically, analyzing the user agent provided by a browser (`navigator`
`.userAgent`) and using it to make an assumption about how the browser will behave (in other words, browser detection). Unfortunately, most user-agent string analysis proves to be a superb source of future-induced errors. Assuming that a bug, issue, or proprietary feature will always be linked to a specific browser is a recipe for disaster.

But reality intervenes when it comes to minimizing assumptions: It's virtually impossible to remove all of them. At some point, we'll have to assume that a browser will do what it's supposed to do. Figuring out where to strike that balance is completely up to the developer, and it's what "separates the men from the boys," as they say (with apologies to our female readers).

For example, let's reexamine the event-attaching code that we've already seen in this chapter:

```
function bindEvent(element, type, handle) {
  if (element.addEventListener) {
    element.addEventListener(type, handle, false);
  }
  else if (element.attachEvent) {
    element.attachEvent("on" + type, handle);
  }
}
```

Without looking ahead, see if you can spot three assumptions made by this code. Go on, we'll wait. (*Jeopardy* theme plays...)

How'd you do? The preceding code has at least these three assumptions:

- The properties that we're checking are callable functions.
- They're the correct functions, performing the actions that we expect.
- These two methods are the only possible ways of binding an event.

We could easily get rid of the first assumption by adding checks to see whether the properties are, in fact, functions. Tackling the remaining two points is much more difficult.

In this code, we always need to decide how many assumptions are optimal for our requirements, our target audience, and us. Frequently, reducing the number of assumptions also increases the size and complexity of the code base. It's fully possible, and rather easy, to attempt to reduce assumptions to the point of complete insanity, but at some point we have to stop and take stock of what we have, say "good enough," and work from there. Remember that even the least-assuming code is still prone to regressions introduced by a browser.

14.5 Summary

- Although the situation has improved considerably, browsers unfortunately aren't bug-free and usually don't support web standards consistently.
- When writing JavaScript applications, choosing which browsers and platforms to support is an important consideration.
- Because it's not possible to support all combinations, quality should never be sacrificed for coverage!
- The biggest challenges to writing JavaScript code that can be executed in various browsers are bug fixes, regressions, browser bugs, missing features, and external code.
- Reusable cross-browser development involves juggling several factors:
  - *Code size*—Keeping the file size small
  - *Performance overhead*—Keeping the performance level above a palatable minimum
  - *API quality*—Making sure that the APIs work uniformly across browsers

- There's no magic formula for determining the correct balance of these factors.
- The development factors are something that have to be balanced by every developer in their individual development efforts.
- By using smart techniques such as feature detection, we can defend against the numerous directions from which reusable code will be attacked without making any undue sacrifices.

14.6 Exercises

1 What should we take into account when deciding which browsers to support?
2 Explain the problem of greedy IDs.
3 What is feature detection?
4 What is a browser polyfill?

<div style="background-color:#f0f0d0;padding:1em;">

This appendix covers

- Template literals
- Destructuring
- Object literal enhancements

</div>

This appendix covers some of the "smaller" ES6 features that don't fit neatly into the previous chapters. *Template literals* enable string interpolation and multiline strings, *destructuring* enables us to easily extract data from objects and arrays, and *enhanced object literals* improve dealings with, well, object literals.

Template literals

Template literals are a new ES6 feature that make manipulating strings much more pleasant than before. Just think back; how many times have you been forced to write something as ugly as this?

```
const ninja = {
  name: "Yoshi",
  action: "subterfuge"
};

const concatMessage = "Name: " + ninja.name + " "
                   + "Action: " + ninja.action;
```

In this example, we have to construct a string with data dynamically inserted. To achieve this, we have to resort to some messy concatenations. But not anymore! In ES6 we can achieve the same result with template literals; just take a look at the following listing.

Listing A.1   Template literals

```
const ninja = {
  name: "Yoshi",
  action: "subterfuge"
};

const concatMessage = "Name: " + ninja.name + " "
                   + "Action: " + ninja.action;
const templateMessage = `Name: ${ninja.name} Action: ${ninja.action}`;  ◁──

assert(concatMessage === templateMessage,
      "Our messages match");
```

Uses backticks to create template literals that can contain JavaScript expressions encapsulated in ${}

As you can see, ES6 provides a new type of string that uses backticks (`` ` ``), a string that can contain placeholders, denoted with the ${} syntax. Within these placeholders, we can place any JavaScript expression: a simple variable, an object property access (as we did with `ninja.action`), and even function calls.

When a template string gets evaluated, the placeholders are replaced with the result of evaluating the JavaScript expression contained within those placeholders.

In addition, template literals aren't limited to a single line (as are standard double and single quoted ones), and there's nothing stopping us from making them multi-line, as shown in the following listing.

Listing A.2   Multiline template literals

```
const name = "Yoshi", action = "subterfuge";
const multilineString =
`Name: ${name}
 Yoshi: ${action}`;
```

Template strings aren't limited to a single line.

Now that we've given you a short intro to template literals, let's look at another ES6 feature: destructuring.

Destructuring

Destructuring allows us to easily extract data from objects and arrays by using patterns. For example, imagine that you have an object whose properties you want to assign to a couple of variables, as in the following listing.

Listing A.3   Destructuring objects

```
const ninja = { name:"Yoshi", action: "skulk", weapon: "shuriken"};

const nameOld = ninja.name;            Old way: we have to explicitly
const actionOld = ninja.action;        assign each object property
const weaponOld = ninja.weapon;        to a variable.

const {name, action, weapon} = ninja;        ◁──  Object destructuring: we
                                                  can assign each property
assert(name === "Yoshi", "Our ninja Yoshi");      to a variable of the same
assert(action === "skulk", "is skulking");        name, all at once.
assert(weapon === "shuriken", "with a shuriken");

const {name: myName, action: myAction, weapon: myWeapon} = ninja;    ◁──

assert(myName === "Yoshi", "Our ninja Yoshi");      If necessary, we can explicitly
assert(myAction === "skulk", "is skulking");        name the variables to which
assert(myWeapon === "shuriken", "with a shuriken");      we want to assign values.
```

As listing A.3 shows, with object destructuring, we can easily extract multiple variables from an object literal, all in one go. Consider the following statement:

```
const {name, action, weapon} = ninja;
```

This creates three new variables (`name`, `action`, and `weapon`) whose values are the values of the matching properties of the object on the right-hand side of the statement (`ninja.name`, `ninja.action`, and `ninja.weapon`, respectively).

When we don't want to use the names of object properties, we can fine-tune them, as in the following statement:

```
const {name: myName, action: myAction, weapon: myWeapon} = ninja;
```

Here we create three variables (`myName`, `myAction`, and `myWeapon`) and assign to them values of the specified object properties.

Earlier, we mentioned that we can also destructure arrays, as arrays are just a special kind of object. Take a look at the following listing.

Listing A.4   Destructuring arrays

```
const ninjas = ["Yoshi", "Kuma", "Hattori"];
const [firstNinja, secondNinja, thirdNinja] = ninjas;        ◁──  Array items are, in
                                                                  order, assigned to
assert(firstNinja === "Yoshi", "Yoshi is our first ninja");       specified variables.
assert(secondNinja === "Kuma", "Kuma the second one");
```

```
assert(thirdNinja === "Hattori", "And Hattorithe third");

const [, , third] = ninjas;
assert(third === "Hattori", "We can skip items");

const [first, ...remaining] = ninjas;
assert(first === "Yoshi", "Yoshi is again our first ninja");
assert(remaining.length === 2, "There are two remaining ninjas");
assert(remaining[0] === "Kuma", "Kuma is the first remaining ninja");
assert(remaining[1] === "Hattori", "Hattori the second remaining ninja");
```

> **We can skip certain array items.**

> **We can capture trailing items.**

Destructing arrays is slightly different from destructuring objects, primarily in syntax, because the variables are wrapped in brackets (as opposed to braces, which are used for object destructuring), as shown in the following fragment:

```
const [firstNinja, secondNinja, thirdNinja] = ninjas;
```

In this case, Yoshi, the first ninja, is assigned to the variable firstNinja. Kuma is assigned to the variable secondNinja. Hattori is assigned to the variable thirdNinja.

Array destructuring also has some advanced uses. For example, if we want to skip certain items, we can omit variable names, while keeping commas, as in the following statement:

```
const [, , third] = ninjas;
```

In this case, the first two ninjas will be ignored, while the value of the third ninja, Hattori, will be assigned to the variable third.

In addition, we can extract only certain items, while assigning remaining items to a new array:

```
const [first, ...remaining] = ninjas;
```

The first item, Yoshi, is assigned to the variable first, and the remaining ninjas, Kuma and Hattori, are assigned to the new array, remaining. Notice that in this case, the remaining items are marked in the same way as the rest parameters (the . . . operator).

Enhanced object literals

One of the great things about JavaScript is its ease of creating objects with object literals: We define a couple of properties and wrap them within curly braces, and voilà, we've created a new object. In ES6, the object literal syntax has gained some new extensions. Let's look at an example. Say we want to create a ninja object, and assign to it a property based on the value of a variable that's in scope, a property whose name is dynamically computed, and a method, as shown in the following listing.

Listing A.5   Enhanced object literals

```
constname = "Yoshi";
const oldNinja = {
  name: name,
```

> **Creates a property with the same name as a variable in scope and assigns the value of that variable to it**

```
  getName: function(){
    return this.name;
  }
};
```
Defines a method on an object

```
oldNinja["old" + name] = true;
```
Creates a property whose name is dynamically calculated

```
assert(oldNinja.name === "Yoshi", "Yoshi here");
assert(typeof oldNinja.getName === "function", "with a method");
assert("oldYoshi" in oldNinja, "and a dynamic property");


constnewNinja = {
  name,
  getName(){
    return this.name;
  },
  ["new" + name]: true
};
```
Property value shorthand syntax; assigns the value of the same named variable to the property

Method definition shorthand; there's no need to add a colon and the function keyword. Using parentheses after the property name signals that we're dealing with a method.

```
assert(newNinja.name === "Yoshi", "Yoshi here, again");
assert(typeof newNinja.getName === "function", "with a method");
assert("newYoshi" in newNinja, "and a dynamic property");
```

A computed property name

This example starts by creating an `oldNinja` object using the old pre-ES6 object literal syntax:

```
const name = "Yoshi";
const oldNinja = {
  name: name,
  getName: function(){
    return this.name;
  }
};
oldNinja["old" + name] = true;
```

We contrast this with enhanced object literals that achieve exactly the same effect, with less syntactic clutter:

```
const newNinja = {
  name,
  getName(){
    return this.name;
  },
  ["new" + name]: true
};
```

This completes our exploration of important, new concepts introduced by ES6.

appendix B
Arming with
testing and debugging

This appendix covers

- Tools for debugging JavaScript code
- Techniques for generating tests
- Building a test suite
- Surveying some of the popular testing frameworks

This appendix presents some fundamental techniques in developing client-side web applications: debugging and testing. Constructing effective test suites for our code is always important. After all, if we don't test our code, how do we know that it does what we intend? Testing gives us a means to ensure that our code not only runs, but runs *correctly*.

Moreover, as important as a solid testing strategy is for *all* code, it can be crucial when external factors have the potential to affect the operation of our code, which

is *exactly* the case we're faced with in cross-browser JavaScript development. Not only do we have the typical problems of ensuring the quality of the code (especially when dealing with multiple developers working on a single code base) and guarding against regressions that could break portions of an API (generic problems that all program-mers need to deal with), but we also have the problem of determining whether our code works in all the browsers that we choose to support.

In this chapter, we'll look at tools and techniques for debugging JavaScript code, generating tests based on those results, and constructing a test suite to reliably run those tests. Let's get started.

Web developer tools

For a long time, the development of JavaScript applications was hindered by the lack of a basic debugging infrastructure. The only way to debug JavaScript code was to scat-ter `alert` statements that would notify us about the value of the alerted expression, all around the code that was acting strangely. As you might imagine, this made debug-ging (hardly ever a fun activity) even more difficult.

