Secrets of the JavaScript Ninja

SECOND EDITION

John Resig
Bear Bibeault
Josip Maras

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Secrets of the JavaScript Ninja
Second Edition
Version 5

For more information on this and other Manning titles go to
www.manning.com

welcome

Welcome to the MEAP for the second edition of *Secrets of the JavaScript Ninja*. Thank you for supporting the development of this book!

Since the release of the first edition, a lot has happened in the web development world. Applications based on the ideas originating on the web have become ubiquitous. Nowadays, in addition to run-of-the-mill web applications, we can develop cross-platform standard desktop applications, hybrid mobile applications, server-side applications, and even applications for embedded devices by using ideas and technologies that have originated on the web. This whole thriving ecosystem is based on one of the most popular programming languages today – JavaScript. In order to keep up with these changes and to make us developers more productive, the ECMAScript committee, in charge of steering the development of JavaScript as a language, has been working hard in adding new language features. Since the conception of the first edition of this book, two new versions of the JavaScript standard: ECMAScript 5 and ECMAScript 6 have been released (with a third one, ECMAScript 7 already in the works). These new editions have brought significant changes to the JavaScript world, which we'll explore in this second edition of the book.

At this time, we are releasing the first three chapters. The first chapter gives an overview of the book, and introduces important concepts that we'll thoroughly explore throughout the rest of the book. In chapter 2, we'll make an in-depth look at how a page is built and interacted with at run-time, and in chapter 3 we'll teach you why testing is so important and give you a brief survey of some of the testing tools available. Then we'll develop some surprisingly simple testing tools that you'll use throughout the rest of your training.

Looking forward, we'll continue with, some would argue, the most important concept in JavaScript – functions. We'll pay special attention to significant changes that were introduced in new versions of JavaScript, for example: new ways of defining functions and changes to the scoping rules.

I hope you'll visit the Author Online forum and that we'll have a fruitful discussion on any comments and questions that you might have as you read through the second edition. These will surely help us improve the book.

—Josip Maras

brief contents

1

Enter the Ninja

This chapter covers

- An overview of JavaScript and how it has evolved
- An overview of the client-side Browser API
- How client-side web development techniques transfer easily to JavaScript development on other platforms.
- An overview of three JavaScript development best practices : debugging, testing and performance analysis.

Applications based on the ideas originating from the web are becoming ubiquitous; the technologies we once exclusively used to develop client-side web applications, executed in the browser, have spread to other domains, as shown in figure 1.1 below.

Now, we can use similar or even the same ideas and techniques to develop standard desktop applications (for example, code editors like Atom.io), hybrid mobile applications (for example Netflix), server-side applications (by using Node.js), and even applications for embedded devices (with Espruino or Tessel). In the history of computing, it has been very rare that a particular skill set is so easily transferable and useful across so many different domains.

In order to take full advantage of this emerging application landscape, we need a deep understanding of the fundamental principles standing behind *client-side web development* because the ideas, tools, and techniques that have originated in client-side web development have been adapted to work in these other application domains also.

In this book, we are going to focus on these core principles:

- A deep understanding of the core mechanics of JavaScript
- Understanding the browser: the engine in which our JavaScript applications are

executed

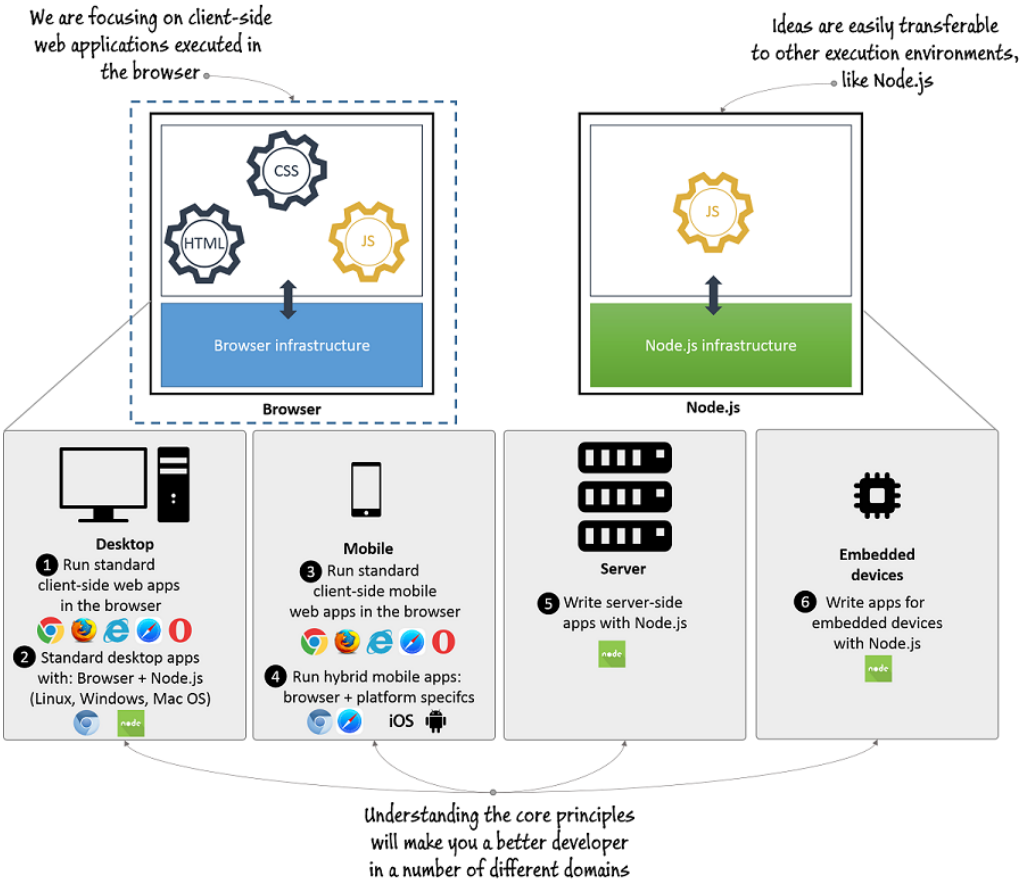


Figure 1.1 The JavaScript eco-system: JavaScript applications can nowadays be executed on different devices (desktop computers, mobile devices, servers, and even on embedded devices). Our focus is on developing client-side web apps executed in the browser.

In this chapter, we'll start by exploring the core principles of JavaScript, and we'll take a dep look at the browsers, as the engines in which our JavaScript code will be executed. We'll put a special focus on the cross-browser issues that are important because users can access our web applications with different browsers; browsers whose implementation details often slightly differ. We'll continue by exploring some of the best practices of JavaScript development, and finally we'll conclude the chapter by emphasizing how the skills that you'll learn throughout the book – JavaScript development for the browser – can easily be transferred to other

application domains. These skills can help you become a better craftsman, and they can increase your productivity in a wide range of application domains.

Let's start with the core principles of JavaScript.

1.1 Understanding the JavaScript language

Many JavaScript coders, as they advance through their careers, get to the point where they're actively using the vast number of elements comprising the language. In many cases, however, those skills may not be taken beyond fundamental levels. Our guess is that this might often be because JavaScript, using a C-like syntax, on the surface level looks pretty similar to other wide-spread C-like languages (such as C# or Java), and thus leaves the impression of familiarity. People often feel that knowing C# or Java, they already have a pretty solid understanding of how JavaScript works. However, it's a trap! When compared to other main-stream languages, like Java, C#, or C++, JavaScript is much more functionally oriented, which means there are some fundamentally different concepts with respects to most other languages, such as:

- *Functions as first-class objects*
- *Function closures*

  Function closures are a concept that 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 keeps alive ("closes over") the external variables used in its body*. Don't worry if for now, you don't see the many benefits of closures – we'll make sure that all is crystal clear in chapters 5 and 6. In addition to closures, we'll thoroughly explore the many different aspects of functions themselves in chapters 4 and 6, as well as identifier scopes in chapter 5.

  Scopes are particularly interesting because, up to recently, the scoping rules in JavaScript were more limited than the scoping rules in other C-like languages – up to the latest version of JavaScript there were no 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 main-stream languages (such as C#, Java, or Ruby) which use class-based object orientation, JavaScript is the only widely-used language that uses prototypes. In prototype-based object orientation, *an object is merely a collection of properties with some values*, and every object can have a prototype to which a search for a particular property can be delegated to, when the object itself does not have that property. Prototype-based object orientation in particular is the source of woes for many developers. Often, when developers come to JavaScript from class-based languages, for example Java, they try to use JavaScript as it were Java, essentially writing Java's class-based code using the syntax of JavaScript. Then, for some reason, developers often get 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 is implemented in JavaScript.

JavaScript actually consists of a close relationship between objects and prototypes, and functions and closures, which is shown in Figure 1.2 below.

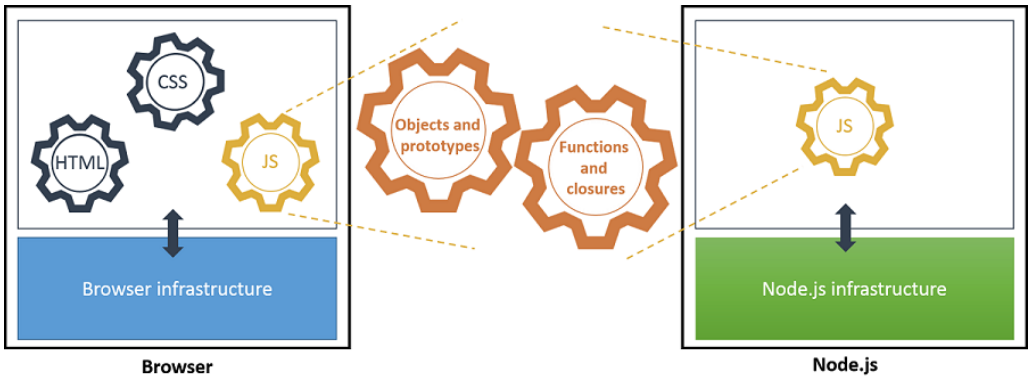


Figure 1.2 JavaScript consists of a close relationship between objects and prototypes, and functions and closures; concepts applicable across multiple domains.

Understanding the strong relationship between these concepts can vastly improve our JavaScript programming ability, giving us a strong foundation for any type of application development, regardless of whether our JavaScript code will be executed within a web page, a desktop app, a mobile app, or on the server.

In addition to these fundamental concepts, there are also other JavaScript features that can help us write better and more efficient code. In particular, we are going to focus on

- *regular expressions,* which allow us to simplify what would otherwise be quite complicated pieces of code
- *promises*, that give us better control over asynchronous code.
- *Anything else? Traversing arrays?*

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

Some of these language features have been a part of JavaScript for a while now, but some are a relatively recent addition. In the next section, we'll go through how JavaScript has become the language that it currently is.

1.1.1 History and the current state of JavaScript

JavaScript was created by Brendan Eich in 10 (yes, you read that right – TEN) days in the May of 1995, by combining features from several quite different languages:

- *Java*, which had a strong influence on the look and the feel of the basic syntax;
- *Scheme*, which inspired the use of functions as first-class objects;
- *Self*, through the use of prototypal inheritance
- *Perl*, for its regular expressions.

All these influences have led to the creation of JavaScript as the language that has nailed the sweet spot between object-orientation and functional programming.

JavaScript is standardized through the ECMAScript specification (often abbreviated simply as ES), beginning with ES1 in 1997 and ES2 in 1998. The third version, ES3, standardized in 1999, was the first widely "popular" version that appeared just as the web has started to mature as a development platform. The first edition of this book mostly deals with the ES3 version of JavaScript.

Then, unfortunately, in the next decade, nothing much really happened. The next version of JavaScript, ES4, was under development, but due to various reasons, mostly feature creep and politics, didn't make it.

Fortunately, in 2009, the ECMAScript comity decided to reduce the scale a bit and the ES5 specification was created. This version was much smaller in scope, but it has still managed to introduce some really useful additions, such as:

- the *strict* mode, which changes some JavaScript semantics, so that errors are thrown instead silently picked up. This increases the confidence we have in our code. We'll spend some time discussing the strict mode in chapters 4 and 5.
- Object extensions that allow us to create and configure objects and their properties (e.g `Object.create`, `Object.defineProperty`, getters and setters), which gives us much more control when working with objects. We'll study these extensions in chapters 8 and 9.
- Native JSON handling; over the years JSON (JavaScript Object Notation) has emerged as one of the most popular data-interchange formats. Previously, we as developers had to rely on different libraries for JSON support, but since ES5, this burden has been lifted from our shoulders, with a native implementation.
- A number of useful functions that help us work with dates (`Date.now`), arrays (`Array.isArray`, `forEach`, `every`, `filter`, etc., that we'll study in chapter 10), and functions (`Function.bind` that we'll explore in chapter 6).

After ES5, the ECMAScript comity started their work on the next, ES6 version, which was released in 2015. ES6 is the current, latest version of JavaScript, and it has introduced a lot of new additions that will have a huge influence on the way we write our JavaScript code. New features include:

- *Classes* – even though JavaScript is built on top of prototype inheritance, a lot of

developers are used to class-based inheritance, and through the years a number of libraries that try to mimic classes have been developed. The ECMAScript comity has finally embraced this fact, and has given us built-in syntactic sugar (nota that we still don't have true classes, as in Java or in C#), that allows us to mimic class-based inheritance in JavaScript. We'll spend a good deal of chapter 8 on working with classes.

- Generators – a completely new type of functions that, unlike standard functions, can be paused and resumed. We'll put a special focus on generator functions in chapter 7.
- Promises – objects that represent values that will be known at a later point in time, which are incredibly helpful when dealing with asynchronous code. A good deal of chapter 7 will be focused on promises. In addition, we'll show you how to combine promises and generators to significantly increase the ease of dealing with asynchronous code.
- Proxies – objects through which we can control other objects. In chapter 9, we'll show you how to use them for logging, performance measurement, and detecting object changes (which is really useful for developing web UI's).
- Block level variables – one of the tripping points with earlier versions of JavaScript is that it only provided support for function level and global variables; there was no block scope as in C, Java, or C#. This is finally fixed with the introduction of two new keywords: `const` and `let`, through which you can define block level variables. We'll explore them in depth in chapter 5.
- Various additions that make dealing with functions a lot easier. For example, arrow functions allow us to define functions with a lot less syntactic clutter, while default function parameters and rest parameters facilitate our dealing with function parameters. We'll explore these additions in chapter 4.
- A number of small improvements that we'll visit throughout the book, such as the new `for…of` loop which enables us to more easily loop through collections; template strings, which significantly improve string handling; spread operator, maps, and sets for working with collections of data, etc.
- From this list you can see that ES6 has introduced a large number of new features to JavaScript, features that we'll thoroughly explore in this second edition of the "Secrets of the JavaScript Ninja" book.

ECMAScript Language Specification

The current ES6 specification of JavaScript can be found at: http://www.ecma-international.org/ecma-262/6.0/. While, we suggest that you take a look at it, it's not really the type of reading that too many people enjoy (nor is it meant to be as such). On the other hand, Mozilla keeps a really good, easily readable reference, with lots of examples at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference.

Currently, the ECMAScript comity is busy working on the next, ES7 version of JavaScript, which should come out by the end of 2016. The ES7 version (at least when compared to ES6) will be a relatively smaller upgrade to JavaScript, because the idea is to focus on smaller, yearly increments to the language. (For this reason, ES6 is sometimes referred to as ES2015 and ES7 as ES2016.) Since the feature list hasn't yet been finalized, in this book we'll explore only the most important ES7 features, such as the new type of function, the async function, which helps us deal with asynchronous code (in chapter 7), and a new way of tracking object changes with `Object.observe`, in chapter 9.

The following figure presents a short summary of different versions of JavaScript.

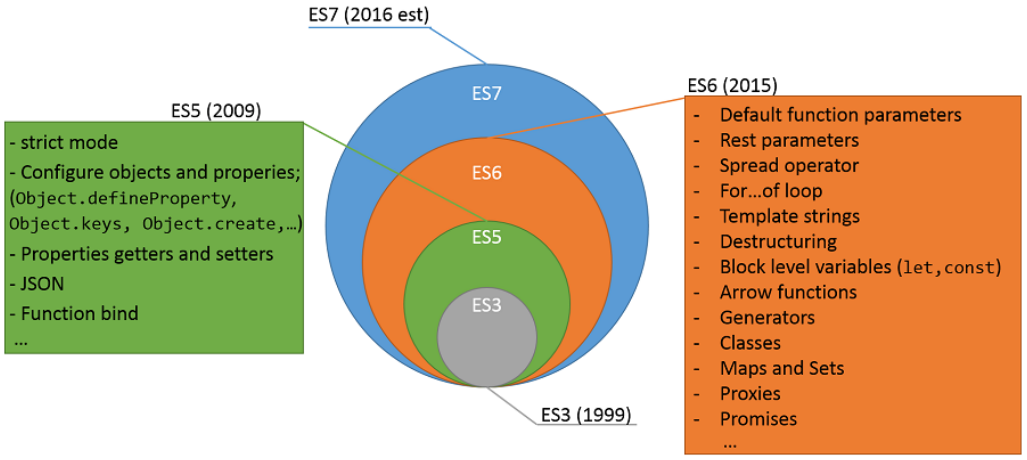


Figure 1.3 ECMAScript (ES) specification standardizes JavaScript. The current version is ES6.

As you can see from the figure 1.3, each new version simply builds on top of the last version. This means that our current ES5 code is completely valid ES6 code!

Unfortunately, it's not all sunshine and rainbows.

As you probably know, our JavaScript code has to be executed by a JavaScript engine (which is in most cases built into our browsers, but it can also be a part of our server-side web applications as with node.js). This means that the JavaScript engine running our code has to be able to understand our well-crafted, cutting-edge ES6 code. Unfortunately, while the browsers are trying to keep up, and are doing better, week after week, there is a chance that you'll run into some unsupported features. Luckily, there's a list of browsers and the current state of their support available at: http://kangax.github.io/compat-table/es6/.

USING ES6 TODAY WITH TRANSPILERS

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

benefits of newest JavaScript features, but if we are, at the same time, taken hostage by reality in which the users of our web applications are still using some older browsers?

The answer to this problem is in using *transpilers* (coming from "transformation + compiling"), tools that take our bleeding-edge ES6 code and transform it into equivalent (or if that's not possible, really, really similar) ES5 code that works in most current browsers. Currently, the most popular transpilers are Traceur (https://github.com/google/traceur-compiler) and Babel (https://babeljs.io/). Setting them up is really easy, just follow one of the tutorials, for example: 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. For this reason, in order to effectively use the browser a platform, we have to get our hands dirty, and study the inner-workings of browsers.

1.2 Understanding the browser

These days, JavaScript applications can be executed in different environments, but the environment from which it all began, the environment from which all other environments have taken a lot of ideas, and the environment on which we'll especially focus on, is the *browser*.

The browser provides a number of concepts and APIs that we'll thoroughly explore; see figure 1.4.

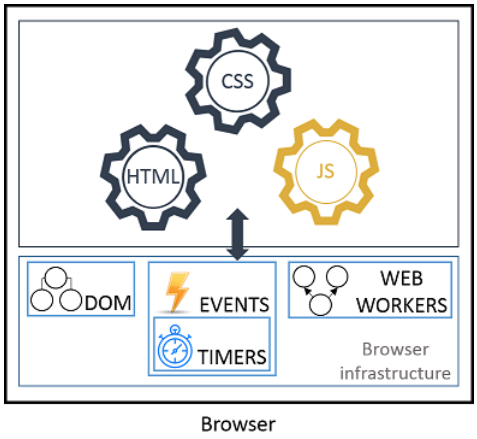


Figure 1.4 Client-side web applications rely on the infrastructure provided by the browser, out of which we will particularly focus on the DOM, events, timers, and browser API's.

As figure 1.4 shows we'll particularly focus on:

- *The Document Object Model*, or DOM, a structured representation of the UI of our client-side web applications which is, at least initially, built from the HTML code of a web application. In order to develop great applications, in addition to having a deep

understanding of the core JavaScript mechanics, we'll also go deep into how the DOM is constructed in chapter 2 and how to write effective code that manipulates the DOM in chapter 13. This will put the creation of advanced, highly dynamic UIs at our fingertips.

- *Events*. A huge majority of JavaScript applications are *event-driven* applications, meaning that the majority of code is executed in the context of a response to some event (for example: network events, timers, or user generated events such as clicks, mouse moves, keyboard presses, and so on). For this reason, we'll thoroughly explore the mechanisms behind events. We'll pay special attention to *timers*, which are all too frequently a mystery, but which give us the ability to tackle complex coding tasks such as long-running computations and smooth animations.
- *Browser API*. In order to help us interact with the world (for example to communicate with remote servers), the browser offers an API that we'll explore throughout the book.

Perfecting our JavaScript programming skills and achieving deep understanding of APIs offered by the browser will take us far. But sooner, rather than later, we're going to run face first into *The Browsers* and their various issues and inconsistencies.

CROSS-BROWSER CONSIDERATIONS

In a perfect world, all browsers would be bug-free and would support web standards in a consistent fashion, but 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'll need to deal with. Developing a comprehensive strategy for tackling these browser issues, and becoming intimately familiar with their differences and quirks, is almost as important as proficiency in JavaScript itself.

When writing browser applications or JavaScript libraries to be used in them, picking and choosing which browsers to support is an important consideration. We'd probably like to support them all, but limitations on development and testing resources dictate otherwise. For this reason, we'll thoroughly explore different strategies to cross-browser development in chapter 16.

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

1.3 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, which we'll examine in depth in chapter 3, are known as *best practices* and, in addition to mastery of the language, include such elements as

- Debugging skills
- Testing

- Performance analysis

It's vitally important to adhere to these practices in our coding. Let's examine a couple of these practices.

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 due to 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 Dev Tools – developed by the Chrome team, used in Chrome and Opera
- Firefox DevTools – a tool included into Firefox, built by the Firefox team.
- F12 developer tools – Included in Internet Explorer and Microsoft Edge.
- WebKit Inspector – used by Safari.

As we 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!

As all of them are based on similar ideas, which were mostly introduced by Firebug, they offer similar functionality, including: exploring the DOM, debugging JavaScript, editing CSS styles, tracking network events, and so on. Any of them will do a perfectly fine job – simply use the one offered by your browser of choice, or in the browser in which you are investigating bugs. (In addition, it's nice to know that you can use some of them (for example Chrome Dev Tools) to debug other kinds of applications, such as node.js apps.) The following figure shows the developer tools used from Chrome, Firefox, IE, and Safari.

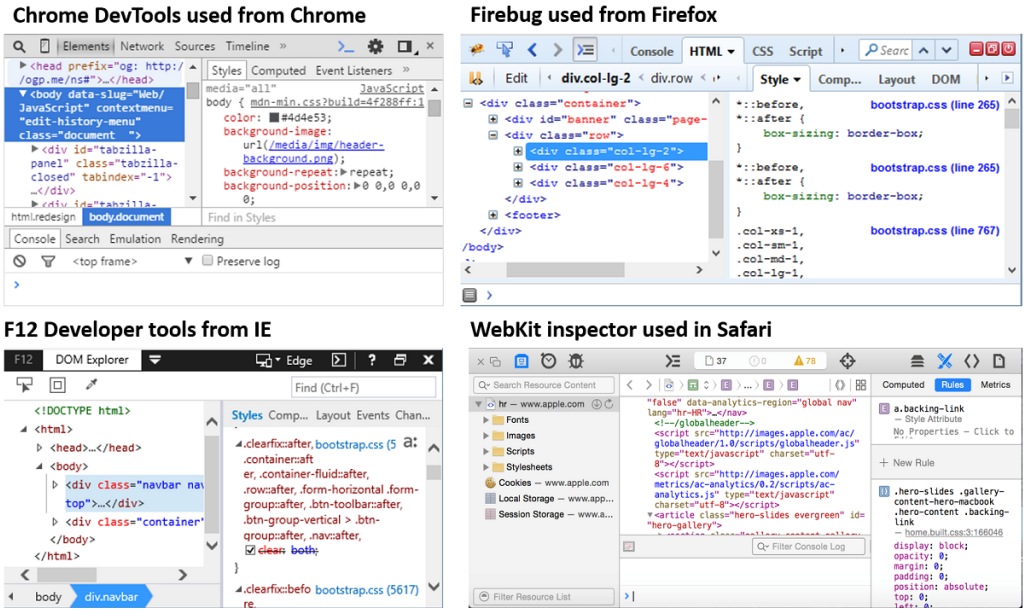


Figure 1.5 Every major browser offers a set of developer tools that will help you debug your web applications. Simply use the one offered by your browser of choice.

We'll introduce you to some of the debugging techniques in Chapter 3.

1.3.2 Testing

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

The general form of this function is:

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

where 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 that the `assert` function isn't a standard feature of the language, so we'll be implementing it ourselves. We'll be discussing its implementation and use in chapter 3.

1.3.3 Performance analysis

Another important practice is performance analysis. The JavaScript engines in the browsers have made astounding strides in the performance of JavaScript itself, but that's not an excuse for us to write sloppy and inefficient code.

We'll be using code such as the following later in this book for collecting performance information:

Listing 1.1 A simple performance counter

```
console.time("My operation");              #A
for(var n = 0; n < maxCount; n++){         #B
  /*perform the operation to be measured*/      #B
}                                             #B
console.timeEnd("My operation"); #C
```

#A Start the timer
#B Perform the operation multiple times
#C Stop 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 our operation begins executing, the call to `console.time` starts a timer with a certain name (in this case: "My operation")
- Then we run the code in the `for` loop some number of times (in this case `maxCount` number of 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 upon 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 will cause the browser to output the time that elapsed since the timer was started.

This way of determining the performance of our code is perfectly fine to test a certain idea, if we want to do it easily and quickly. However, for a more rigorous approach, we suggest using jsPerf (http://jsperf.com/), an online tool that allows you to create and share performance test cases.

A classic case for performance checks: for-vs-foreach

A simple performance test case you can do yourself is the for-vs-foreach (http://jsperf.com/for-vs-foreach) comparison between different ways of looping through arrays to determine the sum of the array's items. By studying figure 1.5 below, you can see that from the performance perspective, it's

much better to use a simple for loop than a `forEach` method (the `forEach` loop is in this case around 90% slower). Don't worry if you haven't used the `forEach` method on arrays yet, we'll go deep into arrays in chapter 10.

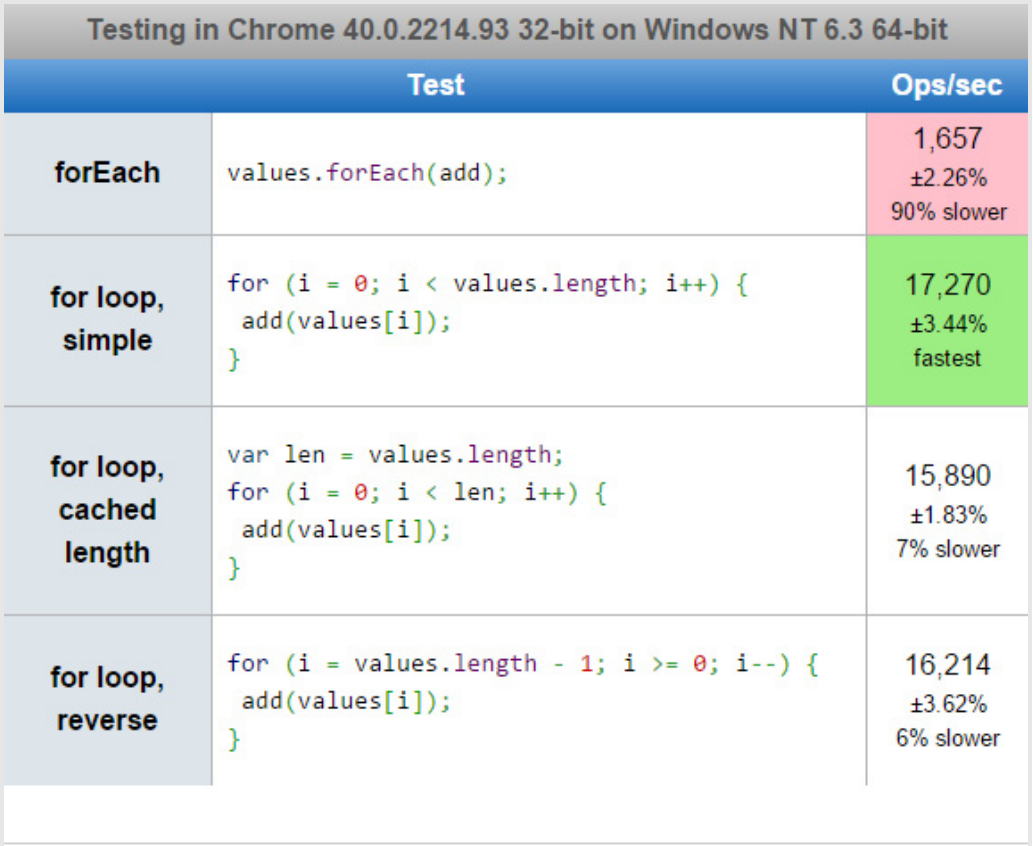


Figure 1.6 Use http://jsperf.com/ to compare the performance of different implementations

Measurements such as these are vital if we want to make informed decision when faced with a number of possible alternatives.