Luckily, Firebug, an extension to Firefox, was developed in 2007. Firebug holds a special place in the hearts of many web developers, because it was the first tool that provided a debugging experience that closely matched debugging in state-of-the-art integrated development environments (IDEs), such as Visual Studio or Eclipse. In addition, Firebug has inspired the development of similar developer tools for all major browsers: F12 Developer Tools, included in Internet Explorer and Micro-soft Edge; WebKit Inspector, included in Safari; Firefox Developer Tools, included in Firefox; and Chrome DevTools included in Chrome and Opera. Let's explore them a bit.

FIREBUG

Firebug, the first advanced web application debugging tool, is available exclusively for Firefox, and is accessed by pressing the F12 key (or by right-clicking anywhere on the page and selecting Inspect Element with Firebug). You can install Firebug by opening the page in Firefox (https://getfirebug.com/) and following the instructions. Figure B.1 shows Firebug.

Firebug offers advanced debugging functionalities, some of which it has even pio-neered. For example, we can easily explore the current state of the DOM by using the HTML pane (the pane shown in figure B.1), run custom JavaScript code within the context of the current page by using the console (the bottom of figure B.1), explore the state of our JavaScript code by using the Script pane, and even explore network communications from the Net pane.

FIREFOX DEVELOPER TOOLS

In addition to Firebug, if you're a Firefox user, you can use the built-in Firefox DevTools, shown in figure B.2. As you can see, the general look and feel of Firefox developer tools is similar to Firebug (apart from some minor layout and label

**Debug
JavaScript code.**

**Explore network
communication.**

**Run custom JavaScript
code in the console.**

**The current state
of the DOM.**

**Additional node
properties.**
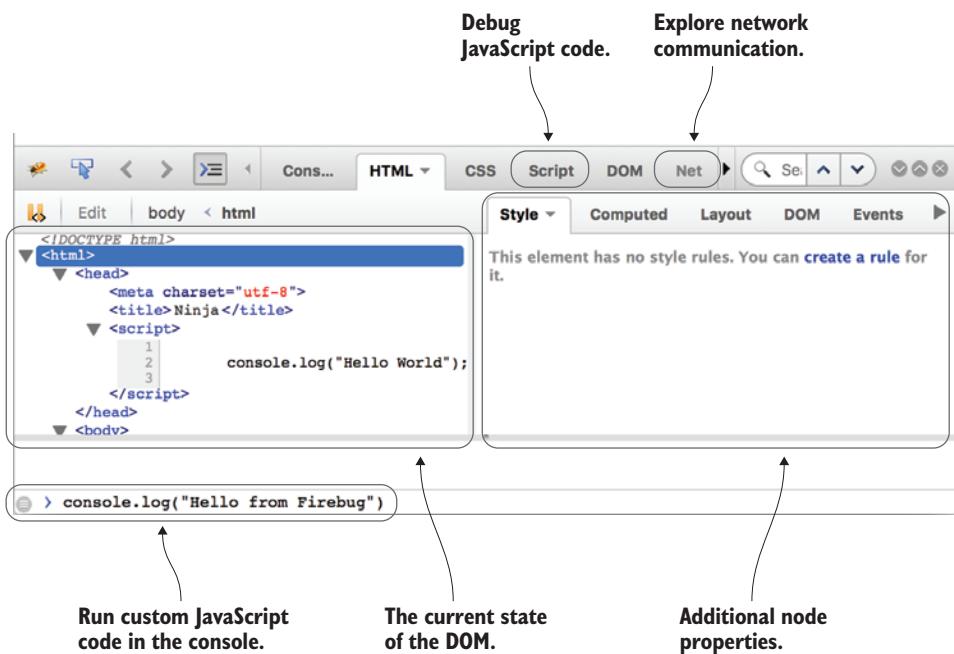
Figure B.1   Firebug, available only in Firefox, was the first advanced debugging tool for web applications.

**Inspect the current
state of the DOM.**

**Debug
JavaScript code.**

**Measure
performance.**

**Explore network
communication.**

**Responsive
Design mode.**

**Run custom JavaScript
code in the console.**
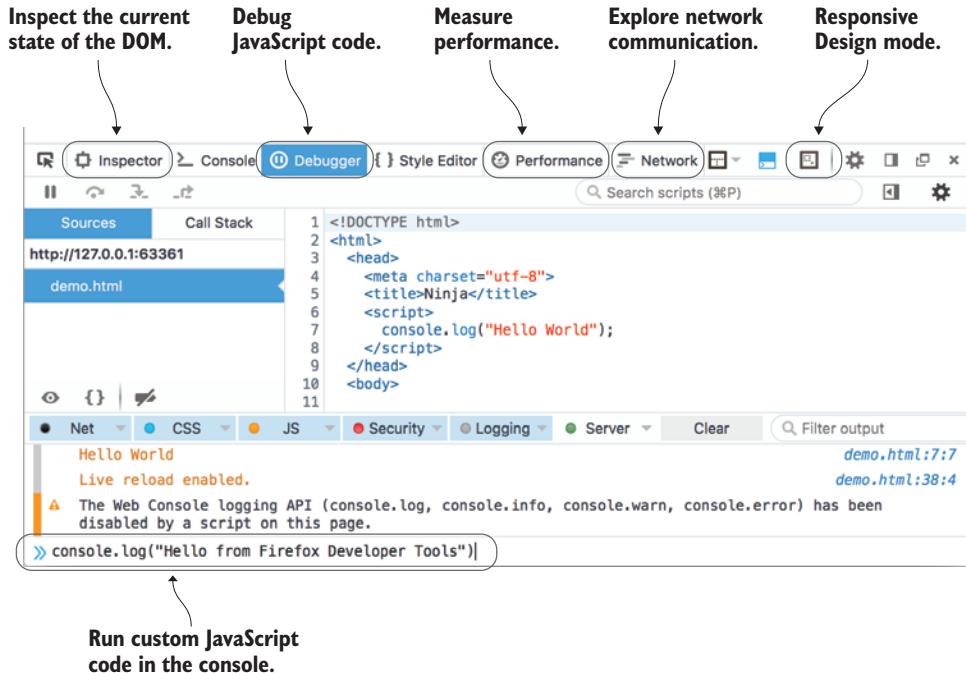
Figure B.2   Firefox developer tools, built into Firefox, offer all the Firebug features and then some.

differences; for example, the HTML pane from Firebug is called Inspector in Firefox developer tools).

Firefox developer tools are built by the Mozilla team, which has taken advantage of this close integration with Firefox by bringing in some additional useful features. The Performance pane, for instance, provides detailed insight about the performance of our web applications. In addition, Firefox developer tools are built with the modern web in mind. For example, they offer Responsive Design mode, which helps us explore the look and feel of our web applications across different screen sizes—which is something we have to be careful about, because nowadays users access web applications not only from their PCs, but also from mobile devices, tablets, and even TVs.

F12 DEVELOPER TOOLS

If you're in the Internet Explorer (IE) camp, you'll be happy to know that IE and Microsoft Edge (the successor to IE) offer their own developer tools, the F12 developer tools. (Quickly, try to guess which key toggles them on and off.) These tools are shown in figure B.3.

Again, notice the similarities between the F12 developer tools and Firefox's developer tools (with only slight differences in labels). The F12 tools also enable us to explore the current state of the DOM (the DOM Explorer pane, figure B.3), run custom JavaScript code through the console, debug our JavaScript code (the Debugger pane), analyze the network traffic (Network), deal with responsive design (UI Responsiveness), and analyze performance and memory consumption (Profiler and Memory).
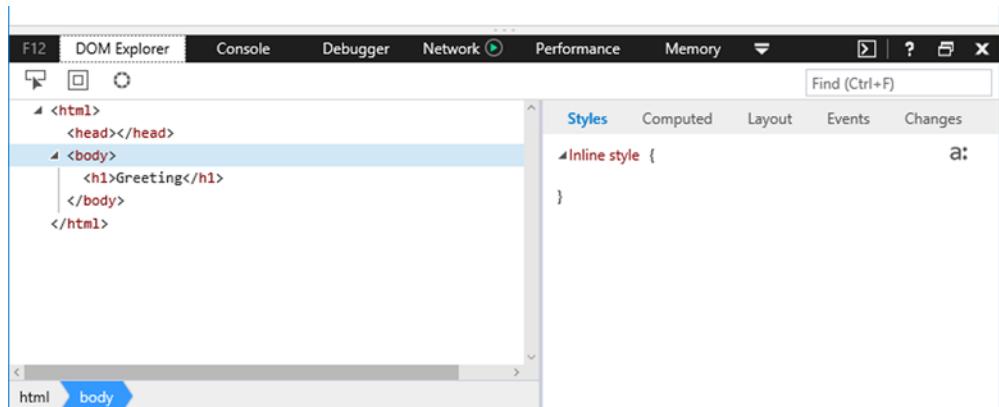
Figure B.3   **F12 developer tools (toggled by pressing F12) are available in Internet Explorer and Edge.**

WEBKIT INSPECTOR

If you're an OS X user, you can use WebKit Inspector, which is offered by Safari, as shown in figure B.4. Although the UI of Safari's WebKit Inspector is slightly different
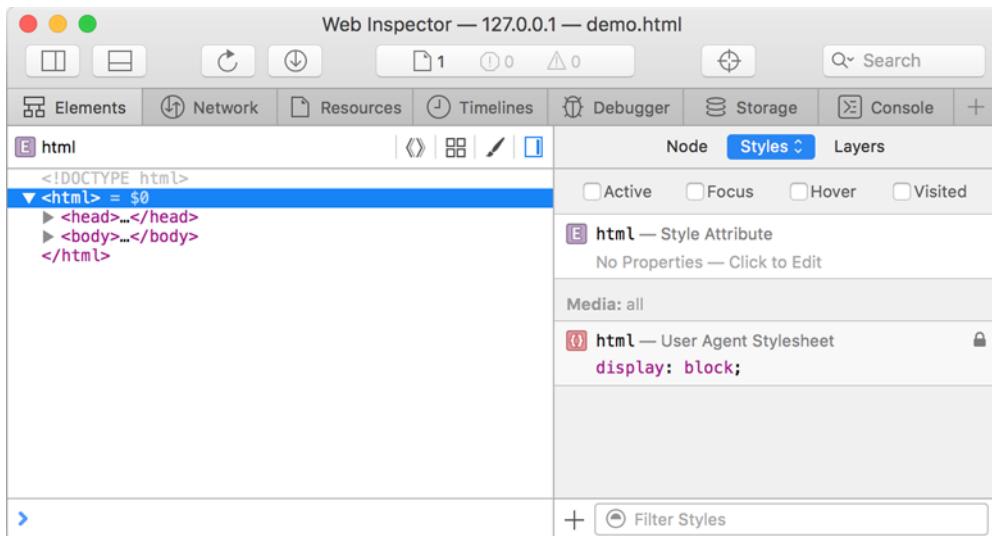
Figure B.4   WebKit Inspector, available in Safari

from that of F12 developer tools or Firefox's developer tools, rest assured that the WebKit Inspector also supports all important debugging features.

CHROME DEVTOOLS

We'll complete our little survey of developer tools with Chrome DevTools—in our opinion, the current flagship of web application developer tools that's been driving a lot of innovations lately. As you can see in figure B.5, the basic UI and features are similar to the rest of the developer tools.
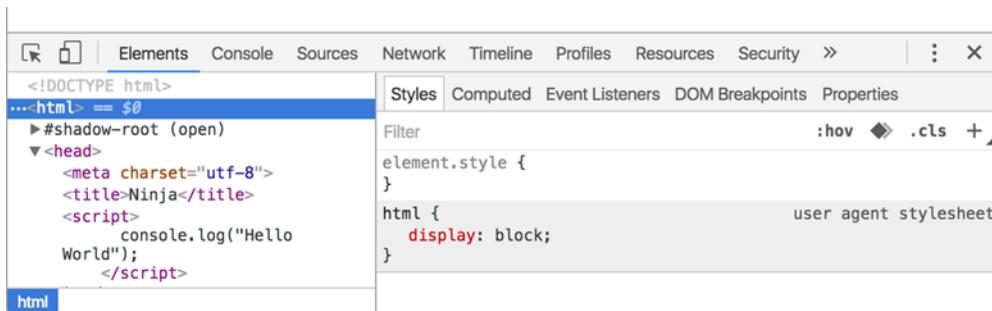
Figure B.5   Chrome DevTools, available in Chrome and Opera

Throughout this book, we've used Chrome DevTools, for the sake of convention. But as you've seen throughout this section, most developer tools offer similar features (and if one of them offers something new, the others catch up quickly). You can just as easily use the developer tools offered by your browser of choice.

Now that you've had an introduction to the tools you can use for debugging code, let's explore some debugging techniques.

Debugging code

A significant portion of time developing software is spent on removing annoying bugs. Although this can sometimes be interesting, almost like solving a whodunit mystery, typically we'll want our code working correctly and bug-free as soon as possible.

Debugging JavaScript has two important aspects:

- *Logging*, which prints out what's going on as our code is running
- *Breakpoints*, which allow us to temporarily pause the execution of our code and explore the current state of the application

They're both useful for answering the important question, "What's going on in my code?" but each tackles it from a different angle. Let's start by looking at logging.

Logging

Logging statements are used for outputting messages during program execution, without impeding the normal flow of the program. When we add logging statements to our code (for example, by using the `console.log` method), we benefit from seeing messages in the browser's console. For example, if we want to know the value of a variable named x at certain points of program execution, we might write something like the following listing.

> **Listing B.1   Logging the value of variable `x` at various points of program execution**

```
   <!DOCTYPE html>
1: <html>
2:   <head>
3:    <title>Logging</title>
4:    <script>
5:      var x = 213;
6:      console.log("The value of x is: ", x);
7:
8:      x = "Hello " + "World";
9:      console.log("The value of x is now:", x);
10:   </script>
11:  </head>
12:  <body></body>
13:</html>
```

Figure B.6 shows the result of executing this code in the Chrome browser with the JavaScript console enabled.

**Open the JavaScript
console tab.**

**Shows the file and the line
from which the log was made**

**Result of executing
console.log("The value of x is:", x);**

**Result of executing
console.log("The value of x is now:", x);**
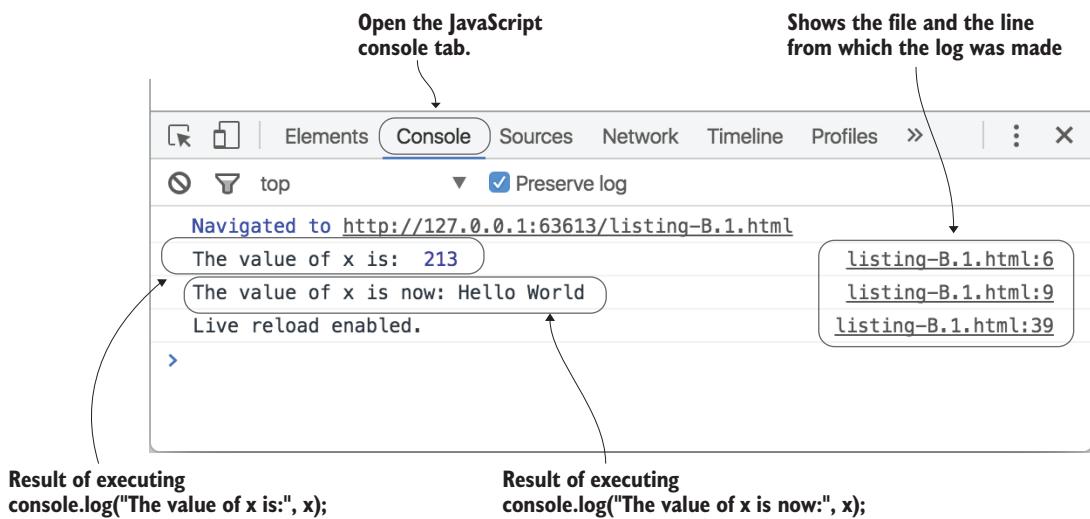
Figure B.6   Logging lets us see the state of our code as it's running. In this case, we can see that the value of
213 is logged from line 6, and the value of "Hello World" from line 9 of listing B.1. All developer tools, including
Chrome DevTools shown here, have a Console tab for logging purposes.

As you can see, the browser logs the messages directly to the JavaScript console, show-
ing both the logged message and the line in which the message was logged.

This is a simple example of logging a value of a variable at different points of pro-
gram execution. But in general, you can use logging to explore various facets of your
running applications, such as the execution of important functions, the change of an
important object property, or the occurrence of a particular event.

Logging is all well and good for seeing the state of things while the code is run-
ning, but sometimes we'll want to stop the action and take a look around. That's
where breakpoints come in.

Breakpoints

Using *breakpoints* can be more complex than logging, but they possess a notable advantage:
They halt the execution of a script at a specific line of code, pausing the browser. This
allows us to leisurely investigate the state of all sorts of things at the point of the break.

Let's say that we have a page that logs a greeting to a famous ninja, as shown in the
following listing.

Listing B.2   A simple "greet a ninja" page

```
<!DOCTYPE html>
<html>
  <head>
   <title>Ninja greeting</title>
   <script>
    function logGreeting(name){
      console.log("Greetings to the great " + name);
```

```
  }
  var ninja = "Hattori Hanzo";
  logGreeting(ninja);                    ⟵──── Line where we'll break
 </script>
 </head>
 <body>
 </body>
</html>
```

Say we set a breakpoint by using the Chrome DevTools on the annotated line that calls the `logGreeting` function in listing B.2 (by clicking the line number gutter in the Debugger pane) and refresh the page to cause the code to execute. The debugger would then stop the execution at that line and show us the display in figure B.7.

The pane on the right shows the state of the application within which our code is running, including the value of the `ninja` variable (`Hattori Hanzo`). The debugger breaks on a line *before* the breakpointed line is executed; in this example, the call to the `logGreeting` function has yet to be executed.

STEPPING INTO A FUNCTION

If we're trying to debug a problem with our `logGreeting` function, we might want to *step into* that function to see what's going on inside it. While our execution is paused on the `logGreeting` call (with a breakpoint that we've previously set), we click the Step Into button (shown as an arrow pointing to a dot in most debuggers) or press F11, which will cause the debugger to execute up to the first line of our `logGreeting` function. Figure B.8 shows the result.

1. Shows the current code of our web page

2. Create a breakpoint by clicking in the line gutter.

3. When the page is loaded, execution will stop at the breakpoint and highlight the line next to be executed.

4. The current state of our application (ninja: "Hattori Hanzo")
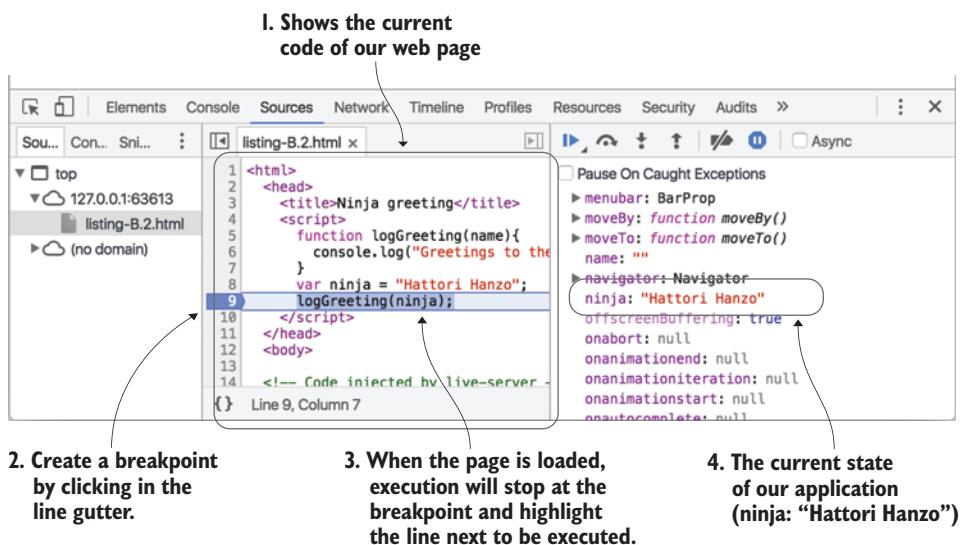
Figure B.7 When we set a breakpoint on a line of code (by clicking the line gutter) and load the page, the browser will stop executing JavaScript code before that line is executed. Then you can leisurely explore the current state of the application in the pane on the right.

Note that the look of Chrome DevTools has changed a bit (when compared to figure B.7) to allow us to poke around the application state in which the `logGreeting` function executes. For example, now we can easily explore the local variables of our `logGreeting` function and see that we have a `name` variable with the value `Hattori Hanzo` (the variable values are even shown inline, with the source code on the left). Also notice that in the upper-right corner is a Call Stack pane, which shows that we're currently within the `logGreeting` function, which was called by global code.

Step Over and Step Out

In addition to the Step Into command, we can use Step Over and Step Out.

The Step Over command executes our code line by line. If the code in the executed line contains a function call, the debugger steps over the function (the function will be executed, but the debugger won't jump into its code).

If we've paused the execution of a function, clicking the Step Out button will execute the code to the end of the function, and the debugger will again pause right after the execution has left that function.

The line in which the execution is currently paused

The current values of local variables

The list of active breakpoints

Variable values are also shown inline!

The Call Stack (marking that we're currently in the logGreeting function)

Clicking the Step Into button will execute up to the first line of a function.
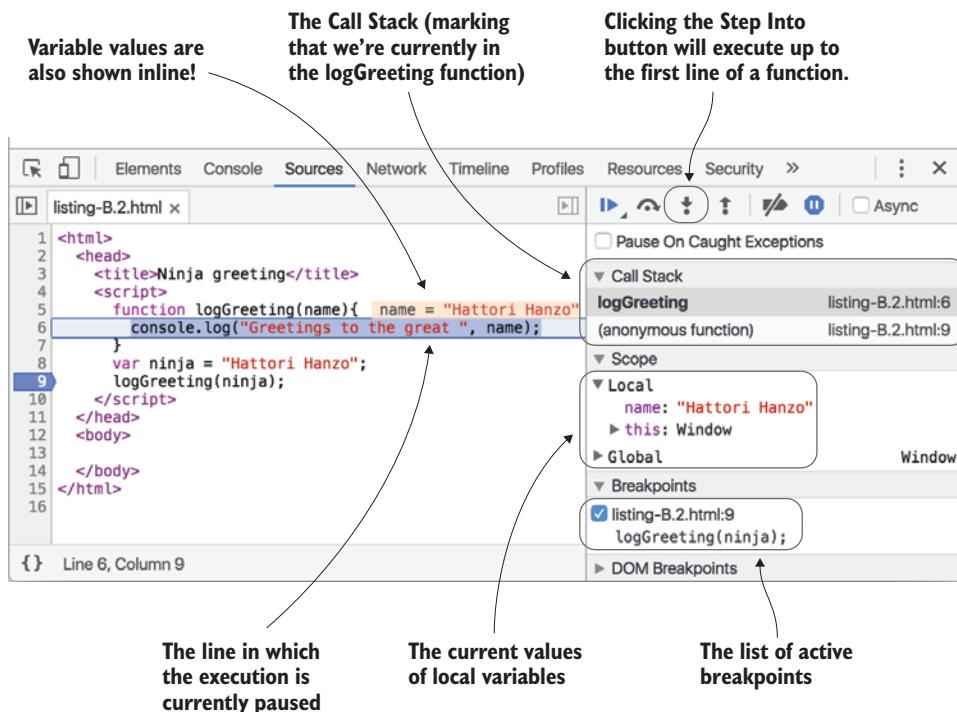
Figure B.8   Stepping into a function lets us see the new state in which the function is executed. We can explore the current position by studying the Call Stack and the current values of local variables.