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

1.4 Skill transferability

In this book, we are going to put a special focus on developing effective cross-browser JavaScript applications. But, as we have already mentioned, the ideas, tools, and techniques that have originated in client-side web development have also permeated other application domains also. Let's take a look at figure 1.5 that shows how combining a deep understanding of fundamental JavaScript principles with the knowledge of core APIs can make you a more versatile developer.

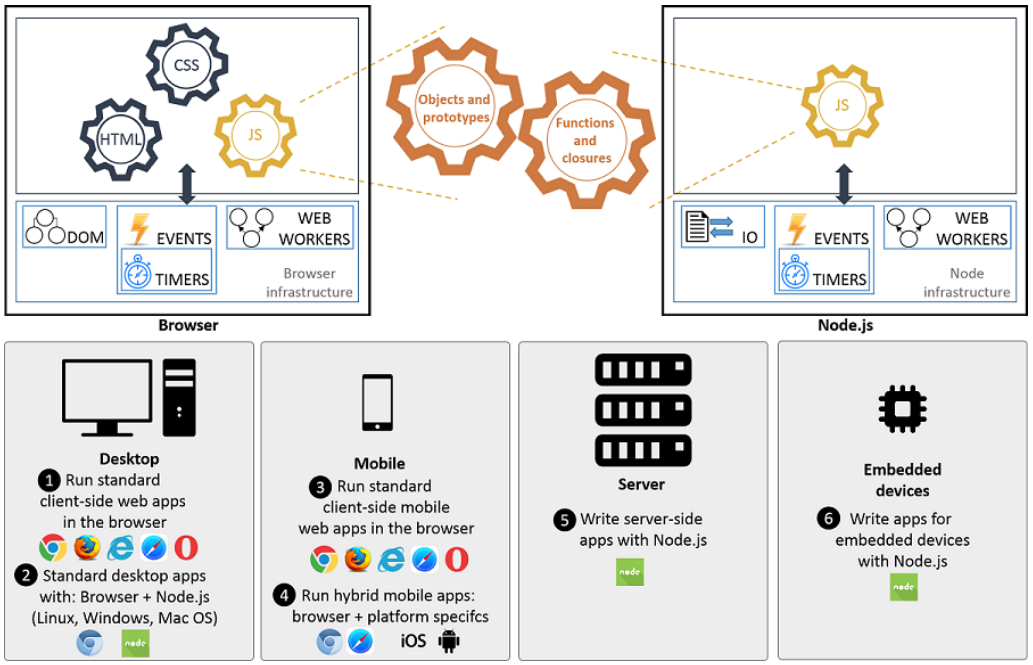


Figure 1.7 Understanding the core principles behind client-side web applications will make you a better developer in different domains.

By using the browsers and Node.js (an environment derived from the browser), we can develop almost any type of application imaginable:

- Desktop applications with web technologies by using for example NW.js (http://nwjs.io/) and Electron (http://electron.atom.io/). These technologies usually wrap the browser so that we can build desktop UIs using standard HTML, CSS, and JavaSript (so we can rely on our core JavaScript and browser knowledge), with an additional Node.js support that makes it possible to interact with the file-system. This enables us to build truly platform independent desktop applications that have the same look and feel on Windows, Mac, and Linux.

- Hybrid mobile apps also enable us to take the same knowledge and use it for the development of mobile apps (for example by using Ionic http://ionicframework.com/). Similarly to desktop apps built with web technologies, frameworks for hybrid mobile apps also use a wrapped browser but with additional platform specific APIs that allow us to interact with the mobile platform.
- Server-side applications and applications for embedded devices with Node.js, an environment derived from the browser that uses a lot of the same underlying principles as the browser does. For example, Node.js executes JavaScript code and relies on events.

As you can see, it doesn't matter if your target application is a standard desktop application, a mobile application, a server-side application, or even an embedded application – all these different types of applications can be developed with the same underlying principles that are behind standard client-side web applications that we will put our special focus on. By understanding how the core mechanics of JavaScript work (notice how the JavaScript mechanics are shared between browsers and Node.js in Figure 1.6), and by understanding the core APIs provided by the browsers, such as events (which also share a lot in common with mechanisms provided by Node.js) you can boost your development skills across the board; in the process becoming a more versatile developer that has the knowledge and the understanding of tools to tackle a wide variety of problems.

1.5 Summary

In this chapter, you saw that:

- Client-side web applications are one of the most popular application domains, and the concepts, tools, and techniques once used only for their development have also permeated other application domains. This means that, deeply understanding the foundations on top of which client-side web applications have been built will enable you to develop applications for a wide variety of different domains.
- In order to better ourselves as developers, we have to gain a deep understanding of the core mechanics of JavaScript, as well as the infrastructure provided by the browsers.
- At the same time, we have to be aware of the fact that cross-browser web application development is hard.

This exploration will certainly be informative and educational—let's enjoy the ride!

2

Building the page at runtime

This chapter covers:

- Different steps in the lifecycle of a web application
- How HTML code is processed in order to produce a web page
- The exact order of executing JavaScript code
- How to achieve 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 which executes our JavaScript code. In order to have a strong base from which we'll 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 are going to 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 the much needed dynamicity to our pages, and finally we'll take a look into how events are handled in order to develop interactive applications that respond to our user's actions.

While we'll focus especially on how the browser goes through this process, the exact same knowledge works for both hybrid mobile web apps and for desktop apps built with web technologies, which means that understanding these concepts will help you become a better developer across the board.

Let's dive in!

2.1 The lifecycle overview

The lifecycle of a typical client-side web application begins with the user typing in a URL in the browser's address bar or clicking on a link, shown in the upper left of figure 2.1.

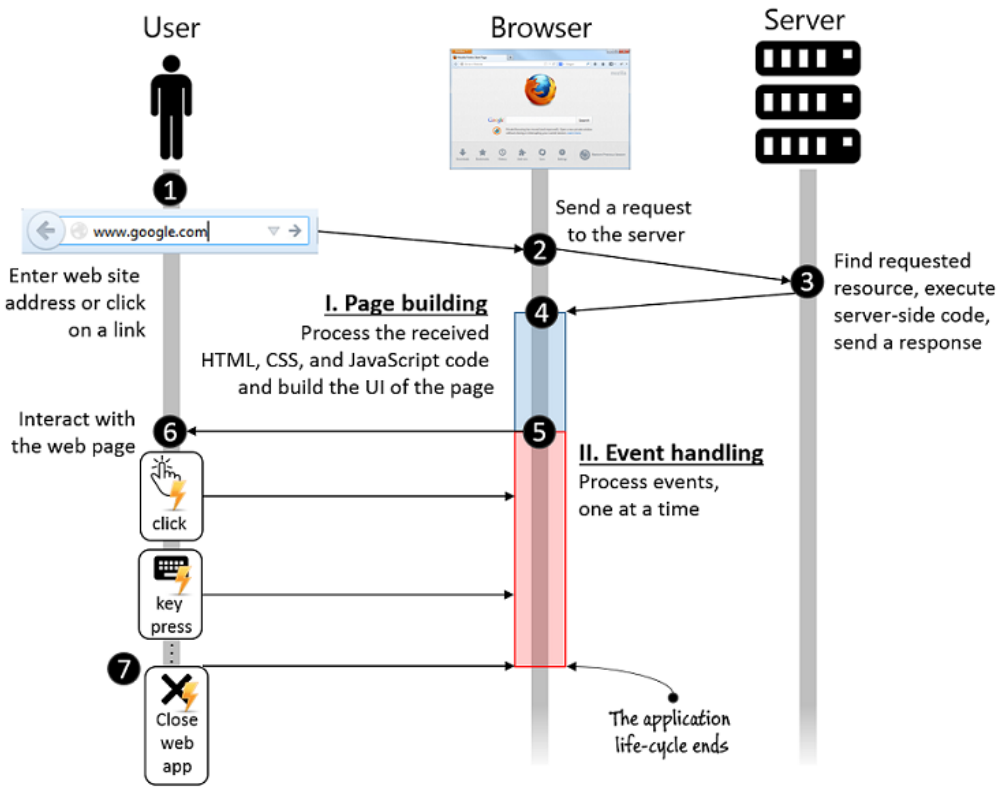


Figure 2.1 The lifecycle of a client-side web application starts with the user specifying a web site address (or clicking on a link) and ends when the user leaves the web page. It is composed out of two steps: *Page building* and *Event handling*.

On behalf of the user, the browser formulates a request that is sent to a server **#2**, which processes the request **#3** and formulates a response that is usually composed of HTML, CSS, and JavaScript code. The moment in which the browser receives this response **#4** is when our client-side web application truly starts coming to life.

As client-side web applications are 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. Set up the user interface – ***page building.***

2. Enter a loop **#5** waiting for events to occur **#6** and start invoking event handlers – ***event handling.***
3. The lifecycle of the application ends when the user closes or leaves the web page **#7**.

Now, let's take a look at an example web application with a simple UI that reacts to user actions, an application that we'll use throughout this chapter.

Listing 2.1 A small web application with a GUI reacting to events

```
<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){                    #A
        var messageElement = document.createElement("li");      #A
        messageElement.textContent = message;                   #A
        element.appendChild(messageElement);                    #A
      }                                                         #A

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

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

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

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

#A – Define a function that adds a message to an element
#B – Attach a mouse move event handler to `body`
#C – Attach a mouse click event handler to `body`

In listing 2.1, we first define two CSS rules `#first` and `#second` that specify the text color for the elements with the id `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 already 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 will be executed every time the user moves a mouse and that will add 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 on the page, will log 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 the following figure.

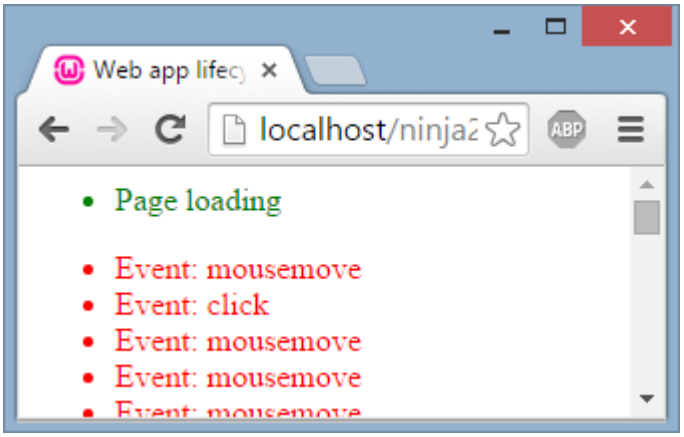


Figure 2.2 When running code from listing 2.1 messages are logged depending on user actions.

We'll be using this example application to explore and illustrate the differences between different phases of web application lifecycle. Let's start with the first phase – the *page building phase*.

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 that was 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:

- Parsing the HTML and building the Document Object Model (DOM)
- Executing JavaScript code
- The first step, *parsing HTML and building the DOM*, is performed when the browser is processing HTML nodes, while the second step, *executing JavaScript code*, is performed whenever a special type of HTML element, the `script` element (that contains 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 the following figure.

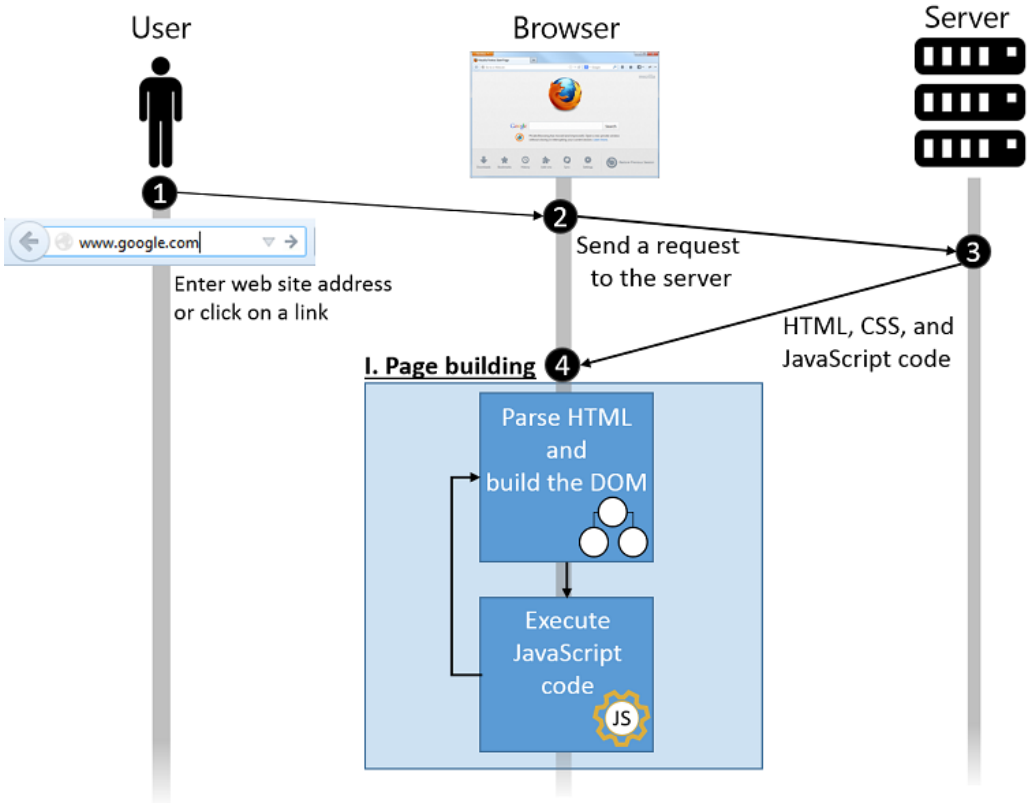


Figure 2.3 The page building phase starts when the browser receives the code of the page, and is performed in two steps: i) parsing the HTML and building the DOM, and ii) executing JavaScript code.

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 basis on top of which the browser will build the UI of the page. The browser does this by parsing the HTML code, one HTML element at a time, and building a document object model (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 our example page that is built until the first `script` element is reached.

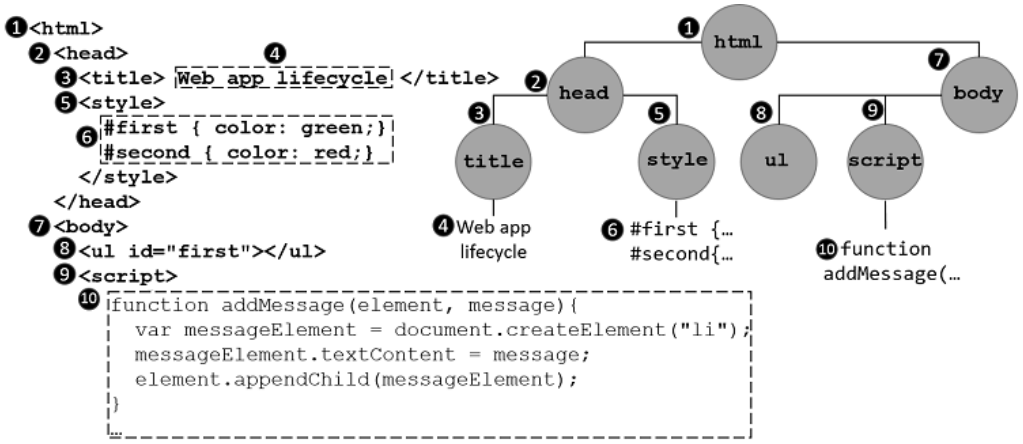


Figure 2.4 By the time the browser encounters the first script element, it has already created a DOM with multiple HTML elements (the nodes on the right).

Notice how the nodes in figure 2.4 are organized in a way that each node, except for the first one, the root `html` node **#1**, has exactly one parent. For example, the `head` node **#2** has the `html` node **#1** as its parent. At the same time, a node can have any number of children. For example, the `html` node **#1** has two children: the `head` node **#2** and the `body` node **#7**. Children of the same element are called siblings. (The `head` node **#2** and the `body` node **#7** are siblings.)

It is important to emphasize that, while the HTML and the DOM are closely linked, with the DOM being constructed from HTML and all, they are not one and the same. You should think of the HTML code as a *blueprint* that the browser follows when constructing the initial DOM – the UI – of the page. The browser can even fix problems that it finds with this blueprint in order to create a valid DOM. Let's consider an example as shown in figure 2.5.

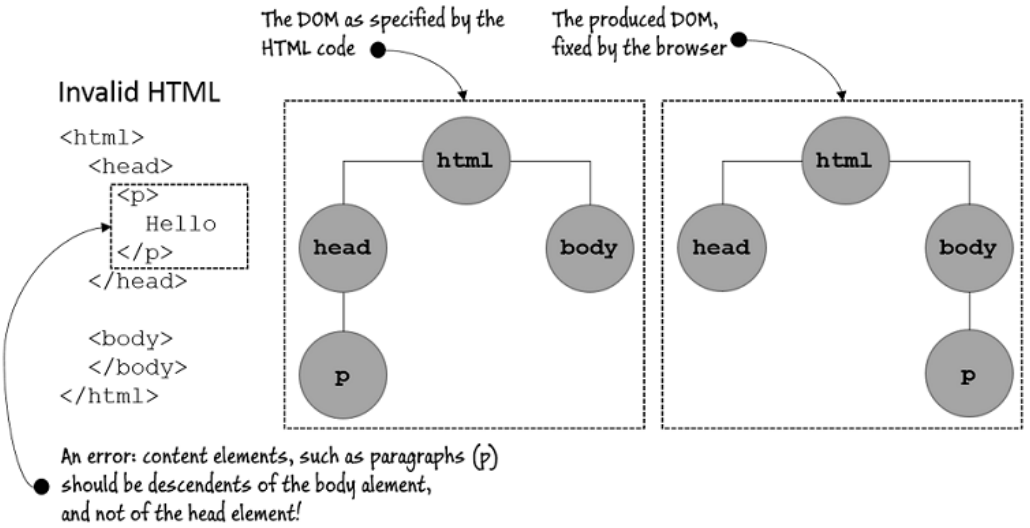


Figure 2.5 An example of "invalid" HTML that is fixed by the browser while the DOM is constructed.

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

HTML specification and the DOM specification

Both HTML and the DOM are specified by the World Wide Web Consortium (W3C). The current version of HTML is HTML5, whose specification is available at http://www.w3.org/TR/html5/ (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, with the current version being DOM3, whose specification is available at http://www.w3.org/DOM/DOMTR (again, Mozilla has prepared a report that can be found at https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model).

During DOM 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 with 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's V8, or IE's (Edge's) Chakra. As 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 our page is contained. The `window` object is *the* global object through which all other global objects, global variables (even our user defined ones), and browser APIs are accessible. One of the most important properties of the global `window` object is `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 take a look at a snippet of code from listing 2.1:

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

In this case, we use the global `document` object to select an element with the id first from the DOM, and we 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 itself from the DOM.

Browser APIs

Throughout the book, we'll be using a number of browser built-in objects and functions (`window` and `document` being one of them). 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 that basic understanding of the global objects provided by the browser, let's take a look at two different types of JavaScript code that define exactly when that code will be executed.

DIFFERENT TYPES OF JAVASCRIPT CODE

In JavaScript, we broadly differentiate between two different types of JavaScript code: *i) global* code and *ii) function* code. Listing 2.2 will help us understand the differences between these two types of code.

Listing 2.2 Different types of JavaScript code – global and function code

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

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

#A – function code is the code contained within a function
#B – global code is the code outside functions

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

These two code types also differ in their execution (we'll look into some additional differences later on, especially in chapter 5). The global code is executed automatically by the JavaScript engine (more on that soon) in a straightforward fashion, line by line, as it is encountered. For example, in the code of listing 2.2, the global code that defines the `addMessage` function, uses the built-in `getElementById` method to fetch the element with id `first`, and calls the `addMessage` function, would be executed one after another as they are encountered.

On the other hand, the 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 global code, causes the execution of the `addMessage` function code) 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. In this phase, this means executing the global JavaScript code contained in the script element (and of course, functions called by the global code will also be executed).

Let's go back to our example from listing 2.1.

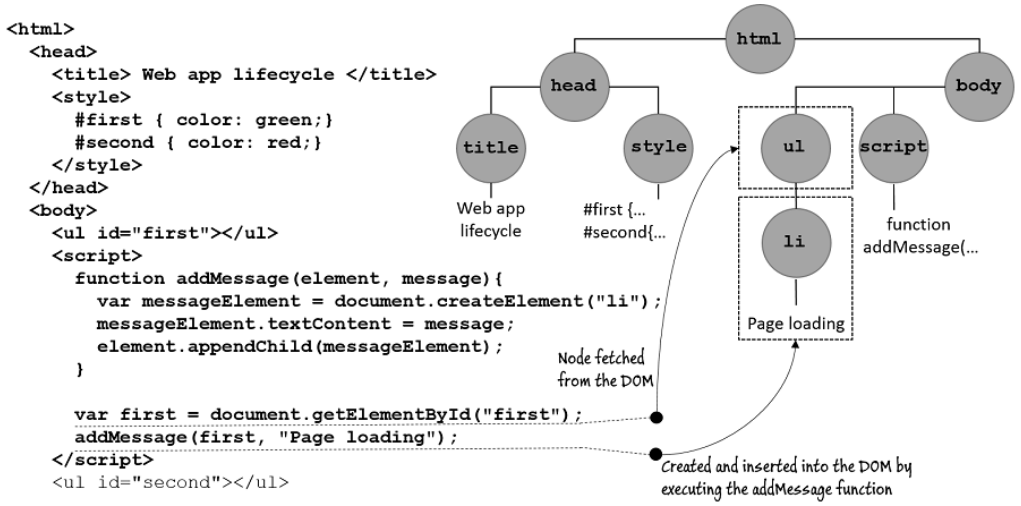


Figure 2.6 The DOM of the page after executing the JavaScript code contained in the script element

Figure 2.6 shows the state of the DOM after our global JavaScript code has been executed, but just in case, 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");
```

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, our JavaScript code has modified the current DOM by creating a new element and inserting it into the DOM. But in general, the JavaScript code can modify the DOM to any degree: it can create new nodes, and modify or remove existing DOM nodes. However, there are also some things that it cannot do, such as selecting and modifying elements that haven't yet been created. For example, we cannot select and modify the `ul` element with the id `second`, as that element is found after the current `script` node, and hasn't yet been reached and created. For this reason, you'll often see that people tend to put their `script`

elements at the very bottom of the page. In that way, we don't have to worry about whether a particular HTML element has been reached or not.

Once the JavaScript engine executes the last line of JavaScript code in the `script` element (in our example from 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, in 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 is important to note that the global state of our JavaScript application in the meantime persists. 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 whole lifecycle of the page.

These two steps of:

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 and the browser 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; for example: mouse moves, clicks, keyboard presses, and so on. For this reason, the JavaScript code executed during the page building phase, besides influencing the global application state and modifying the DOM, can also register event listeners (or handlers), functions that will be 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, there's only *one* teller window open! So the customers (events) only get processed 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 fiscal year while they're at the teller's window (we've all run into them!), to gum up the whole works.

Whenever an event occurs, the browser should execute the associated event handler function. However, there is no guarantee that we have extremely patient users that will always wait an appropriate amount of time before triggering another event. For this reason, the browser has to have a way of keeping track of the events that have occurred, but have yet to be processed. To do this, the browser uses an *event queue,* shown in figure 2.7.

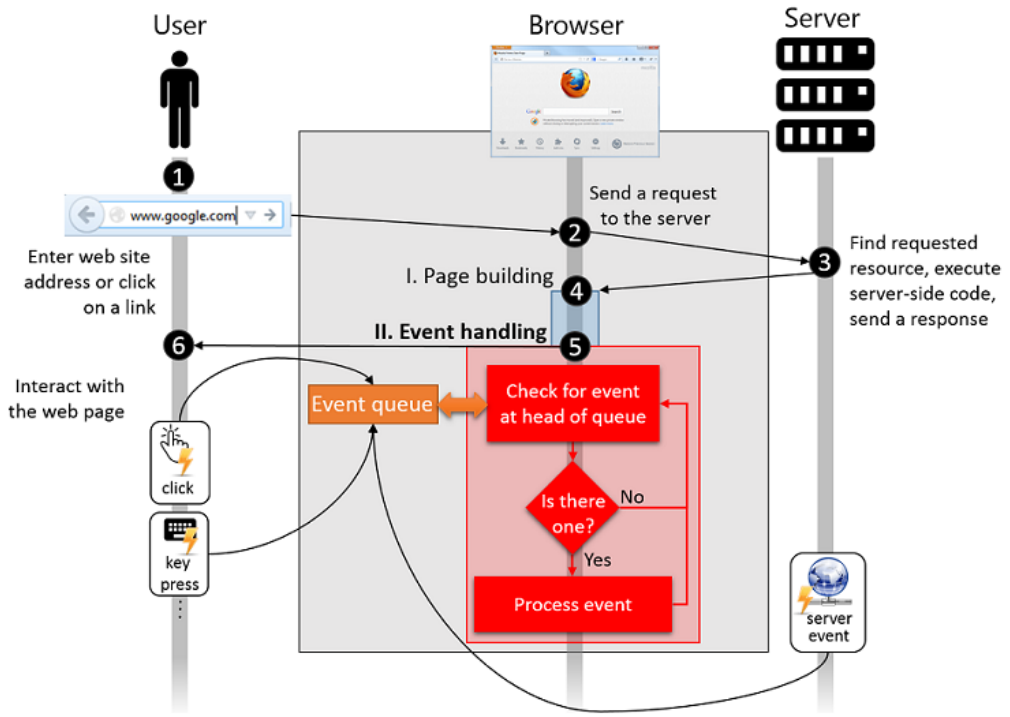


Figure 2.7 In the event handling phase, all events (whether coming from the user, such as mouse clicks or key presses; or coming from the server, such as AJAX events) are queued up as they occur and are processed as the single thread of execution allows.

All generated events (it doesn't matter if they are user generated like mouse moves or key presses, or even server generated such as AJAX events) are placed in the same event queue, in the order in which they are detected by the browser. As shown in the middle of figure 2.7, 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 is an event at the head of the event queue, the browser takes it and executes

it. During this execution, the rest of the events are patiently waiting in the event queue, for their turn to be processed.

Since 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!

It's important to note that the browser mechanism that puts the events *onto* the queue is external to our 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

The events, when they happen, can occur at unpredictable times and in an unpredictable order (it's tricky to force our users to press keys or click in some particular order). So we say that the handling of the 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 (`XMLHttpRequest` events, server-side events)
- User events, such as mouse clicks, mouse moves, or key presses
- Timer events, such as when a timeout expires or an interval fires

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

The concept of event handling is central to on-page JavaScript, and it's something we'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 that we place onto our page is going to execute as the result of some event.