Standard breakpoints cause the debugger to stop the application execution every time a debugger reaches that particular point in program execution. In certain cases, this can be tiring. Consider the following listing.

Listing B.3   Counting Ninjas and conditional breakpoints

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      for(var i = 0; i < 100; i++){
        console.log("Ninjas: " + i);
      }
    </script>
  </head>
  <body>
  </body>
</html>
```

> What if we want to explore the application state when counting the 50th ninja? Should we have to tediously go through the first 49?

Imagine that we want to explore the state of the application when counting the 50th ninja. How tiring would it be to have to visit all 49 ninjas before finally reaching the one we want?

Welcome to conditional breakpoints! Unlike traditional breakpoints, which halt every time the breakpointed line is executed, a *conditional breakpoints* causes the debugger to break only if an expression associated with the conditional breakpoints is satisfied. You can set a conditional breakpoint by right-clicking in the line-number gutter and choosing Add (see figure B.9 for how it's done in Chrome).

By associating the expression: `i == 49` with a conditional breakpoint, the debugger will halt only when that condition is satisfied. In that way, we can jump immediately to the point in the application execution that we're interested in, and ignore the less interesting ones.

So far, you've seen how to use various developer tools from different browsers in order to debug our code with logging and breakpoints. These are all great tools that
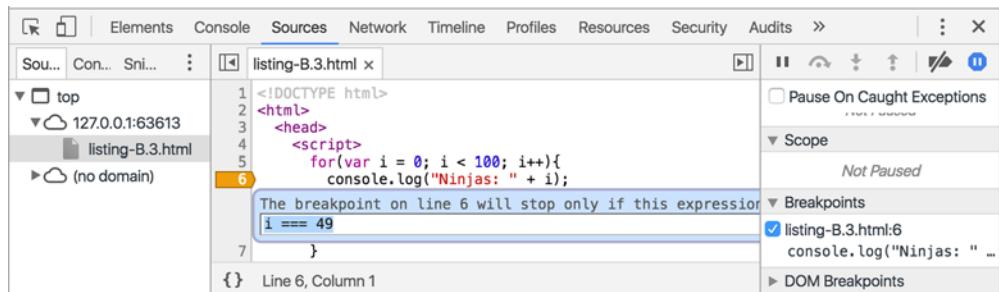
Figure B.9   Right-click in the line-number margin to set a breakpoint as conditional; notice that these are shown in a different color, usually orange.

help us locate specific bugs and achieve a better understanding of the execution of a particular application. But in addition to this, we want to have an infrastructure in place that will help us detect bugs as soon as possible. This can be achieved with testing.

Creating tests

Robert Frost wrote that "good fences make good neighbors," but in the world of web applications, and indeed any programming discipline, good tests make good code. Note the emphasis on the word *good*. It's possible to have an extensive test suite that doesn't help the quality of our code one iota if the tests are poorly constructed.

Good tests exhibit three important characteristics:

- *Repeatability*—Our test results should be highly reproducible. Tests that run repeatedly should always produce the exact same results. If test results are non-deterministic, how would we know which results are valid and which are invalid? Additionally, reproducibility ensures that our tests aren't dependent on external factors, such as network or CPU loads.
- *Simplicity*—Our tests should focus on testing *one* thing. We should strive to remove as much HTML markup, CSS, or JavaScript as we can without disrupting the intent of the test case. The more we remove, the greater the likelihood that the test case will be influenced by only the specific code that we're testing.
- *Independence*—Our tests should execute in isolation. We must avoid making the results from one test dependent on another. Breaking tests into the smallest possible units will help us determine the exact source of a bug when an error occurs.

We can use various approaches to construct tests. The two primary approaches are deconstructive and constructive:

- *Deconstructive test cases*—Existing code is whittled down (deconstructed) to isolate a problem, eliminating anything that's not germane to the issue. This helps achieve the three characteristics listed previously. We might start with a complete website, but after removing extra markup, CSS, and JavaScript, we'll arrive at a smaller case that reproduces the problem.
- *Constructive test cases*—We start from a known good, reduced case and build up until we're able to reproduce the bug in question. To use this style of testing, we'll need a couple of simple test files from which to build up tests, and a way to generate these new tests with a clean copy of our code.

Let's look at an example of constructive testing.

When creating reduced test cases, we can start with a few HTML files with minimum functionality already included in them. We might even have different starting files for various functional areas; for example, one for DOM manipulation, one for Ajax tests, one for animations, and so on.

For example, the following listing shows a simple DOM test case used to test jQuery.

Listing B.4   A reduced DOM test case for jQuery

```
<style>
  #test { width: 100px; height: 100px; background: red; }
</style>
<div id="test"></div>
<script src="dist/jquery.js"></script>
<script>
  $(document).ready(function() {
    $("#test").append("test");
  });
</script>
```

Another alternative is to use a prebuilt service designed for creating simple test cases, for example JSFiddle (http://jsfiddle.net/), CodePen (http://codepen.io/), or JS Bin (http://jsbin.com/). All have similar functionality; they allow us to build test cases that become available at a unique URL. (And you can even include copies of popular libraries.) An example in JSFiddle is shown in figure B.10.
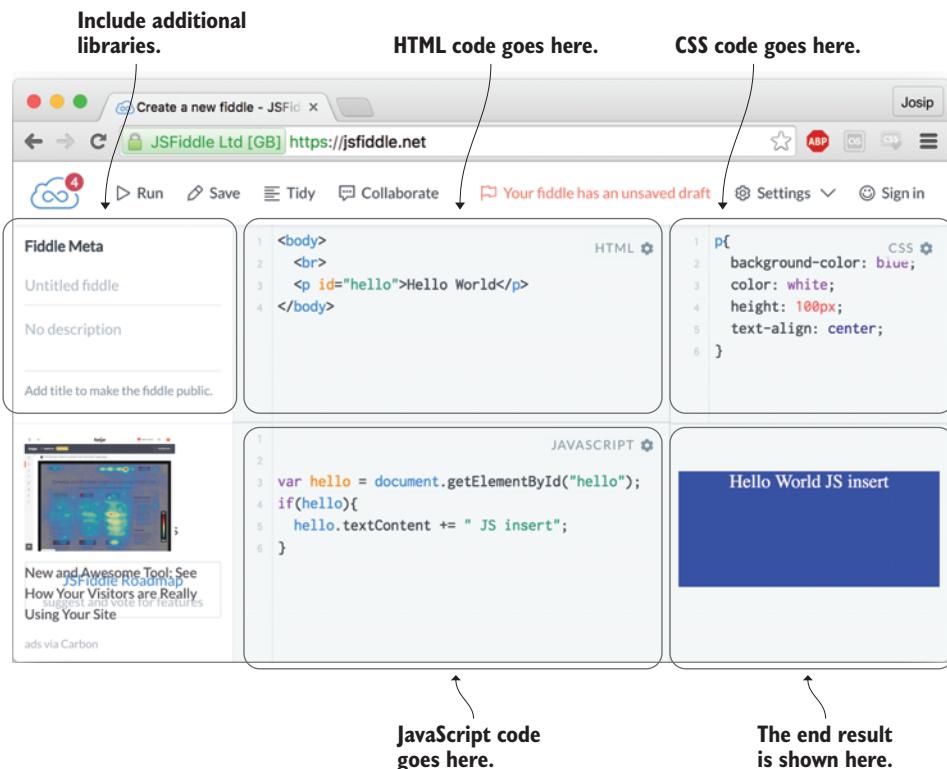
Figure B.10   JSFiddle enables us to test combinations of HTML, CSS, and JavaScript snippets in a sandbox to see if everything works as intended.

Using JSFiddle (or similar tools) is all nice and practical when we have to do a quick test of a certain concept, especially because you can easily share it with other people, and maybe even get some useful feedback. Unfortunately, running such tests requires that you manually open the test and check its result, which might be fine if you have only a couple of tests, but normally we should have lots and lots of tests that check every nook and cranny of our code. For this reason, we want to automate our tests as much as possible. Let's look at how to achieve that.

The fundamentals of a testing framework

The primary purpose of a testing framework is to allow us to specify individual tests that can be wrapped into a single unit, so that they can be run in bulk, providing a single resource that can be run easily and repeatedly.

To better understand how a testing framework works, it makes sense to look at how it's constructed. Perhaps surprisingly, JavaScript testing frameworks are easy to construct.

You'd have to ask, though, "Why would I want to build a new testing framework?" For most cases, writing your own JavaScript testing framework isn't necessary, because many good-quality ones are already available (as you'll soon see). But building your own test framework can serve as a good learning experience.

The assertion

The core of a unit-testing framework is its assertion method, customarily named `assert`. This method usually takes a *value*—an expression whose premise is *asserted*—and a description of the purpose of the assertion. If the value evaluates to `true`, the assertion passes; otherwise, it's considered a failure. The associated message is usually logged with an appropriate pass/fail indicator.

A simple implementation of this concept can be seen in the following listing.

Listing B.5   A simple implementation of a JavaScript assertion

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Suite</title>
    <script>
      function assert(value, desc) {
        var li = document.createElement("li");
        li.className = value ? "pass" : "fail";           Defines the assert
        li.appendChild(document.createTextNode(desc));    method
        document.getElementById("results").appendChild(li);
      }
      window.onload = function() {
        assert(true, "The test suite is running.");        Executes tests
        assert(false, "Fail!");                            using assertions
      };
    </script>
    <style>
```

```
      #results li.pass { color: green; }          Defines styles
      #results li.fail { color: red; }            for results
    </style>
  </head>
  <body>
    <ul id="results"></ul>              ◁——— Holds test results
  </body>
</html>
```

The function named `assert` is almost surprisingly straightforward. It creates a new `<li>` element containing the description, assigns a class named `pass` or `fail`, depending on the value of the assertion parameter (`value`), and appends the new element to a list element in the document body.

The test suite consists of two trivial tests: one that will always succeed, and one that will always fail:

```
assert(true, "The test suite is running."); //Will always pass
assert(false, "Fail!"); //Will always fail
```

Style rules for the `pass` and `fail` classes visually indicate success or failure using colors.

The result of running our test suite in Chrome is shown in figure B.11.

> **TIP** If you're looking for something quick, you can use the built-in `console.assert()` method (see figure B.12).

Now that we've built our own rudimentary testing framework, let's meet some of the widely available, more popular testing frameworks.
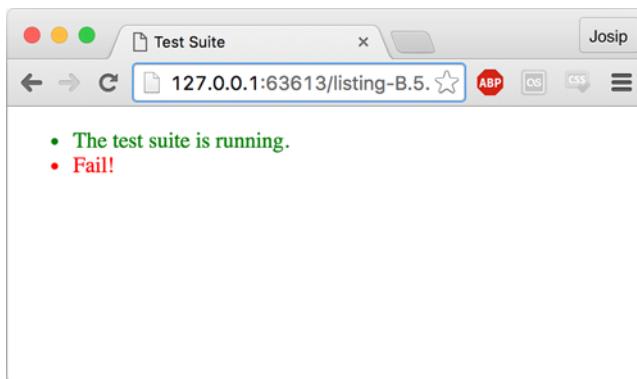
Figure B.11 **The result of running our first test suite**

**Assertion passes; nothing
is logged in the console.**

**Assertion fails; a message
is logged in the console.**
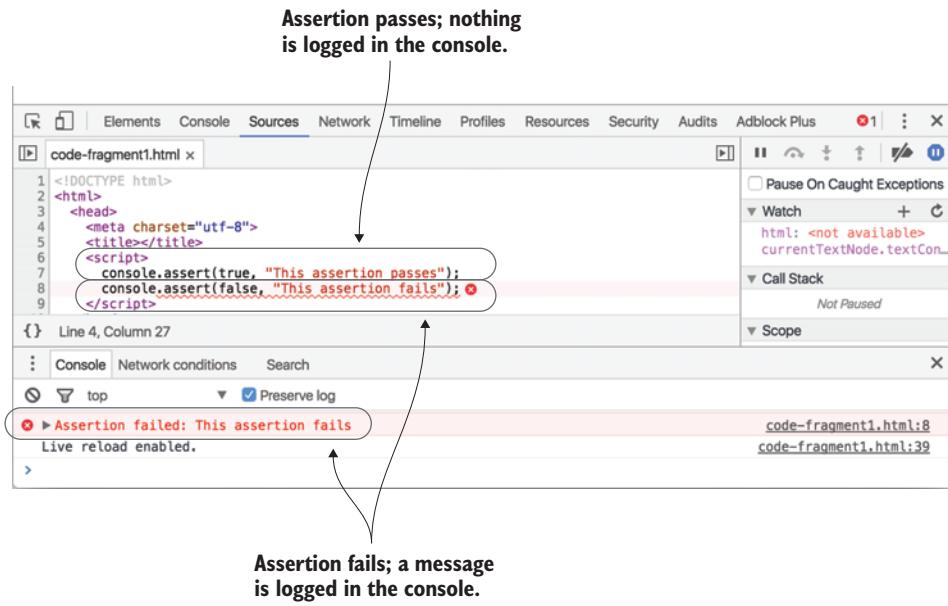
**Figure B.12   You can use the built-in `console.assert` as a quick way to test code. The fail
message is logged to the console only if an assertion fails.**

Popular testing frameworks

A test framework should be a fundamental part of your development workflow, so
you should pick a framework that works particularly well for your coding style and
your code base. A JavaScript testing framework should serve a single need: display-
ing the results of the tests, and making it easy to determine which tests have passed
or failed. Testing frameworks can help us reach that goal without having to worry
about anything other than creating the tests and organizing them into collections
called *test suites*.

There are several features that we might want to look for in a JavaScript unit-testing
framework, depending on the needs of the tests. Some of these features include the
following:

- The ability to simulate browser behavior (clicks, key presses, and so on)
- Interactive control of tests (pausing and resuming tests)
- Handling asynchronous test time-outs
- The ability to filter which tests are to be executed

Let's meet the two currently most popular testing frameworks: QUnit and Jasmine.

QUNIT

QUnit is the unit-testing framework originally built to test jQuery. It has since
expanded beyond its initial goals and is now a standalone unit-testing framework.

QUnit is primarily designed to be a simple solution to unit testing, providing a minimal but easy-to-use API. QUnit's distinguishing features are as follows:

- Simple API
- Supports asynchronous testing
- Not limited to jQuery or jQuery-using code
- Especially well-suited for regression testing

Let's look at a QUnit test example in the following listing that tests whether we've developed a function that accurately says "Hi" to a ninja.

Listing B.6   QUnit test example

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="qunit/qunit-git.css"/>      Includes QUnit
 <script src="qunit/qunit-git.js"></script>                  code and styles
  </head>
  <body>                                    Creates an HTML element that
    <div id="qunit"></div>                  QUnit fills with test results
 <script>
  function sayHiToNinja(ninja) {            Declares the function
    return "Hi " + ninja;                   that we want to test
  }
                                                        Specifies a
  QUnit.test("Ninja hello test", function(assert){        QUnit test case
    assert.ok( sayHiToNinja("Hatori") == "Hi Hatori", "Passed");
    assert.ok( false, "Failed");
  });                                       Tests a passing assertion
</script>
 </body>
</html>
```
Tests a failing assertion

When you open this example in a browser, you should get the results shown in figure B.13, with one passing assertion from executing the line `sayHiToNinja("Hatori")`, and one failing assertion from `assert.ok(false, "Failed")`.

More information on QUnit can be found at http://qunitjs.com/.

JASMINE

Jasmine is another popular testing framework, built on slightly different foundations than QUnit. The principal parts of the framework are as follows:

- The `describe` function, which describes test suites
- The `it` function, which specifies individual tests
- The `expect` function, which checks individual assertions
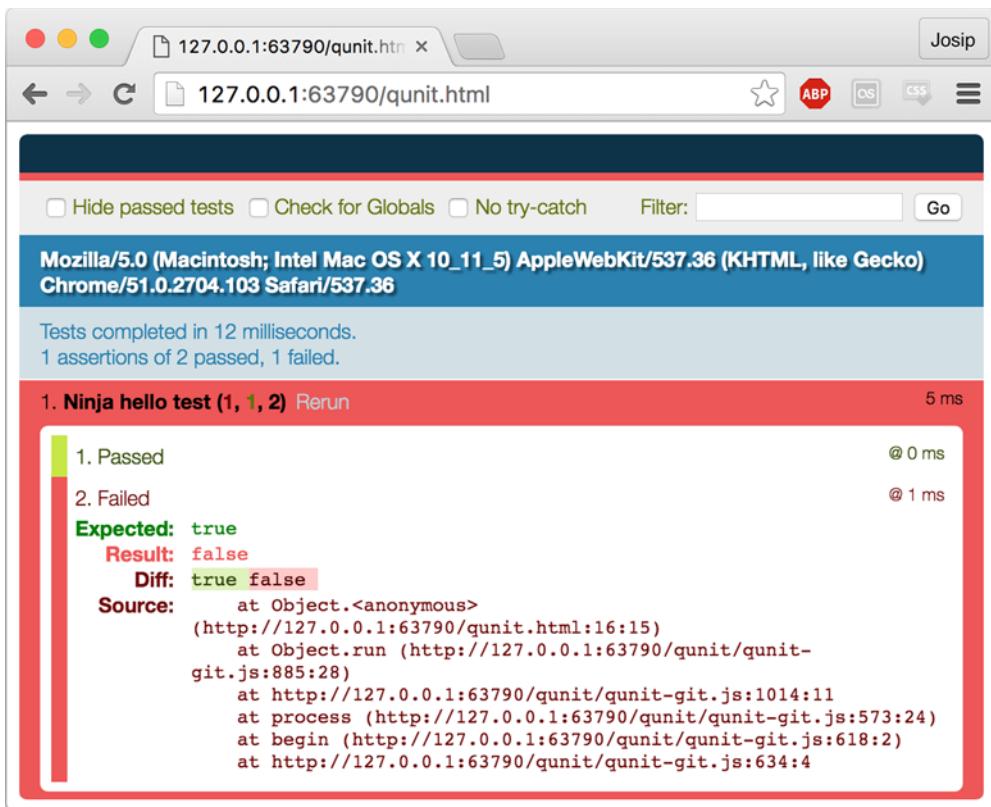
Figure B.13 An example of a QUnit test run. As a part of our test, we have one passing and one failing assertion (one assertion of two passed, one failed.) The displayed results put a much bigger emphasis on the failing test, to make sure we fix it as soon as possible.

The combination and naming of these functions are geared toward making the test suite almost conversational in nature. For example, the following listing shows how to test the sayHiToNinja function using Jasmine.

Listing B.7   Jasmine test example

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="lib/jasmine-2.2.0/jasmine.css">

  <script src="lib/jasmine-2.2.0/jasmine.js"></script>          Includes
  <script src="lib/jasmine-2.2.0/jasmine-html.js"></script>     Jasmine files
  <script src="lib/jasmine-2.2.0/boot.js"></script>
</head>
<body>
<script>
  function sayHiToNinja(ninja) {                 Declares the function
                                                 that we want to test
```

```
    return "Hi " + ninja;
  }

  describe("Say Hi Suite", function() {
    it("should say hi to a ninja", function() {
      expect(sayHiToNinja("Hatori")).toBe("Hi Hatori");
    });

    it("should fail", function(){
      expect(false).toBe(true);
    })
  });
</script>
</body>
</html>
```

Defines a test suite
that calls "Say Hi
Suite"

Specifies a single
test that checks
our function

Asserts
that our
function
produces
the right
result

Fails on purpose

The result of running this Jasmine test suite in the browser is shown in figure B.14.
More information on Jasmine can be found at http://jasmine.github.io/.
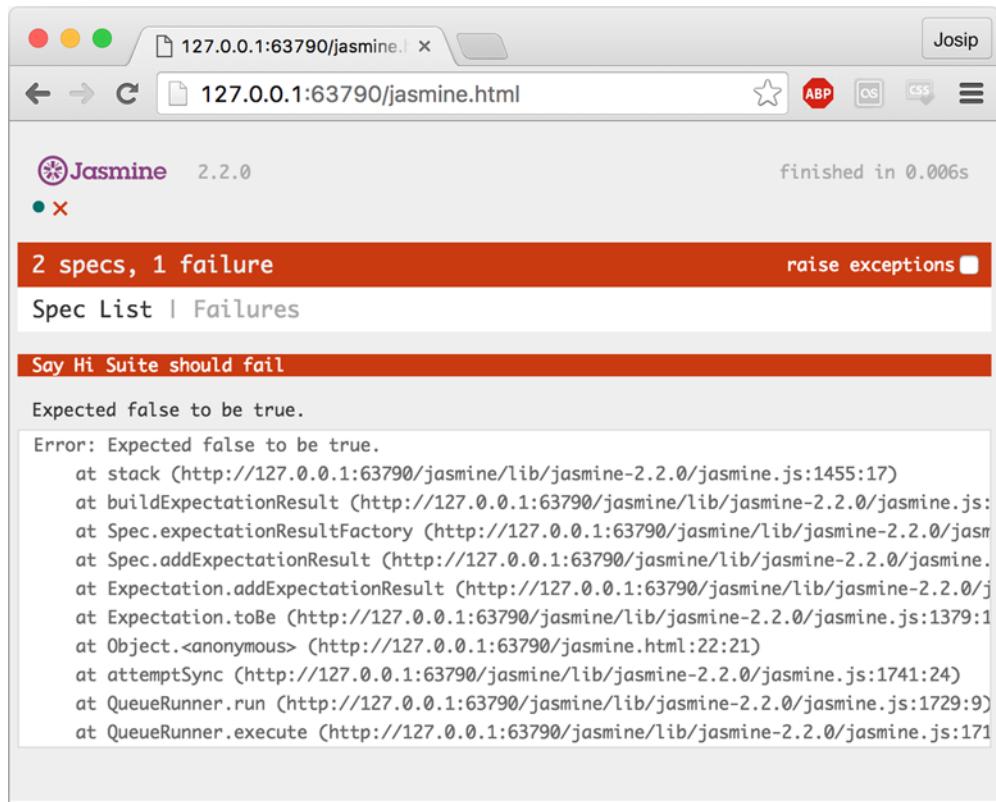
Figure B.14 **The result of running a Jasmine test suite in the browser. We have two tests: one passing and one failing (two specs, one failure).**

MEASURING CODE COVERAGE

It's difficult to say what makes a particular test suite *good*. Ideally, we should test all possible execution paths of our programs. Unfortunately, except for the most trivial cases, this isn't possible. A step in the right direction is trying to test as much code as we can, and a metric that tells us the degree to which a test suite covers our code is called *code coverage*.

For example, saying that a test suite has 80% code coverage means that 80% of our program code is executed by the test suite, whereas 20% of our code isn't. Although we can't be entirely sure that this 80% of code doesn't contain bugs (we might have missed an execution path that leads to one), we're completely in the dark about the 20% that wasn't even executed. This is why we should measure the code coverage of our test suites.

In JavaScript development, we can use several libraries to measure the coverage of our test suites, most notably Blanket.js (https://github.com/alex-seville/blanket) and Istanbul (https://github.com/gotwarlost/istanbul). Setting up these libraries goes beyond the scope of this book, but their respective web pages offer all the info we might need on properly setting them up.

appendix C
Exercise answers

Chapter 2. Building the page at runtime

1 What are the two phases in the lifecycle of a client-side web application?

A: The two phases in the lifecycle of a client-side web application are page building and event handling. In the page-building phase, the user interface of our page is built by processing HTML code and by executing mainline JavaScript code. After the last HTML node is processed, the page enters the event-handling phase, in which various events are processed.

2 What is the main advantage of using the `addEventListener` method to register an event handler versus assigning a handler to a specific element property?

A: When assigning event handlers to specific element properties, we can register only one event handler; `addEventListener`, on the other hand, enables us to register as many event handlers as necessary.