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

2.3.2 Registering event handlers

As we've already mentioned, event handlers are functions that you as a developer want executed whenever a particular event occurs. In order for this to happen, you have to notify the browser that you are interested in an event. This is called *event handler registration*. In client-side web applications there are two different ways of registering events:

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

For example, by writing the following code:

```
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 to the right hand side of the assignment operator looks a bit funky to you right now; we'll be talking at great length about functions in later chapters.) Similarly, if you want to register a handler for the click event on the document's body, you could write something along these lines:

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

Assigning functions to special properties is an easy and straight-forward way of registering event-handlers. However, we don't recommend that you register your event handlers in this way, because it this comes with a drawback: it's only possible to register one function handler for a particular event. This means that it's very easy to overwrite your event handler functions, which can be a little frustrating. While this is a limitation that can be circumvented by manually keeping track of a list of functions that should be called when an event occurs, there is a built-in way to achieve the same functionality, by using the `addEventListenerMethod`.

Let's go back to an excerpt of our example listing 2.1.

Listing 2.3 Registering event handlers

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

  document.body.addEventListener("click", function(){           #B
    var second = document.getElementById("second");
    addmessage(second, "Event: click");
  });
</script>
```

#A – register a handler for the mousemove event
#B – register a handler for the click event

In listing 2.3, we have used the built-in `addEventListener` method on an HTML element to specify the type of event (`mousemove` or `click`) and the event handler function. In the example application, this means that whenever a mouse is moved over our page, the browser will call a function that will add a message `"Event: mousemove"` to the list element with the id `second` (a similar message `"Event: click"` will be added to the same element, whenever the body is clicked).

Now that we know how to set up event handlers, let's recall the simple flowchart we considered 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, and 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 our application from listing 2.1, and study figure 2.8 which shows an example execution in which a very quick user has moved and clicked a mouse.

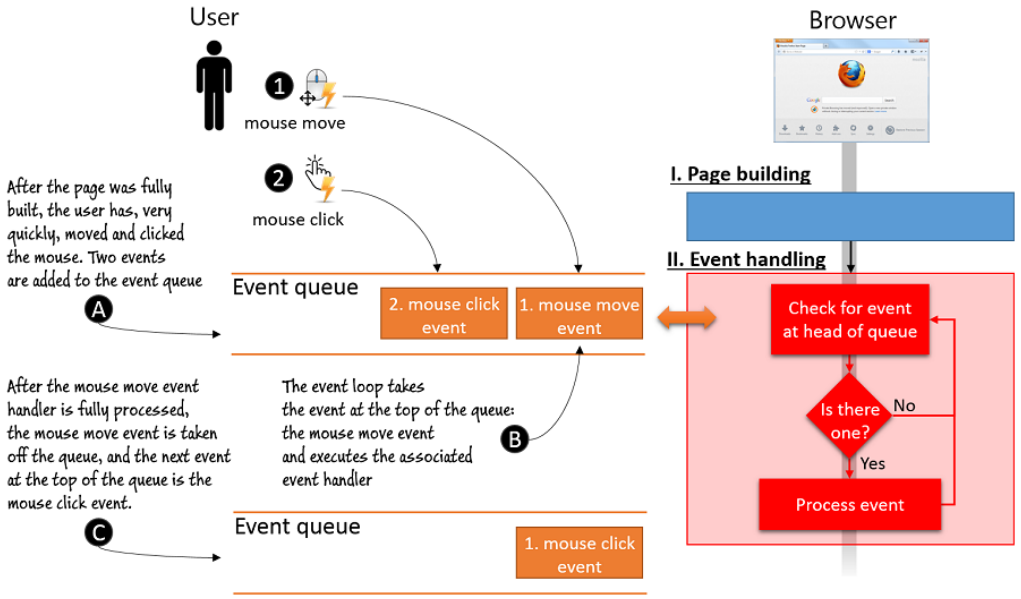


Figure 2.8 Example of an event handling phase where two events: mouse move and mouse click are handled.

Let's examine what's going on here.

As a response to these user actions, the browser puts the mouse move and click events onto the event queue, in the order in which they have occurred: first the mouse move event, and then the click event **#A**.

In the event handling phase, the event loop then checks the queue, sees that there is a mouse move event at the front of the queue and executes the associated event handler **#B**. While the mouse move handler is being processed, the click event simply waits in the queue for its turn. When the last line of the mouse move handler function has finished executing and the JavaScript engine exits the handler function, the mouse move event is fully processed **#C**, and the event loop again checks the queue. This time, at the front of the queue, the event loop finds the mouse click event and processes it. Once the click handler has finished executing, there are no new events in the queue, and the event loop simply 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 our DOM in figure 2.9.

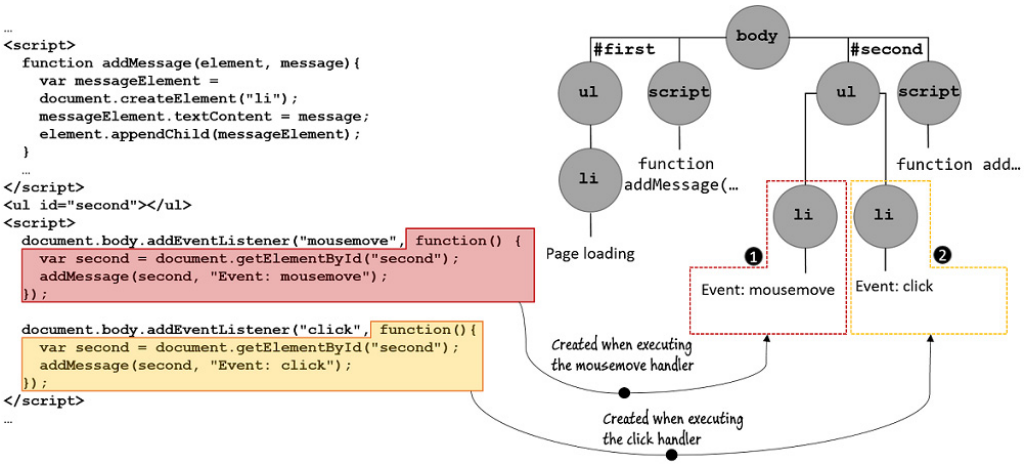


Figure 2.9 The DOM of the example application after handling the mousemove and click events.

The execution of the mouse move handler selects the second list element with id `second`, and by using the `addMessage` function adds a new list item element **#1** with the text "Event: mousemove". Once the execution of the mouse move handler is done, the event loop executes the click handler, which leads to the creation of another list item element **#2**, 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 chapter we'll take a look at some of the best practices in JavaScript application development.

2.4 Summary

In this chapter we've learned:

- The HTML code, received by the browser, is used as a blueprint for creating the DOM, an internal representation of the UI of our client-side web application.
- We use JavaScript code to dynamically modify the DOM to bring dynamic behavior to our web applications.
- The execution of client-side web applications is performed in two different phases:
  - *Page building*, in which the HTML code is processed to create the DOM, and where 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 will be executed when a particular event occurs (for example, a mouse click or a keyboard press). Registering event

handlers is easy, just use the built-in `addEventListener` method.

- *Event handling*, in which various events are processed one at a time, in the order in which they were generated. The event handling phase heavily relies on the event queue in which all events are stored, in the order in which they have occurred. In the event-handling phase, 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.

3

Arming with testing and debugging

This chapter covers

- Tools for debugging JavaScript code
- Techniques for generating tests
- Building a test suite
- Surveying some of the popular testing framework

In the previous chapter, we've made sure that we're all on the same page regarding how client-side web applications work and we've established a common vernacular that we're going to use from now on. But before going into the core mechanics of JavaScript, we must properly set our foundations. This is why we're going take a look at some of the fundamental techniques for the development of client-side web applications: debugging and testing.

"Testing?" you might be wondering, "We haven't even written any real code yet, and you want to talk about *testing*?"

Yes. Yes we do.

Constructing effective test suites for our code is always important, so we're going to discuss it *now*, before we go into any discussions on coding. After all, if we don't test our code, how do we know that it actually *does* what we intended it to do? 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 for situations where external factors have the potential to affect the operation of your 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 programmers need

to deal with), but we also have the problem of determining if our code works in all the browsers that we choose to support.

We'll discuss the problem of cross-browser development in depth when we look at cross-browser strategies in chapter 17, but for now, it's vital that the importance of testing be emphasized and testing strategies defined; we'll be using these strategies throughout the rest of the book. For example, almost all code examples that we'll present throughout the book will be in the form of tests, which you can easily run and check that they actually behave as we claim they do.

In this chapter, we're going to look at some tools and techniques for debugging JavaScript code, for generating tests based upon those results, and for constructing a test suite to reliably run those tests.

Let's get started.

3.1 Web Developer tools

For a long time, the development of JavaScript applications was hindered by the fact that the only way to debug our JavaScript code was to scatter `alert` statements, that would notify us about the value of the alerted expression, all around the code that's acting strangely. As you might imagine, debugging (hardly ever a fun activity), was made even more difficult by this lack of basic debugging infrastructure.

Luckily for us, around 2007, Firebug, an extension to Firefox, was developed. Firebug holds a special place in the hearts of many web developers, since it was the first tool that has provided us with a debugging experience that closely matched debugging in state of the art integrated development environments (IDE's), 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 Microsoft Edge; *WebKit Inspector*, included in Safari; *Firefox DevTools*, 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: https://getfirebug.com/ in Firefox and following the given instructions. Firebug is shown in the following figure.

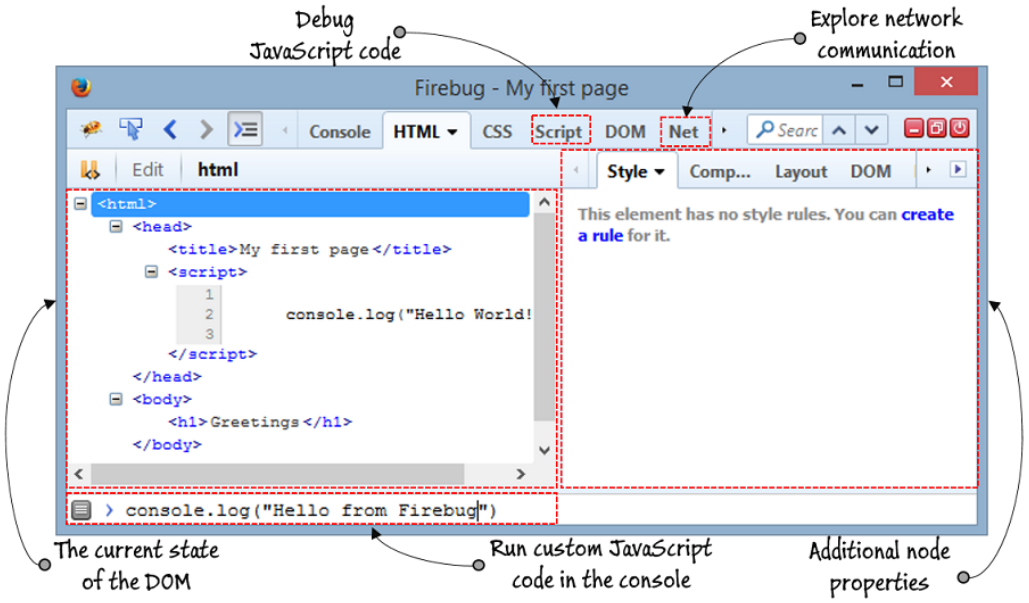


Figure 3.1 Firebug, available only in Firefox, was the first advanced debugging tool for web applications.

Firebug offers a number of advanced debugging functionalities, some of which it has even pioneered. For example, we can easily explore the current state of the DOM by using the "HTML" pane (the pane shown in figure 3.1), run custom JavaScript code within the context of the current page by using the console (the bottom of figure 3.1), explore the state of our JavaScript code by using the "Script" pane, and even explore network communications from the "Net" pane.

FIREFOX DEVTOOLS

In addition to Firebug, if you're a Firefox user, you can use the built-in Firefox DevTools, shown in the following figure.

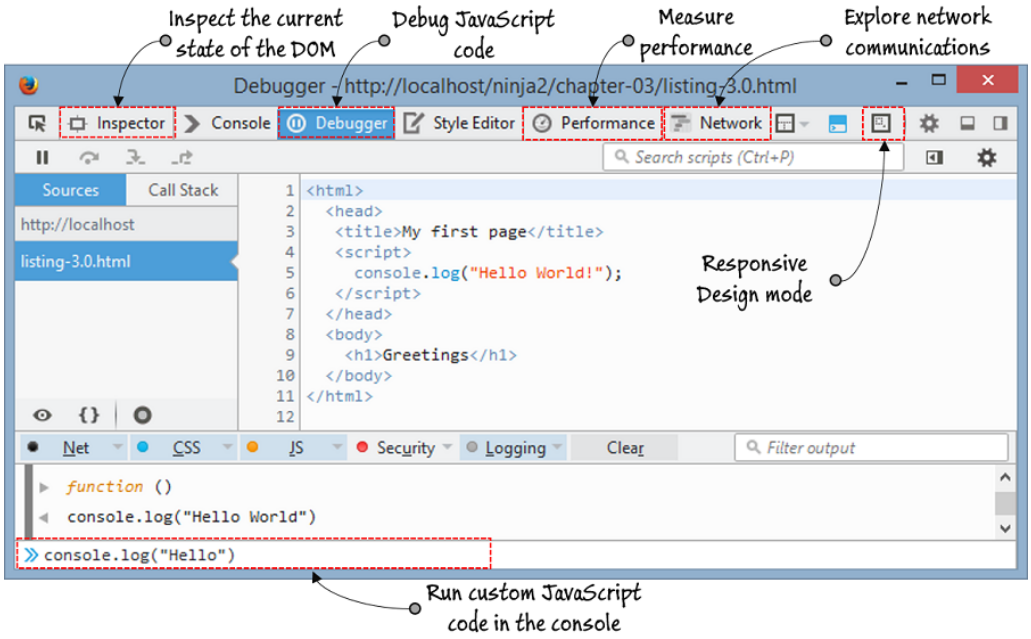


Figure 3.2 Firefox DevTools, built into Firefox, offer all the Firebug features and then some.

As you can see from figure 3.2, the general look and feel of Firefox DevTools is very similar to Firebug (apart from some minor layout and label differences, for example the "HTML" pane from Firebug is called "Inspector" in Firefox DevTools).

Firefox DevTools are built by the Mozilla team, and they have took advantage of this close integration with Firefox, by bringing some additional useful features, such as the "Performance" pane with which we can get detailed insight about the performance of our web applications (more on this in chapter 13). In addition, Firefox DevTools are built with the modern web in mind. For example, they offer the "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, since nowadays users access our web applications not only from their PC's, but also from mobiles, tablets, and even TV's.

F12 DEVELOPER TOOLS

In case you're in the Internet Explorer (IE) camp, you'll be happy to know that IE and Microsoft Edge (the successor to IE) also offer their own developer tools, the F12 developer tools. (Quickly, try to guess which key toggles them on and off.) The F12 developer tools are shown in the following figure.

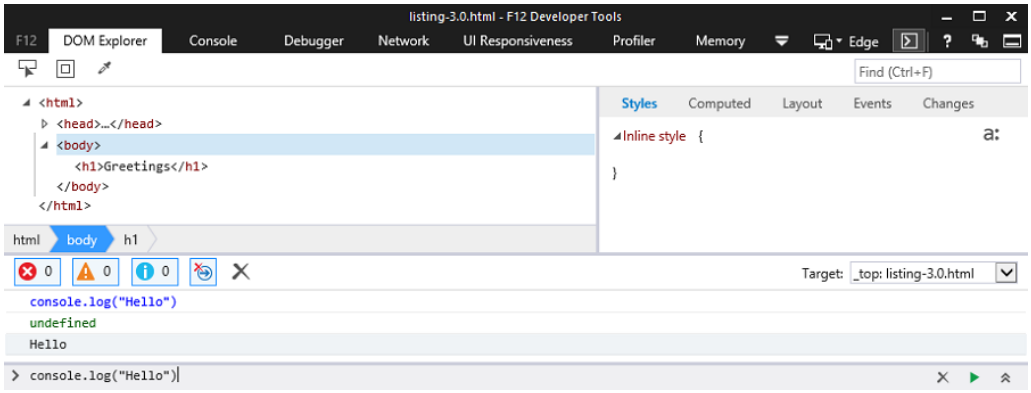


Figure 3.3 F12 developer tools (toggled by pressing F12) are available in Internet Explorer and Edge.

Again, notice a number of similarities between the F12 developer tools and Firefox's DevTools (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 3.3), 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").

WEBKIT INSPECTOR

If you are a Mac OS user, you can use WebKit Inspector, which is offered by Safari, as shown in the following figure.

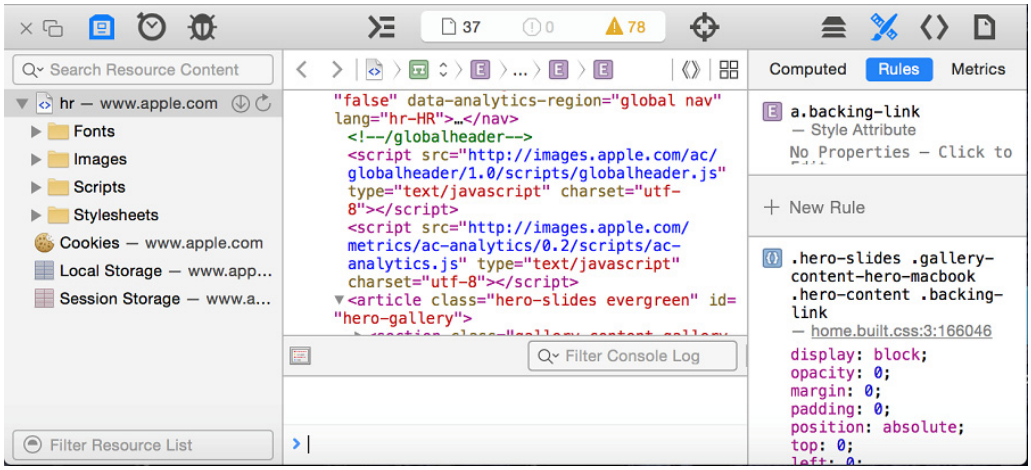


Figure 3.4 WebKit Inspector, available in Safari

While the UI of Safari's WebKit Inspector is slightly different than that of F12 developer tools or Firefox's DevTools, rest assured that the WebKit Inspector also supports all important debugging features.

CHROME DEVTOOLS

We'll complete our little survey of different 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 3.5, the basic UI and features are again quite similar to the rest of the developer tools.

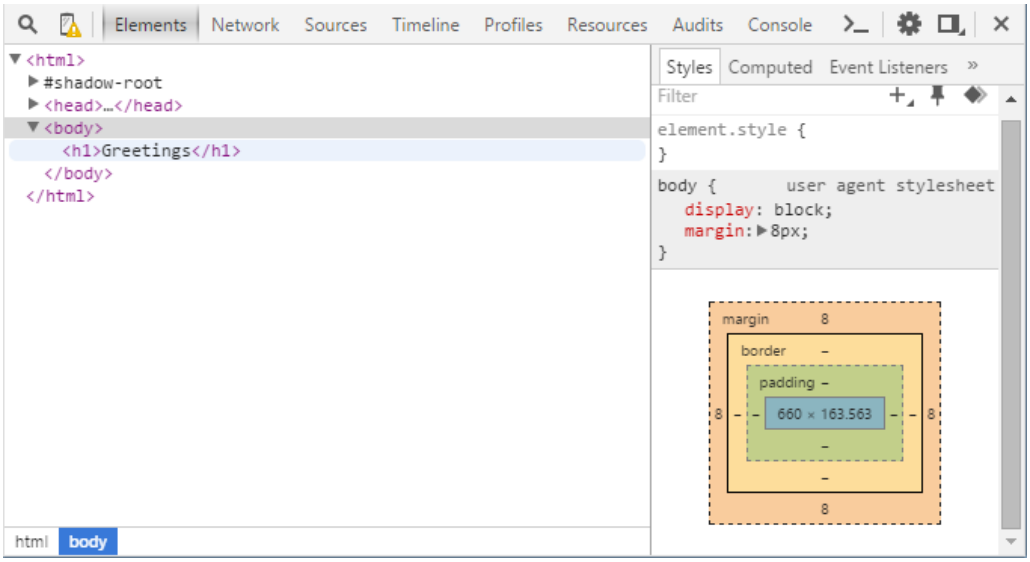


Figure 3.5 Chrome DevTools, available in Chrome and Opera

Throughout this book, we'll use Chrome DevTools, simply for the sake of convention. But, as you've seen throughout this section, most of developer tools offer basically the same features (and if one of them offers something new, the others catch up quickly), so you can just as easily use the developer tools offered by your browser of choice.

Now that we had an introduction to the tools that we can use for debugging our code, let's explore some of the debugging techniques.

3.2 Debugging code

A significant portion of time developing software is spent on removing annoying bugs. While this can sometimes be interesting, almost like solving a whodunit mystery, normally we want our code working correctly and bug-free as soon as possible.

There are two important aspects to debugging JavaScript:

- *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 are 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.

3.2.1 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 wanted to know what the value of a variable named `x` was at a certain points of program execution, we might write something like this:

Listing 3.1 Logging the value of variable x at different points of program execution

```
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>
```

The result of executing the code from listing 3.1 in the Chrome browser with the JavaScript console enabled is shown in figure 3.6.

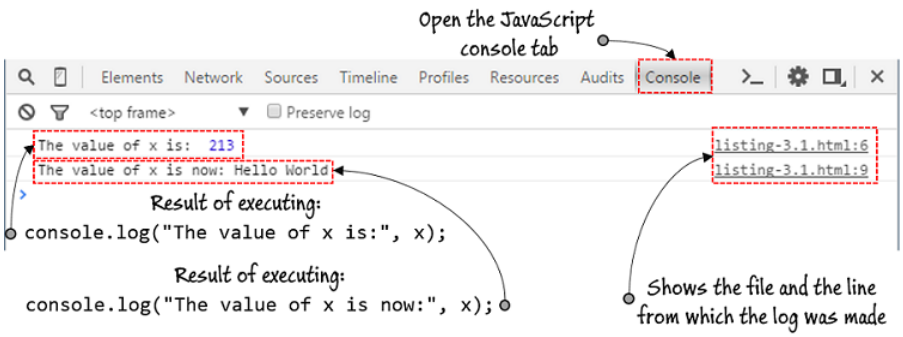


Figure 3.6 Logging lets us see the state of things in our code as it is running. In this case, we can see the value of 213 is logged from line 6, and the value of "Hello World" from line 9 of our listing-3.1.html. All developer tools, including the shown Chrome Dev Tools, have a console tab for logging purposes.

As you can see from figure 3.6 our browser logs the messages directly to the JavaScript console, showing both the logged message and the line in which the message was logged.

This was a pretty simple example of logging a value of a variable at different points of program 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. We'll take another look at logging in chapter 9.

Logging is all well and good for seeing what the state of things might be while the code is running, but sometimes we'll want to stop the action and take a look around.

That's where breakpoints come in.

3.2.2 Breakpoints

Breakpoints are a somewhat more complex concept than logging, but they possess a notable advantage over logging: 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 3.2 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); #A
  </script>
  </head>
  <body>
  </body>
</html>
```

#A Line where we'll break

If we were to set a breakpoint using the Chrome's DevTools on the annotated line that calls the `logGreeting` function in listing 3.2 (by clicking on the line number gutter in the Debugger pane) and refresh the page to cause the code to execute, the debugger would stop the execution at that line and show us the display seen below in figure 3.7.

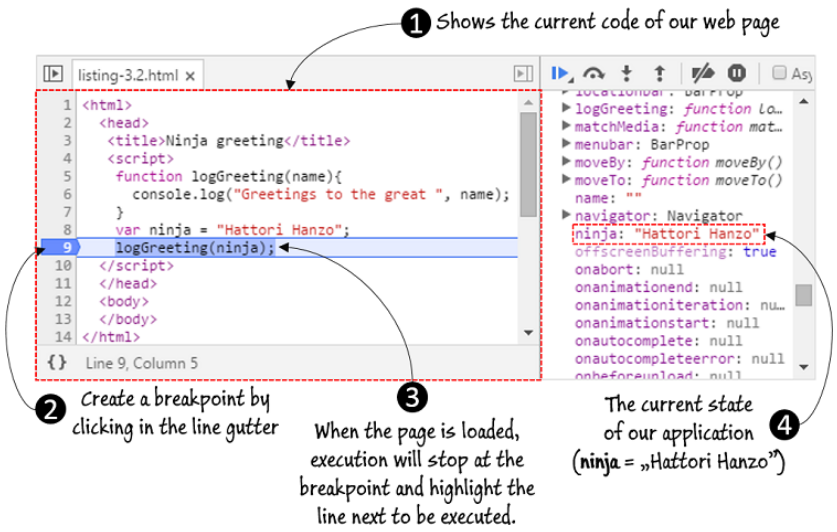


Figure 3.7 When we set a breakpoint on a line of code (by clicking on the line gutter) and load the page, the browser will stop executing JavaScript code before that line is actually executed. Then you can leisurely explore the current state of the application in the pane on the right.

Note how 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 break-pointed line is actually executed; in this example, the call to the `logGreeting` function has yet to be executed.

STEPPING INTO A FUNCTION

If we were 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. So while our execution is paused on the logGreeting call (with a breakpoint that we've previously set) we can click on 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. The result is shown in figure 3.8.

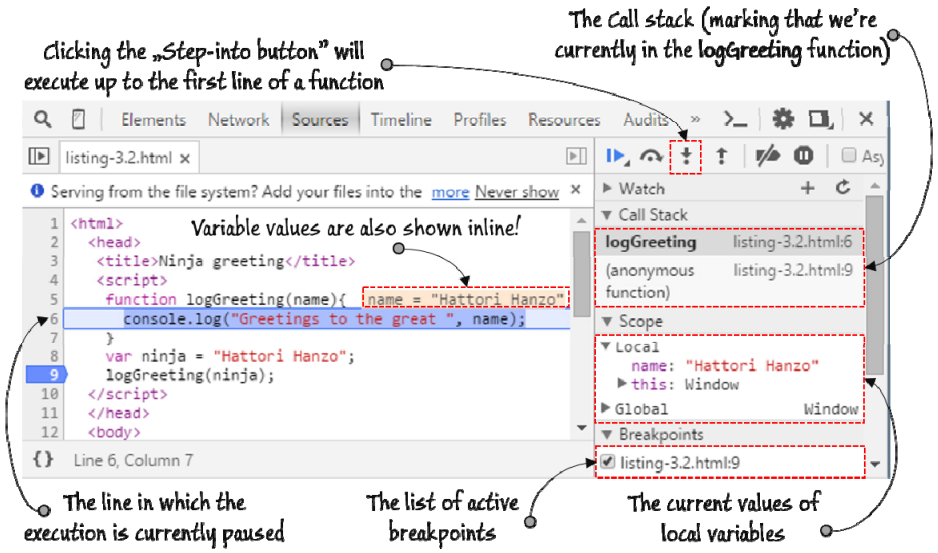


Figure 3.8 Stepping into a function lets us see the new state in which the function is executed. We can explore the current position by studying the Call Stack, and the current values of local variables.

Note how the look of Chrome DevTools has changed a bit (when compared to figure 3.7) 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 a 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 how in the upper right corner of figure 3.8, there is a *Call stack* pane which shows that we are currently within the `logGreeting` function, which was called by global code (we'll revisit the Call Stack later, in chapter 5).

Step over and Step out of

In addition to the step into command, we can also use the *step over* and *step out* commands.

The *step over* command executes our code line-by-line. If the code in the executed line contains a function call, the debugger will simply step over the function (the function will be executed, but the debugger will not jump into its code).

In the case when we have paused the execution in a function, pressing 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 the function in which we were previously at.

CONDITIONAL BREAKPOINTS

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 3.3 Counting Ninjas and conditional breakpoints

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      for(var i = 0; i < 100; i++){
        console.log("Ninjas: " + i); #A
      }
    </script>
  </head>
  <body>
  </body>
</html>
```

#A What if we wanted to explore the application state when counting the 50th ninja? Should we have to tediously go through the first 49?

Now 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 break-pointed line is executed, a *conditional breakpoint* causes the debugger to break only if an expression associated with the conditional breakpoint is satisfied. You can set a conditional breakpoint by right-clicking in the line-number gutter and choosing "Add Conditional Breakpoint" (see figure 3.9 below, for how it is done in Chrome).