3 How many events can be processed at once?

A: JavaScript is based on a single-threaded execution model, in which events are processed one at a time.

4 In what order are events from the event queue processed?

A: Events are processed in the order in which they were generated: first in, first out.

Chapter 3. First-class functions for the novice: definitions and arguments

1 In the following code snippet, which functions are callback functions?

```
//sortAsc is a callback because the JavaScript engine
//calls it to compare array items
numbers.sort(function sortAsc(a,b){
```

411

```
  return a - b;
});

//Not a callback; ninja is called like a standard function
function ninja(){}
ninja();

var myButton = document.getElementById("myButton");
//handleClick is a callback, the function is called
//whenever myButton is clicked
myButton.addEventListener("click", function handleClick(){
  alert("Clicked");
});
```

2 In the following snippet, categorize functions according to their type (function declaration, function expression, or arrow function).

```
//function expression as argument to another function
numbers.sort(function sortAsc(a,b){
  return a - b;
});

//arrow function as argument to another function
numbers.sort((a,b) => b - a);

//function expression as the callee in a call expression
(function(){})();

//function declaration
function outer(){
  //function declaration
  function inner(){}
  return inner;
}

//function expression call wrapped in an expression
(function(){}());

//arrow function as a callee
(()=>"Yoshi")();
```

3 After executing the following code snippet, what are the values of variables samurai and ninja?

```
//"Tomoe", the value of the expression body of the arrow function
var samurai = (() => "Tomoe")();
//undefined, in case an arrow function's body is a block statement
//the value is the value of the return statement.
//Because there's no return statement, the value is undefined.
var ninja = (() => {"Yoshi"})();
```

4 Within the body of the test function, what are the values of parameters a, b, and c for the two function calls?

```
function test(a, b, ...c){ /*a, b, c*/}

// a = 1; b = 2; c = [3, 4, 5]
test(1, 2, 3, 4, 5);
// a = undefined; b = undefined; c = []
test();
```

5 After executing the following code snippet, what are the values of the `message1` and `message2` variables?

```
function getNinjaWieldingWeapon(ninja, weapon = "katana"){
  return ninja + " " + katana;
}

//"Yoshi katana" – there's only one argument in the call
//so weapon defaults to "katana"
var message1 = getNinjaWieldingWeapon("Yoshi");

//"Yoshi wakizashi" – we've sent in two arguments, the default
//value is not taken into account
var message2 = getNinjaWieldingWeapon("Yoshi", "wakizashi");
```

Chapter 4. Functions for the journeyman: understanding function invocation

1 The following function calculates the sum of the passed-in arguments using the `arguments` object. By using the rest parameters introduced in the previous chapter, rewrite the `sum` function so that it doesn't use the `arguments` object.

```
function sum(){
  var sum = 0;
  for(var i = 0; i < arguments.length; i++){
     sum += arguments[i];
  }
  return sum;
}

assert(sum(1, 2, 3) === 6, 'Sum of first three numbers is 6');
assert(sum(1, 2, 3, 4) === 10, 'Sum of first four numbers is 10');
```

A: Add a rest parameter to the function definition and slightly adjust the function body:

```
function sum(... numbers){
  var sum = 0;
  for(var i = 0; i < numbers.length; i++){
     sum += numbers[i];
  }
  return sum;
}

assert(sum(1, 2, 3) === 6, 'Sum of first three numbers is 6');
assert(sum(1, 2, 3, 4) === 10, 'Sum of first four numbers is 10');
```

2 After running the following code in the browser, what are the values of variables
   `ninja` and `samurai`?

```
function getSamurai(samurai){
  "use strict"

  arguments[0] = "Ishida";

  return samurai;
}

function getNinja(ninja){
  arguments[0] = "Fuma";
  return ninja;
}

var samurai = getSamurai("Toyotomi");
var ninja = getNinja("Yoshi");
```

A: `samurai` will have the value `Toyotomi`, and `ninja` will have the value `Fuma`.
Because the `getSamurai` function is in strict mode, the `arguments` parameter
doesn't alias function parameters, so changing the value of the first argument
won't change the value of the `samurai` parameter. Because the `getNinja` func-
tion is in nonstrict mode, any changes made to the `arguments` parameter will be
reflected in the function parameters.

3 When running the following code, which of the assertions will pass?

```
function whoAmI1(){
  "use strict";
  return this;
}

function whoAmI2(){
  return this;
}

assert(whoAmI1() === window, "Window?"); //fail
assert(whoAmI2() === window, "Window?"); //pass
```

A: The `whoAmI1` function is in strict mode; when it's called as a function, the
value of the `this` parameter will be `undefined` (and not `window`). The second
assertion will pass: If a function in nonstrict mode is called as a function, `this`
refers to the global object (the `window` object, when running the code in the
browser).

4 When running the following code, which of the assertions will pass?

```
var ninja1 = {
  whoAmI: function(){
    return this;
  }
```

```
};

var ninja2 = {
  whoAmI: ninja1.whoAmI
};

var identify = ninja2.whoAmI;

//pass: whoAmI called as a method of ninja1
assert(ninja1.whoAmI() === ninja1, "ninja1?");

//fail: whoAmI called as a method of ninja2
assert(ninja2.whoAmI() === ninja1, " ninja1 again?");

//fail: identify calls the function as a function
//because we are in non-strict mode, this refers to the window
assert(identify() === ninja1, "ninja1 again?");

//pass: Using call to supply the function context
//this refers to ninja2
assert(ninja1.whoAmI.call(ninja2) === ninja2, "ninja2 here?");
```

5 When running the following code, which of the assertions will pass?

```
function Ninja(){
  this.whoAmI = () => this;
}

var ninja1 = new Ninja();
var ninja2 = {
  whoAmI: ninja1.whoAmI
};

//pass: whoAmI is an arrow function inherits the function context
//from the context in which it was created.
//Because it was created during the construction of ninja1
//this will always point to ninja1
assert(ninja1.whoAmI() === ninja1, "ninja1 here?");

//false: this always refers to ninja1
assert(ninja2.whoAmI() === ninja2, "ninja2 here?");
```

6 Which of the following assertions will pass?

```
function Ninja(){
  this.whoAmI = function(){
    return this;
  }.bind(this);
}

var ninja1 = new Ninja();
var ninja2 = {
  whoAmI: ninja1.whoAmI
};
```

```
//pass: the function assigned to whoAmI is a function bound
//to ninja1 (the value of this when the constructor was invoked)
//this will always refer to ninja1
assert(ninja1.whoAmI() === ninja1, "ninja1 here?");
//fail: this in whoAmI always refers to ninja1
//because whoAmI is a bound function.
assert(ninja2.whoAmI() === ninja2, "ninja2 here?");
```

Chapter 5. Functions for the master: closures and scopes

1 Closures allow functions to

A: Access external variables that are in scope when the function is defined (option a)

2 Closures come with

A: Memory costs (closures keep alive the variables that are in scope when the function is defined) (option b)

3 In the following code example, mark the identifiers accessed through closures:

```
function Samurai(name) {
  var weapon = "katana";

  this.getWeapon = function(){
    //accesses the local variable: weapon
    return weapon;
  };

  this.getName = function(){
    //accesses the function parameter: name
    return name;
  }

  this.message = name + " wielding a " + weapon;

  this.getMessage = function(){
    //this.message is not accessed through a closure
   //it is an object property (and not a variable)
    return this.message;
  }
}

var samurai = new Samurai("Hattori");

samurai.getWeapon();
samurai.getName();
samurai.getMessage();
```

4 In the following code, how many execution contexts are created, and what's the largest size of the execution context stack?

```
function perfom(ninja) {
  sneak(ninja);
  infiltrate(ninja);
```

```
  }

  function sneak(ninja) {
    return ninja + " skulking";
  }

  function infiltrate(ninja) {
    return ninja + " infiltrating";
  }


  perfom("Kuma");
```

A: The largest stack size is 3, in the following situations:

- global code -> `perform` -> `sneak`
- global code -> `perform` -> `infiltrate`

5 Which keyword in JavaScript allows us to define variables that can't be reassigned to a completely new value?

A: `const` variables can't be reassigned to new values.

6 What's the difference between `var` and `let`?

A: The keyword `var` is used to define only function- or global-scoped variables, whereas `let` enables us to define block-scoped, function-scoped, and global-scoped variables.

7 Where and why will the following code throw an exception?

```
getNinja();
getSamurai();  //throws an exception

function getNinja() {
  return "Yoshi";
}

var getSamurai = () => "Hattori";
```

A: An exception will be thrown when trying to invoke the `getSamurai` function. The `getNinja` function is defined with a function declaration and will be created before any of the code is executed; we can call it "before" its declaration has been reached in code. The `getSamurai` function, on the other hand, is an arrow function that's created when the execution reaches it, so it will be undefined when we try to invoke it.

Chapter 6. Functions for the future: generators and promises

1 After running the following code, what are the values of variables `a1` to `a4`?

```
function *EvenGenerator(){
  let num = 2;
  while(true){
```

```
    yield num;
    num = num + 2;
  }
}

let generator = EvenGenerator();

//2 the first value yielded
let a1 = generator.next().value;

//4 the second value yielded
let a2 = generator.next().value;
//2, because we have started a new generator
let a3 = EvenGenerator().next().value;
//6, we go back to the first generator
let a4 = generator.next().value;
```

2 What's the content of the ninjas array after running the following code? (Hint: Think about how the for-of loop can be implemented with a while loop.)

```
function* NinjaGenerator(){
  yield "Yoshi";
  return "Hattori";
  yield "Hanzo";
}

var ninjas = [];
for(let ninja of NinjaGenerator()){
  ninjas.push(ninja);
}

ninjas;
```

A: The ninjas array will contain only Yoshi. This happens because the for-of loop iterates over a generator until the generator says it's done (without including the value passed along with done). This happens either when there's no more code in the generator to execute, or when a return statement is encountered.

3 What's the value of variables a1 and a2, after running the following code?

```
function *Gen(val){
  val = yield val * 2;
  yield val;
}

let generator = Gen(2);
//4. The value of the first value passed in through next: 3 is ignored
//because the generator hasn't yet started its execution, and there
//is no waiting yield expression.
//Because the generator is created with val being 2
//the first yield occurs for val * 2, i.e. 2*2 == 4
let a1 = generator.next(3).value;
```

```
//5: passing in 5 as a argument to next
//means that the waiting yielded expression will get the value 5
//(yield val * 2) == 5
//because that value is then assigned to val, the next yield expression
//yield val;
//will return 5
let a2 = generator.next(5).value;
```

4 What's the output of the following code?

```
const promise = new Promise((resolve, reject) => {
  reject("Hattori"); //the promise was explicitly rejected
});

//the error handler will be invoked
promise.then(val => alert("Success: " + val))
       .catch(e => alert("Error: " + e));
```

5 What's the output of the following code?

```
const promise = new Promise((resolve, reject) => {
  //the promise was explicitly resolved
  resolve("Hattori");
  //once a promise has settled, it can't be changed
  //rejecting it after 500ms will have no effect
  setTimeout(()=> reject("Yoshi"), 500);
});

//the success handler will be invoked
promise.then(val => alert("Success: " + val))
       .catch(e => alert("Error: " + e));
```

Chapter 7. Object orientation with prototypes

1 Which of the following properties points to an object that will be searched if the
   target object doesn't have the searched-for property?
   A: `prototype` (option c)

2 What's the value of variable a1 after the following code is executed?

```
function Ninja(){}
Ninja.prototype.talk = function (){
  return "Hello";
};

const ninja = new Ninja();
const a1 = ninja.talk(); //"Hello"
```

A: The value of variable a1 will be `Hello`. Even though the object `ninja` doesn't
possess the `talk` method, its prototype does.

3 What's the value of a1 after running the following code?

```
function Ninja(){}
Ninja.message = "Hello";

const ninja = new Ninja();

const a1 = ninja.message;
```

A: The value of variable `a1` will be undefined. The `message` property is defined in the constructor function `Ninja`, and isn't acessible through the `ninja` object.

4 Explain the difference between the `getFullName` method in these two code fragments:

```
//First fragment
function Person(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;

  this.getFullName = function () {
    return this.firstName + " " + this.lastName;
  }
}

//Second fragment
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Person.prototype.getFullName = function () {
  return this.firstName + " " + this.lastName;
}
```

A: In the first fragment, the `getFullName` method is defined directly on the instance created with the `Person` constructor. Each object created with the `Person` constructor gets its own `getFullName` method. In the second fragment, the `getFullName` method is defined on the prototype of the `Person` function. All instances created with the `Person` function will have access to this single method.

5 After running the following code, what will `ninja.constructor` point to?

```
function Person() { }
function Ninja() { }

const ninja = new Ninja();
```

A: When accessing `ninja.constructor`, the constructor property is found on `ninja`'s prototype. Because `ninja` was created with the `Ninja` constructor function, the constructor property points to the `Ninja` function.

6 After running the following code, what will `ninja.constructor` point to?

```
function Person() { }
function Ninja() { }
Ninja.prototype = new Person();
const ninja = new Ninja();
```

A: The `constructor` property is the property of the prototype object that was created with the constructor function. In this example, we override the built-in prototype of the `Ninja` function with a new `Person` object. Therefore, when a `ninja` object is created with the `Ninja` constructor, its prototype is set to the new `person` object. Finally, when we access the `constructor` property on the `ninja` object, because the `ninja` object doesn't have its own `constructor` property, its prototype, the new `person` object, is consulted. The `person` object also doesn't have a `constructor` property, so its prototype, the `Person.prototype` object, is consulted. That object has a `constructor` property, referencing the `Person` function. This example perfectly illustrates why we should be careful when using the `constructor` property: Even though our `ninja` object was created with the `Ninja` function, the `constructor` property, due to the hiccup of overriding the default `Ninja.prototype`, points to the `Person` function.

7 Explain how the `instanceof` operator works in the following example.

```
function Warrior() { }

function Samurai() { }
Samurai.prototype = new Warrior();

var samurai = new Samurai();

samurai instanceof Warrior; //Explain
```

A: The `instanceof` operator checks whether the prototype of the function on the right-hand side is in the prototype chain of the object on the left-hand side. The object on the left is created with the `Samurai` function, and its prototype has a new `warrior` object, whose prototype is the prototype of the `Warrior` function (`Warrior.prototype`). On the right we have the `Warrior` function. So in this example, the `instanceof` operator will return `true`, because the prototype of the function on the right, `Warrior.prototype`, can be found in the prototype chain of the object on the left.

8 Translate the following ES6 code into ES5 code.

```
class Warrior {
  constructor(weapon){
    this.weapon = weapon;
  }

  wield() {
```

```
    return "Wielding " + this.weapon;
  }

  static duel(warrior1, warrior2){
    return warrior1.wield() + " " + warrior2.wield();
  }
}
```

A: We can translate the code in the following way:

```
function Warrior(weapon) {
  this.weapon = weapon;
}

Warrior.prototype.wield = function () {
  return "Wielding " + this.weapon;
};

Warrior.duel = function (warrior1, warrior2) {
  return warrior1.wield() + " " + warrior2.wield();
};
```

Chapter 8. Controlling access to objects

1 After running the following code, which of the following expressions will throw an exception and why?

```
const ninja = {
  get name() {
    return "Akiyama";
  }
}
```

A: Calling `ninja.name()` throws an exception because `ninja` doesn't have a name method (option a). Accessing `ninja.name` in `const name = ninja.name` works like a charm; the getter gets activated, and the variable name gets the value `Akiyama`.

2 In the following code, which mechanism allows getters to access a private object variable?

```
function Samurai() {
  const _weapon = "katana";
  Object.defineProperty(this, "weapon", {
    get: () => _weapon
  });
}
const samurai = new Samurai();
assert(samurai.weapon === "katana", "A katana wielding samurai");
```

A: Closures allow getters to access private object variables. In this case, the `get` method creates a closure around the `_weapon` private variable defined in the constructor function, which keeps the `_weapon` variable alive.

3   Which of the following assertions will pass?

```
const daimyo = { name: "Matsu", clan: "Takasu"};
const proxy = new Proxy(daimyo, {
  get: (target, key) => {
    if(key === "clan"){
      return "Tokugawa";
    }
  }
});

assert(daimyo.clan === "Takasu", "Matsu of clan Takasu");      //pass
assert(proxy.clan === "Tokugawa", "Matsu of clan Tokugawa?"); //pass

proxy.clan = "Tokugawa";

assert(daimyo.clan === "Takasu", "Matsu of clan Takasu"); //fail
assert(proxy.clan === "Tokugawa", "Matsu of clan Tokugawa?"); //pass
```

A: The first assertion passes because `daimyo` has a `clan` property with value `Takasu`. The second assertion passes, because we access the property `clan` through a proxy with a get trap that always returns `Tokugawa` as the value of the `clan` property.

When the expression `proxy.clan = "Tokugawa"` is evaluated, the value `Tokugawa` is stored in the `daimyo`'s `clan` property because the proxy doesn't have a set trap, so the default action of setting the property is carried out on the target, `daimyo` object.

The third assertion fails, because the `daimyo`'s `clan` property has the value `Tokugawa` and not `Takasu`.

The fourth assertion passes, because the proxy always returns `Tokugawa`, regardless of the value stored in the target object's `clan` property.

4   Which of the following assertions will pass?

```
const daimyo = { name: "Matsu", clan: "Takasu", armySize: 10000};
const proxy = new Proxy(daimyo, {
  set: (target, key, value) => {
    if(key === "armySize") {
      const number = Number.parseInt(value);
      if(!Number.isNaN(number)){
        target[key] = number;
      }
    } else {
      target[key] = value;
    }
  },
});
```

```
//pass
assert(daimyo.armySize === 10000, "Matsu has 10 000 men at arms");
//pass
assert(proxy.armySize === 10000, "Matsu has 10 000 men at arms");

proxy.armySize = "large";
assert(daimyo.armySize === "large", "Matsu has a large army"); //fail

daimyo.armySize = "large";
assert(daimyo.armySize === "large", "Matsu has a large army");//pass
```

A: The first assertion passes; the value of `daimyo`'s `armySize` property is `10000`. The second assertion also passes; the proxy doesn't have a get trap defined so the value of the target, `daimyo`'s `armySize` property, is returned.

When the expression `proxy.armySize = "large";` is evaluated, the proxy's set trap is activated. The setter checks whether the passed-in value is a number, and only if it is, the value is assigned to the target's property. In this case, the passed-in value isn't a number, so no changes are made to the `armySize` property. For this reason, the third assertion, which assumes the change, fails.

The expression `daimyo.armySize = "large";` directly writes to the `armySize` property, bypassing the proxy. Therefore, the final assertion passes.

Chapter 9. Dealing with collections

1 After running the following code, what's the content of the `samurai` array?

```
const samurai = ["Oda", "Tomoe"];
samurai[3] = "Hattori";
```

A: The value of the `samurai` is `["Oda", "Tomoe", undefined, "Hattori"]`. The array starts with `Oda` and `Tomoe` at indexes `0` and `1`. We then add a new samurai, `Hattori`, at index 3, which "expands" the array, and leaves index 2 with `undefined`.

2 After running the following code, what's the content of the `ninjas` array?

```
const ninjas = [];

ninjas.push("Yoshi");
ninjas.unshift("Hattori");

ninjas.length = 3;

ninjas.pop();
```

A: The value of `ninjas` is `["Hattori", "Yoshi"]`. We start with an empty array, `push` adds `Yoshi` to the end, and `unshift` adds `Hattori` to the beginning.

Explicitly setting `length` to 3 expands the array with `undefined` at index 2. Calling `pop` removes that `undefined` from the array, leaving only `["Hattori", "Yoshi"]`.

3   After running the following code, what's the content of the `samurai` array?

```
const samurai = [];

samurai.push("Oda");
samurai.unshift("Tomoe");
samurai.splice(1, 0, "Hattori", "Takeda");
samurai.pop();
```

A: The value of `samurai` is `["Tomoe", "Hattori", "Takeda"]`. The array starts empty; `push` adds `Oda` to the end, and `unshift` adds `Tomoe` to the beginning; `splice` removes the item at index 1 (`Oda`) and adds `Hattori` and `Takeda` instead.

4   After running the following code, what's stored in the variables `first`, `second`, and `third`?

```
const ninjas = [{name:"Yoshi", age: 18},
    {name:"Hattori", age: 19},
    {name:"Yagyu", age: 20}];

const first = persons.map(ninja => ninja.age);
const second = first.filter(age => age % 2 == 0);
const third = first.reduce((aggregate, item) =>  aggregate + item, 0);
```

A: first: [18, 19, 20]; second: [18, 20]; third: 57

5   After running the following code, what's stored in the variables `first` and `second`?

```
const ninjas = [{ name: "Yoshi", age: 18 },
                { name: "Hattor", age: 19 },
                { name: "Yagyu", age: 20 }];

const first = ninjas.some(ninja => ninja.age % 2 == 0);
const second = ninjas.every(ninja => ninja.age % 2 == 0);
```

A: first: true; second: false

6   Which of the following assertions will pass?

```
const samuraiClanMap = new Map();

const samurai1 = { name: "Toyotomi"};
const samurai2 = { name: "Takeda"};
const samurai3 = { name: "Akiyama"};

const oda = { clan: "Oda"};
const tokugawa = { clan: "Tokugawa"};
const takeda ={clan: "Takeda"};
```

```
samuraiClanMap.set(samurai1, oda);
samuraiClanMap.set(samurai2, tokugawa);
samuraiClanMap.set(samurai2, takeda);

assert(samuraiClanMap.size === 3, "There are three mappings");
assert(samuraiClanMap.has(samurai1), "The first samurai has a
mapping");
assert(samuraiClanMap.has(samurai3), "The third samurai has a
mapping");
```

A: The first assertion fails, because a mapping for `samurai2` was created twice. The second assertion passes, because a mapping for `samurai1` was added. And the third assertion fails, because a mapping for `samurai3` was never created.

7 Which of the following assertions will pass?

```
const samurai = new Set("Toyotomi", "Takeda", "Akiyama", "Akiyama");
assert(samurai.size === 4, "There are four samurai in the set");

samurai.add("Akiyama");
assert(samurai.size === 5, "There are five samurai in the set");

assert(samurai.has("Toyotomi", "Toyotomi is in!");
assert(samurai.has("Hattori", "Hattori is in!");
```

A: The first assertion fails because `Akiyama` is added only once to the set. The second assertion also fails, because trying to add `Akiyama` once more will not change the set (nor its length). The last two assertions will pass.

Chapter 10. Wrangling regular expressions

1 In JavaScript, regular expressions can be created with which of the following?
A: Regular expression literals (option a) and by using the built-in `RegExp` constructor (option B). Answer c is incorrect; no built-in `RegularExpression` constructor exists.

2 Which of the following is a regular expression literal?
A: In JavaScript, a regular expression literal is enclosed in two forward slashes: `/test/` (option a).

3 Choose the correct regular expression flags.
A: With regular expression literals, expression flags are placed after the closing forward slash: `/test/g` (option a). With `RegExp` constructors, they're passed as a second argument: `new RegExp ("test", "gi");` (option c).

4 The regular expression `/def/` matches which of the following strings?
A: The regular expression `/def/` matches only `def`: *d* followed by *e*, followed by *f* (option b).

5 The regular expression `/[^abc]/` matches which of the following?

A: The regular expression /[^abc]/ matches one of the strings d, e, f—one character that isn't *a*, *b*, or *c* (option b).

6 Which of the following regular expressions matches the string hello?

A: Options a, b, and c match. /hello/ matches only the exact string hello. /hell?o/ matches either hello or helo (the second l is optional). /hel*o/, after the first l, matches any number of the letter l.