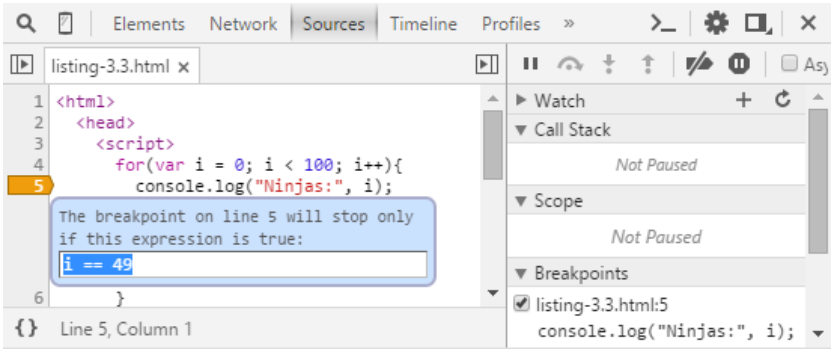


Figure 3.9 Right-click in the line-number margin to set a breakpoint as conditional (notice how conditional breakpoint are shown in orange).

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 in immediately to the point in the application execution in which we are interested in, and simply ignore the less interesting ones.

So far, we've seen how to use different developer tools, from different browsers, in order to debug our code with logging and breakpoints. These are all great tools that help us locate specific bugs and achieve a better understanding of the execution of a particular application. But, in addition to this, we also want to have an infrastructure in place that will help us detect bugs as soon as possible. This can be achieved with testing.

3.3 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 quite possible to have an extensive test suite that doesn't really 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 run repeatedly should always produce the exact same results. If test results are nondeterministic, how would we know which results are valid and which are invalid? Additionally, reproducibility ensures our tests aren't dependent upon external factors issues 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 only be influenced by the specific code that we're testing.
- *Independence*— Our tests should execute in isolation. We must avoid making the results from one test dependent upon another. Breaking tests down into the smallest possible units will help us determine the exact source of a bug when an error occurs.

There are a number of approaches that can be used for constructing tests; the two primary approaches are *deconstructive* and *constructive*:

- *Deconstructive test cases*— Deconstructive test cases are created when existing code is whittled down (deconstructed) to isolate a problem, eliminating anything that's not germane to the issue. This helps us to 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*— With a constructive test case we start from a known good, reduced case and build up until we're able to reproduce the bug in question. In order 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 3.4 A reduced DOM test case for jQuery

```
<script src="dist/jquery.js"></script>
<script>
  $(document).ready(function() {
    $("#test").append("test");
  });
</script>
<style>
  #test { width: 100px; height: 100px; background: red; }
</style>
<div id="test"></div>
```

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 jsbin (http://jsbin.com/). All of them 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 3.10.

Figure 3.10 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 since we 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 automatize our tests as much as possible, let's look into how to achieve that.

3.4 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 is constructed. Perhaps surprisingly, JavaScript testing frameworks are really easy to construct.

One would have to ask, though, "Why would I want to build a new testing framework?" For most cases, it isn't really necessary to write your own JavaScript testing framework, since there are already a number of good-quality ones to choose from (as we will soon see). But building your own test framework can serve as a good learning experience.

3.4.1 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 that describes 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 next listing.

Listing 3.5 A simple implementation of a JavaScript assertion

```
<html>
  <head>
    <title>Test Suite</title>
    <script>
      function assert(value, desc) {                          #A
        var li = document.createElement("li");                #A
        li.className = value ? "pass" : "fail";               #A
        li.appendChild(document.createTextNode(desc));        #A
        document.getElementById("results").appendChild(li);   #A
      }                                                       #A
      window.onload = function() {
        assert(true, "The test suite is running.");     #B
        assert(false, "Fail!");                         #B
      };
    </script>
    <style>
      #results li.pass { color: green; }                  #C
      #results li.fail { color: red; }                    #C
    </style>
```

```
  </head>
  <body>
    <ul id="results"></ul>                                    #D
  </body>
</html>
```

#A Defines the assert method
#B Executes tests using assertions
#C Defines styles for results
#D Holds test results

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 upon 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.

This function is simple, but it will serve as a good building block for future development, and we'll be using this `assert` method throughout the book to test various code snippets, verifying their integrity.

The result of running our test suite in Chrome is shown in figure 3.11.

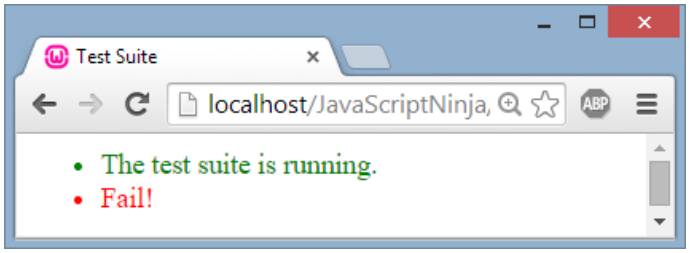


Figure 3.11 The result of running our first test suite

TIP If you are looking for something quick, you can use the built-in `console.assert()` method (see figure 3.12).

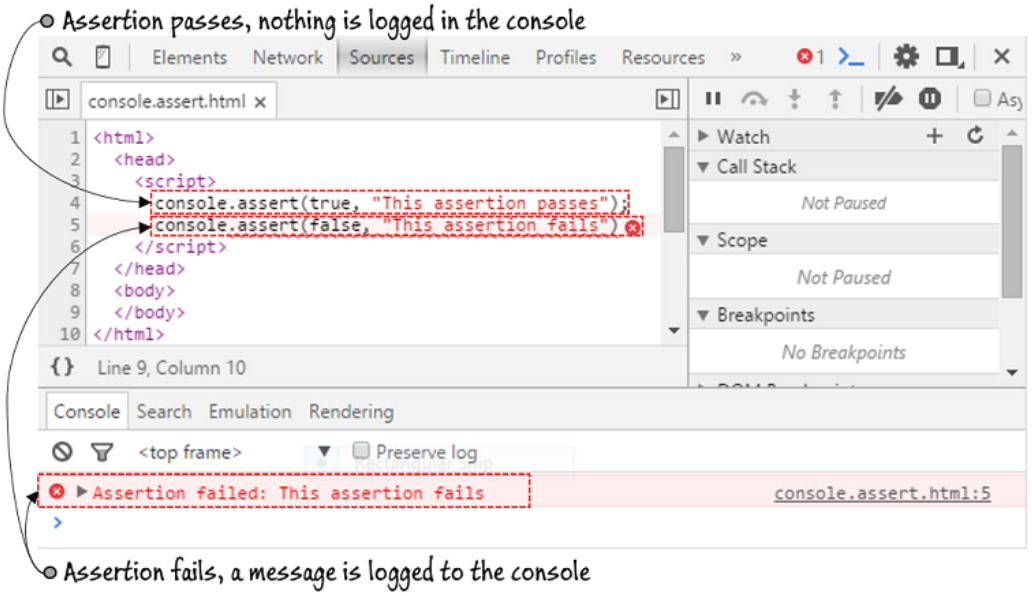


Figure 3.12 You can use the built in console.assert as a quick way for testing code. Only if an assertion passes, the fail message is logged to the console.

Now that we have built our own rudimentary testing framework, let's meet some of the widely available, more popular testing frameworks.

3.4.2 Popular testing frameworks

A test framework should serve as 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: displaying 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 a number of features that we might want to look for in a JavaScript unit-testing framework, depending upon 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 timeouts
- 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 that was 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 have developed a function that accurately says "Hi" to a ninja.

Listing 3.6 QUnit test example

```
<html>
  <head>
    <link rel="stylesheet" href="qunit/qunit-git.css"/> #A
 <script src="qunit/qunit-git.js"></script>         #A
  </head>
  <body>
    <div id="qunit"></div> #B
 <script>
  function sayHiToNinja(ninja) {  #C
      return "Hi " + ninja;       #C
  }                               #C

  QUnit.test("Ninja hello test", function(assert){                      #D
      assert.ok( sayHiToNinja("Hatori") == "Hi Hatori", "Passed");      #E
      assert.ok( false, "Failed");                                      #F
  });
</script>
 </body>
</html>
```

#A Includes QUnit code and styles
#B Creates an HTML element that QUnit fills with test results
#C Declares the function that we want to test
#D Specifies a QUnit test case
#E Tests a passing assertion
#F Tests a failing assertion

When you open this example in a browser, you should get the results as shown in figure 3.13, with one passing assertion from executing line `sayHiToNinja("Hatori")`, and one failing assertion from `assert.ok(false, "Failed")`.

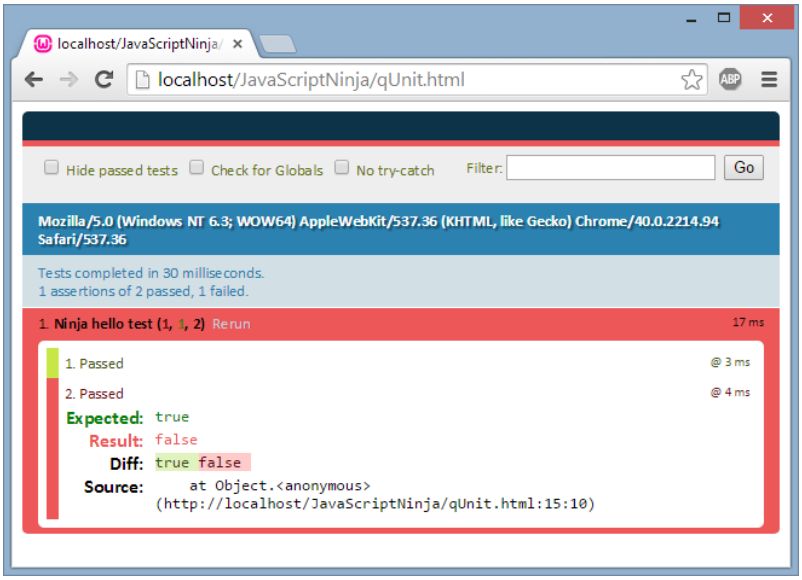


Figure 3.13 An example of a QUnit test run. As a part of our test, we have one passing and one failing assertion (1 assertions of 2 passed, 1 failed.) The displayed results put a much bigger emphasis on the failing test, to make sure we fix it as soon as possible.

More information on QUnit can be found at http://qunitjs.com/.

JASMINE

Jasmine is another popular testing framework, built on slightly different foundations thatn QUnit. The principal parts of the framework are:

- the `describe` function, which is used to describe test suites,
- the `it` function with which we specify individual tests, and
- the `expect` function which we use to check individual assertions.

The combination and naming of these functions are geared towards making the test suite almost conversational in nature.

For example, let's see how to test the `sayHiToNinja` function using Jasmine (Listing 3.6).

Listing 3.6 Jasmine test example

```
<html>
<head>
  <link rel="stylesheet" href="lib/jasmine-2.2.0/jasmine.css">  #A

  <script src="lib/jasmine-2.2.0/jasmine.js"></script>          #A
  <script src="lib/jasmine-2.2.0/jasmine-html.js"></script>     #A
  <script src="lib/jasmine-2.2.0/boot.js"></script>             #A
</head>
<body>
```

```
<script>
  function sayHiToNinja(ninja) {                                    #B
    return "Hi " + ninja;                                           #B
  }                                                                 #B

  describe("Say Hi Suite", function() {                             #C
    it("should say hi to a ninja", function() {                     #D
      expect(sayHiToNinja("Hatori")).toBe("Hi Hatori");             #E
    });

    it("should fail", function(){
      expect(false).toBe(true); #F
    })
  });
</script>
</body>
</html>
```

#A Includes Jasmine files
#B Declares the function that we want to test
#C Defines a test suite calls "Say Hi Suite"
#D Specifies a single test that checks our function
#E Asserts that our function produces the right result
#F Fail on purpose

The result of running this Jasmine test suite in the browser is shown in figure 3.9.

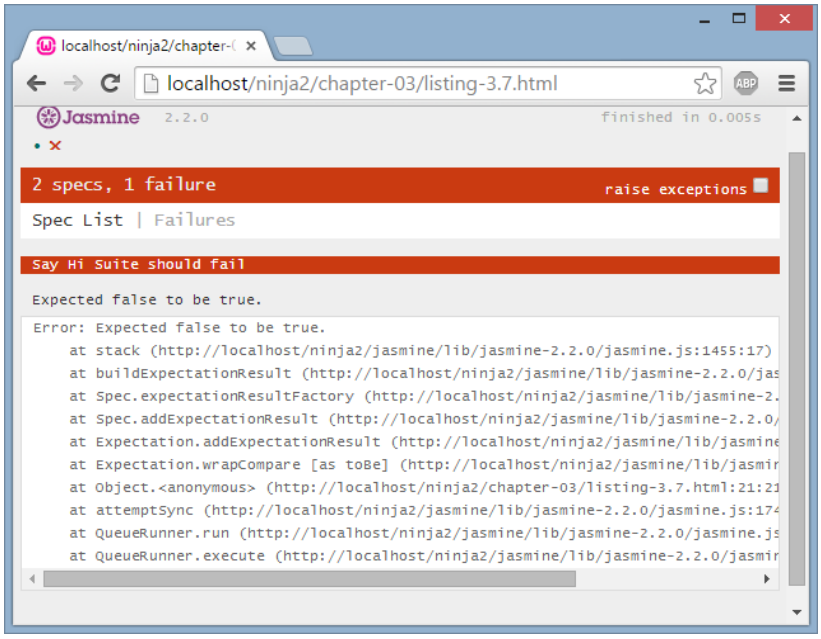


Figure 3.14 The result of running a Jasmine test suite in the browser. We have two tests: one passing and one failing (2 specs, 1 failure).

More information on Jasmine can be found at http://jasmine.github.io/.

MEASURING CODE COVERAGE

It's rather 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 is not 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 80% of our program code is executed by the test-suite, while 20% of our code is not. While we cannot be entirely sure that this 80% of code does not contain bugs (we might've missed an execution path that leads to one), we are completely in the dark about the 20% that was not even executed. This is why we should measure code coverage of our test suites.