7 The regular expression /(cd)+(de)*/ matches which of the following strings?

A: Options a, c, d, and f are correct. /(cd)+(de)*/ matches one or more occurrences of cd followed by any number of occurrences of de.

8 In regular expressions, we can express alternatives with which of the following?

A: We use the pipe character, |, to express alternatives in regular expressions (option c).

9 In the regular expression /([0-9])2/, we can reference the first matched digit with which of the following?

A: \1 (option d)

10 The regular expression /([0-5])6\1/ will match which of the following?

A: In the regular expression /([0-5])6\1/, the first character is a digit from 0–5, the second character is the digit 6, and the third character is the first matched digit, so both 060 and 565 match (options a and d).

11 The regular expression /(?:ninja)-(trick)?-\1/ will match which of the following?

A: In the regular expression /(?:ninja)-(trick)?-\1/, the first group (?:ninja) is a noncapturing one, whereas the second group is a capturing one, which is optional (trick)?. But if this second group is found, in the end we have a backreference to it. Therefore, ninja- and ninja-trick-trick match (options a and c).

12 What is the result of executing "012675".replace(/0-5/g, "a")?

A: The code replaces all occurrences of digits from 0 to 5 with the letter a, so aaa67a results (option a).

Chapter 11. Code modularization techniques

1 Which mechanism enables private module variables in the module pattern?

A: In the module pattern, closures allow us to hide module internals: The methods of the module's public API keep the module internals alive (option b).

2 In the following code that uses ES6 modules, which identifiers can be accessed if the module is imported?

```
const spy = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
export const general = "Minamoto";
```

A: From outside the module, we can access only the `general` identifier, because it's the only identifier that has been explicitly exported (option c).

3 In the following code that uses ES6 modules, which identifiers can be accessed when the module is imported?

```
const ninja = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
const general  = "Minamoto";

export {ninja as spy};
```

A: From outside the module, we can access only the `spy` identifier: This is the only identifier that has been exported as an alias of the `ninja` variable (option a).

4 Which of the following imports are allowed?

```
//File: personnel.js
const ninja = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
const general  = "Minamoto";

export {ninja as spy};
```

A: The first import isn't allowed because the `personnel` module doesn't export `ninja` and `general` identifiers (option a). The second import is allowed because we import the whole module, which is accessible through the object `Personnel` (option b). The third import is also allowed, because we import the exported `spy` identifier (option c).

5 If we have the following module code, which statement will import the `Ninja` class?

```
//Ninja.js
export default class Ninja {
  skulk(){ return "skulking"; }
}
```

A: The first import is allowed: We import the default export (option a). The second import is allowed: We import the whole module (option b). The third import isn't allowed because it isn't syntactically correct (after the `*` should come the "as Name" part) (option c).

Chapter 12. Working the DOM

1 In the following code, which of the following assertions will pass?

```
<div id="samurai"></div>
<script>
  const element = document.querySelector("#samurai");
```

```
      assert(element.id === "samurai", "property id is samurai");
      assert(element.getAttribute("id") === "samurai",
             "attribute id is samurai");

      element.id = "newSamurai";

      assert(element.id === "newSamurai", "property id is newSamurai");
      assert(element.getAttribute("id") === "newSamurai",
             "attribute id is newSamurai");
    </script>
```

A: In this code, all assertions pass. The id attribute and the id property are linked; a change to one of them is reflected in the other.

2 Given the following code, how can we access the element's border-width style property?

```
<div id="element" style="border-width: 1px;
                         border-style:solid; border-color: red">
</div>
<script>
  const element = document.querySelector("#element");
</script>
```

A: The element.border-width expression doesn't make much sense. It calculates the difference between element.border and a variable width, which definitely isn't something we want. The next option, element.getAttribute ("border-width");, fetches the attribute of the HTML element, and not the style property. Finally, the last two options give the value of 1px (options c and d).

3 Which built-in method can get all styles applied to a certain element (styles provided by the browser, styles applied via style sheets, and properties set through the style attribute)?

A: Only the last option, getComputedStyle, is a built-in method that can be used to get the computed style of a certain HTML element (option c). The other three methods aren't included in the standard API.

4 When does layout trashing occur?

A: Layout trashing occurs when our code performs a series of consecutive reads and writes to DOM, each time forcing the browser to recalculate the layout information. This leads to slower, less responsive web applications.

Chapter 13. Surviving events

1 Why is it important that adding tasks into the task queue happens outside the event loop?

A: If the process of adding tasks into the task queue were part of the event loop, any events that occur while JavaScript code is being executed would be ignored. This would definitely be a bad idea.

2   Why is it important that each iteration of the event loop doesn't take much
    more than about 16 ms?

    A: To achieve smooth-running applications, the browser tries to perform ren-
    dering around 60 times per second. Because rendering is performed at the end
    of the event loop, each iteration shouldn't last much longer than 16 ms, unless
    we want to create slow and jagged applications.

3   What's the output from running the following code for 2 seconds?

```
setTimeout(function(){
  console.log("Timeout ");
}, 1000);

setInterval(function(){
  console.log("Interval ");
}, 500);
```

A: `Interval Timeout Interval Interval Interval` (option b). The set-
`Interval` method calls the handler with at least the fixed delay between each
call, until the interval is explicitly cleared. The `setTimeout` method, on the
other hand, calls the callback only once, after the specified delay has elapsed.
In this example, first the `setInterval` callback is fired once after 500 ms. Then
the `setTimeout` callback is invoked after 1000 ms, and another `setInterval`
immediately after. Our examination stops with two more `setInterval` callback
invocations, one at 1500 ms, and the other at 2000 ms.

4   What's the output from running the following code for 2 seconds?

```
const timeoutId = setTimeout(function(){
  console.log("Timeout ");
}, 1000);

setInterval(function(){
  console.log("Interval ");
}, 500);

clearTimeout(timeoutId);
```

A: `Interval Interval Interval Interval` (option c). The `setTimeout` call-
back is cleared before it has the chance to fire, so in this case we have only four
executions of the `setInterval` callback.

5   What's the output from running the following code and clicking the element
    with the ID `inner`?

```
<body>
  <div id="outer">
    <div id="inner"></div>
  </div>
  <script>
    const innerElement = document.querySelector("#inner");
    const outerElement = document.querySelector("#outer");
```

```
      const bodyElement = document.querySelector("body");
      innerElement.addEventListener("click", function(){
        console.log("Inner");
      });

      outerElement.addEventListener("click", function(){
        console.log("Outer");
      }, true);

      bodyElement.addEventListener("click", function(){
        console.log("Body");
      });
  <script>
</body>
```

A: `Outer Inner Body` (option c). The click handlers on the `innerElement` and the `bodyElement` are registered in bubbling mode, whereas the click handler on the `outerElement` is registered in capturing mode. When processing the event, the event first trickles down from the top and calls all event handlers in capturing mode. The first message will be `Outer`. After the event target is reached, in our case the element with ID `inner`, the event bubbling takes place, and the event bubbles up. Therefore, the second message will be `Inner`, and the third one will be `Body`.

Chapter 14. Developing cross-browser strategies

1 What should we take into account when deciding which browsers to support?

   A: When deciding which browsers to support, we should at least take into account the following:

   - The expectations and needs of the target audience
   - The market share of the browser
   - The amount of effort necessary to support the browser

2 Explain the problem of greedy IDs.

   A: When working with form elements, the browser adds properties to the form element for each descendent element with an ID, so that we can access these elements easily through the form element. Unfortunately, this can override some of the built-in form properties such as `action` or `submit`.

3 What is feature detection?

   A: Feature detection works by determining whether a certain object or object property exists, and if so, assumes that it provides the implied functionality. Instead of testing whether the user is using a particular browser and then implementing work-arounds based on that information, we test whether a certain feature works as it's supposed to.

4 What is a browser polyfill?

   A: If we want to use a certain functionality that's not supported by all targeted browsers, we can use feature detection. If a current browser doesn't support a certain functionality, we provide our own implementation, and this is called polyfilling.

index

MORE TITLES FROM MANNING

Functional Programming in JavaScript
by Luis Atencio

ISBN: 9781617292828
272 pages
$44.99
June 2016

### JavaScript Application Design
A Build First approach

by Nicolas G. Bevacqua

ISBN: 9781617291951
344 pages
$39.99
January 2015

Building the Web of Things
With examples in Node.js and Raspberry Pi
by Dominique D. Guinard and Vlad M. Trifa

ISBN: 9781617292682
344 pages
$34.99
June 2016

For ordering information go to www.manning.com

Classes act as syntactic sugar around JavaScript's prototypes:

```
class Person {
  constructor(name){ this.name = name; }
  dance(){ return true; }
}
class Ninja extends Person {
  constructor(name, level){
    super(name);
    this.level = level;
  }
  static compare(ninja1, ninja2){
    return ninja1.level - ninja2.level;
  }
}
```

Proxies control access to other objects. Custom actions can be executed when an object is interacted with (for example, when a property is read or a function is called):

```
const p = new Proxy(target, {
  get: (target, key) => { /*Called when property accessed through proxy*/ },
  set: (target, key, value) => { /*Called when property set through proxy*/ }
});
```

Maps are mappings between a key and a value:

- new Map() creates a new map.
- Use the set method to add a new mapping, the get method to fetch a mapping, the has method to check whether a mapping exists, and the delete method to remove a mapping.

Sets are collections of unique items:

- new Set() creates a new set.
- Use the add method to add a new item, the delete method to remove an item, and the size property to check the number of items in a set.

for...of loops iterate over collections and generators.

Destructuring extracts data from objects and arrays:

- **const {name: ninjaName}** = ninja;
- **const [firstNinja]** = ["Yoshi"];

Modules are larger units of organizing code that allow us to divide programs into clusters:

```
export class Ninja{}; //Export an item
export default class Ninja{} //Default export
export {ninja};//Export existing variables
export {ninja as samurai}; //Rename an export

import Ninja from "Ninja.js"; //Import a default export
import {ninja} from "Ninja.js"; //Import named exports
import * as Ninja from "Ninja.js"; //Import all named exports
import {ninja as iNinja} from "Ninja.js"; //Import with a new name
```

Secrets of the JavaScript Ninja Second Edition

Resig • Bibeault • Maras

Free eBook
SEE INSERT

Javascript is rapidly becoming a universal language for every type of application, whether on the web, on the desktop, in the cloud, or on mobile devices. When you become a JavaScript pro, you have a powerful skill set that's usable across all these domains.

Secrets of the JavaScript Ninja, Second Edition uses practical examples to clearly illustrate each core concept and technique. This completely revised edition shows you how to master key JavaScript concepts such as functions, closures, objects, proto-types, and promises. It covers APIs such as the DOM, events, and timers. You'll discover best practice techniques such as testing, and cross-browser development, all taught from the perspective of skilled JavaScript practitioners.

What's Inside

- Writing more effective code with functions, objects, and closures
- Learning to avoid JavaScript application pitfalls
- Using regular expressions to write succinct text-processing code
- Managing asynchronous code with promises
- Fully revised to cover concepts from ES6 and ES7

You don't have to be a ninja to read this book—just be willing to become one. Are you ready?

John Resig is an acknowledged JavaScript authority and creator of the jQuery library. **Bear Bibeault** is a web developer and au-thor of the first edition, as well as coauthor of *Ajax in Practice*, *Prototype and Scriptaculous in Action*, and *jQuery in Action*. **Josip Maras** is a post-doctoral researcher and teacher.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/secrets-of-the-javascript-ninja-second-edition

" Essential reading for developers of any discipline ... with powerful techniques to improve your JavaScript. **"**
—Becky Huett, Big Shovel Labs

" Excellent and comprehensive insight into the magic of functions and closures for the efficient use of JavaScript. **"**
—Gerd Klevesaat, Siemens

" The essential resource for moving your JavaScript skills to the next level. **"**
—David Starkey, Blum

" Helps you master both the stealthy and bold techniques of modern JavaScript. **"**
—Christopher Haupt
New Relic Inc.

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]