In JavaScript development, there are several libraries that you can use to measure the coverage of your test suites, most notably blanketjs (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.

3.5 Summary

In this chapter we've learned that:

- We can debug our web pages by using web developer tools available in each major browser (Chrome DevTools in Chrome and Opera, Firefox developer tools in Firefox, WebKit Inspector for Safari, and F12 tools for IE and Microsoft Edge)
- Debugging is the act of removing bugs from our code and that there are two important aspects to debugging:
  - Logging, the act of printing out what's going on with our application, as the code is running.
  - Breakpoints, which allow us to temporarily pause code execution which enables us to leisurely explore the state of the application
- Testing your code is important and good tests should be: repeatable, simple, and independent of each other.
- A testing framework allows us to easily specify multiple, individual tests that can be run in bulk, easily and repeatedly.
- The core of a testing framework is usually a method named `assert` which takes a value, an expression whose premise is asserted, and a description that describes the purpose of the assertion. When called, the `assert` method notifies us whether the assertion was successful or not.
- We should strive to test as much code as possible. Code coverage is a metric that tells us the degree to which our tests cover our code.

4

Functions are fundamental

This chapter covers:

- Why understanding functions is so crucial
- How functions are first-class objects
- Defining functions
- The secrets of how parameters are assigned
- The context within a function

You might have been somewhat surprised, upon turning to this part of the book dedicated to JavaScript fundamentals, to see that the first topic of discussion is to be *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 the sophistication of all the code you'll ever write in JavaScript hinges upon this realization.

If you're reading this book, you're not a rank beginner and we're assuming that you know enough object fundamentals to get by for now (and we'll be taking a look at more advanced object concepts in chapter 8), 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 going to be devoted to thoroughly understanding functions in JavaScript.

Most importantly, in JavaScript, functions are *first-class objects*; that is, 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 we'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 also get an added benefit of not polluting the global namespace with unnecessary names. We'll continue our exploration of functions by going through different ways of defining functions in JavaScript. Next, we'll discuss both implicit and explicit function parameters and their relationship to values sent over function invocations. Finally, we'll conclude the chapter with a discussion on different ways of invoking functions, where we'll put a special focus on how the function context, i.e. the object on which the function is executed (represented by the `this` keyword) is determined.

During this exploration, we'll also focus on features introduced by the new ES 5 and 6 standards, such as arrow functions, which will make our code a bit more elegant.

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

4.1 What's with the functional difference?

One of the reasons that functions and functional concepts are so important in JavaScript is that the function is the primary modular unit 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, we'll see that having functions that are versatile and powerful constructs will give us a great deal of flexibility and sway when writing our code. We'll spend significant chunks of the book examining just how the nature of functions as first-class objects can be exploited to our great benefit. But first, let's specify exactly what we mean by stating that functions are "first-class objects".

4.1.1 Functions as first-class objects

In JavaScript, objects enjoy certain capabilities:

- They can be created via literals

```
var ninja = {}
```

- They can be assigned to variables, array entries, and properties of other objects

```
var ninja = {}; //#A
ninjaArray.push({}); //#B
ninja.data = {}; //#C
```

#A Assigns a new object to a variable
#B Adds a new object to an array
#C 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
```

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

- They can be returned as values from functions

```
function returnNewNinja() {
  return {}; //#A
   }
```

#A Returns a new object from a function

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

```
var ninja = {};
ninja.name = "Hanzo"; //#A
```

#A Creates a new property on an object

Functions in JavaScript possess all of these capabilities and are thus treated like any other object in the language. Therefore, 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:

```
var ninjaFunction = function() {}; //#A
ninjaArray.push(function(){}); //#B
ninja.data = function(){}; //#C
```

#A Assigns a new function to a variable
#B Adds a new function to an array
#C Assigns a new function as a property of another object

- Passed as arguments to functions:

```
function call(ninjaFunction){
  ninjaFunction();
}
call(function(){}); //#A
```

#A A newly created function passed as an argument to a function

- Returned as values from functions:

```
function returnNewNinjaFunction() {
  return function(){}; //#A
   }
```

#A Returns a new function

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

```
var ninjaFunction = function(){};
ninjaFunction.name = "Hanzo"; //#A
```

#A Adds a new property to a function

As you can see, whatever you can do with other objects, you can do with functions as well. Actually, in essence, functions *are* objects, just with an additional, special capability of being *invokable*. That is, functions can be called or invoked in order to perform some action.

Functional programming in JavaScript

Having functions as first-class objects is the first step towards *functional programming*, a style of programming that is focused on solving problems by composing functions (instead of specifying sequences of steps, as in more mainstream, imperative programming). It can help you write code that is easier to test, extend, and modularize. Functional programming is a pretty big topic, and in this book we'll only give it a nod (for example in Chapter 10). However, if you're interested about learning how to take advantage of functional programming concepts and apply them to your JavaScript programs, we recommend that you read "Functional Programming in JavaScript" by Luis Atencio, available at: www.manning.com/atencio/.

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 that very important concept.

4.1.2 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 what is termed a *callback*. The term stems from the fact that we establish a function that some other code will later "call back" into at an appropriate point of execution.

Callbacks are an essential part of using JavaScript effectively, and we're about to look at how we can use callbacks to handle events or to easily sort collections; real-world examples of how callbacks are used. But it's a tad complex, so before we dive into it, let's strip the callback concept completely naked and examine it in its simplest form.

We'll see callbacks used extensively as event handlers throughout the remainder of this book, but event handlers are just one example of callbacks; we can even employ callbacks ourselves in our own code. Here's 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(callback) {
  return callback();
}
```

As useless as this function is, it helps us demonstrate 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 our useless function with the code in the following listing:

Listing 4.1 A simple callback example

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

function useless(callback) {    //#A
  return callback();            //#A
}                               //#A

function getText() {            //#B
  return text;                  //#B
}                               //#B

assert(useless(getText) === text,                       //#C
       "The useless function works! " + text);  //#C
```

#A Defines a function that takes a callback function and immediately invokes it
#B Defines a simple function that returns a global variable
#C Call our `useless` function with the `getText` function as a callback

Here, we use the `assert()` testing function that we've set up in the previous chapter to verify that our `getText` callback function is indeed invoked and returns the expected value. In the following figure, you can see how our simple callback example is executed.

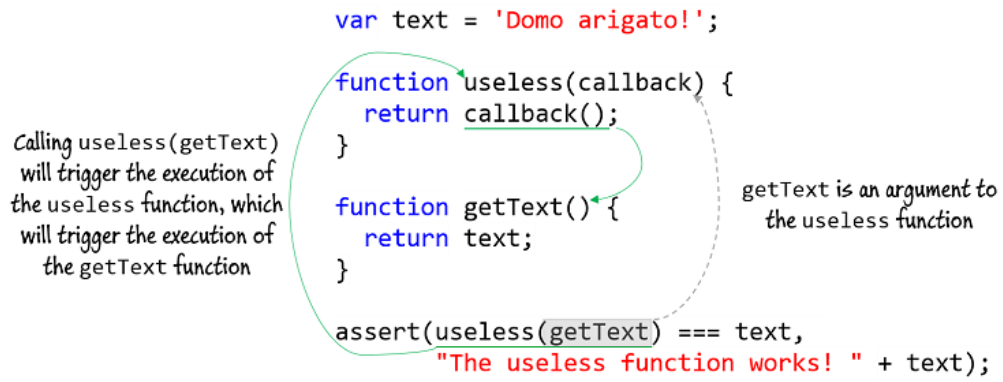


Figure 4.1 The flow of execution when making the useless(getText) call. The useless function is called with getText as an argument. In the body of the useless function there is a call to the passed in function, which in this case triggers the execution of the getText function.

That was really, really easy, and that's 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(callback) {
  return callback();
}

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

#A Define a callback function directly as an argument

Here, we have taken advantage of the fact that we can easily define a function directly as an argument to another function. Actually, in JavaScript, this is a much, much more common way of supplying callbacks. And it makes sense too. This is the only place in our code where we need that function, so why not define it immediately here. With this, we can see right away what our callback will do, instead of having to search around for function definition code.

This was an example of a callback where we have called our own callback. However, callbacks can also be called by the browser. Think back on chapter 2 where we had an example with the following snippet:

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

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

Now let's consider a not-so-simple example and compare it with using callbacks in a nonfunctional language.

SORTING WITH A COMPARATOR

Almost as soon as we *have* a collection of data, odds are we're going to need to sort it in some fashion. Let's say that we have an array of some numbers in a random order: 0, 3, 2, 5, 7, 4, 6, 1. That order might be just fine, but chances are that, sooner or later, we're going to want to rearrange them into some sorted order. Luckily, JavaScript provides a simple means to sort arrays into ascending order:

```
var values = [ 0, 3, 2, 5, 7, 4, 6, 1];
values.sort();
```

The approach is quite easy, you simply call the `sort` method, built into all arrays, and it will automatically sort the array for you.

But if we decide we want a sorting order *other* than ascending—something as simple as descending?

Usually, implementing sorting algorithms is not the most trivial of programming tasks: we have to select the best algorithm form the job at hand, implement it, adapt it to our current need (so that the items are sorted in some particular order), and be very, very careful not to introduce bugs. Out of these tasks, the only one that's application specific is the sorting order. So JavaScript makes our lives a lot easier, by requiring us to only define a comparison algorithm that tells the sort algorithm how the values should be ordered. This is where callbacks jump in! Instead of just 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:

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

See how easy was this? There's no need to think about the low level details of a sorting algorithm (our even which among the sorting algorithms to choose). We simply provide a callback that JavaScript engine will call every time it needs to compare two items.

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, so simply subtracting the values produces the desired return value to sort the array into descending order.

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.

One of the most important features of the JavaScript language is the ability to create functions anywhere in the code where 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.

But regardless of how functions are defined (much, much more on this in the upcoming section), they can be referenced as values and be used as the fundamental building blocks for reusable code libraries. Understanding how functions work at their most fundamental level will drastically improve our ability to write clear, concise, and reusable code.

Now let's take a more in-depth look at how functions are defined and invoked. On the surface it may seem that there's not much to the acts of defining and invoking functions, but there's actually a lot going on that we need to be aware of.

4.2 Defining functions

JavaScript functions are usually defined by using a *function literal* that creates a function value in the same way that 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.

In JavaScript, there are a couple of different ways of defining functions, which can be divided into four groups:

- *Function declarations* and *function expressions*: `function myFun(){}` – two most-common, yet subtly different ways of defining functions. Often people don't even consider them as separate, but as we'll see later on, being aware of their differences can help us evade some subtle bugs.
- *Arrow functions*: `myArg => myArg*2` – 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.
- *Generator functions*: `function* myGen(){}` – another ES6 addition to JavaScript. Generator functions enable us to create functions which, unlike normal functions, can be exited and re-entered later in the application execution, while keeping the values of their variables across these re-entrances. You can define generator versions of *function declarations*, *function expressions*, and *function constructors*. We'll look at this new feature in detail in Chapter 7.
- Function constructors: `new Function('a', 'b', 'return a + b')` – a not-so-often used way of defining functions that enables us to construct a new function from a string which can even be dynamically generated.

That leaves us with function constructors, a JavaScript feature that we'll skip entirely. While it has some interesting applications, especially when dynamically creating and evaluating code, we consider it a corner feature of the JavaScript language. If you are curious to know more about function constructors, you can visit the following page: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function.

> **NOTE** Be careful not to confuse *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* are functions that we use to create object instances. Don't worry, we'll place a special focus on constructor functions later in this chapter.

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

4.2.1 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 ways of defining functions are so similar that, a lot of times, we don't even make a distinction between them, but as we will soon see there are some subtle differences.

Let's start with function declarations.

FUNCTION DECLARATIONS

The most basic way of defining a function in JavaScript is by using function declarations (see figure 4.2).

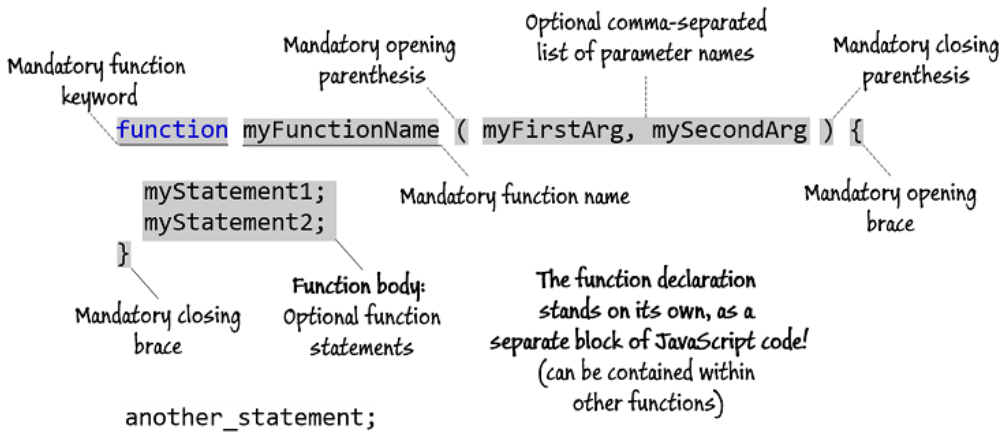


Figure 4.2 Elements of a function declaration

As can be seen in figure 4.2, 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 parenthesis. 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 that every function declaration must satisfy, there is one more condition: a function declaration must be placed on its own, as a separate JavaScript statement, but which can be contained within another function or a block of code; you'll see what we exactly mean by that in the next section. A couple of function declaration examples are shown in the following listing:

Listing 4.2 Examples of function declarations

```
function samurai() {                        #A
  return 'I'm a samurai! ';                 #A
}

function ninja() {                          #B

  function hiddenNinja() {                  #C
    return 'I'm a ninja!';                  #C
  }                                         #C

  return hiddenNinja();                     #B
}                                           #B
```

#A Defines function `samurai` in the global code
#B Defines function `ninja` in the global code
#C Defines function `hiddenNinja` within the `ninja` function

If you take a closer look at listing 4.2, you'll see something that you might not be accustomed to, if you haven't been exposed to functional languages that much: a function defined within another function!

```
function ninja() {
  function hiddenNinja() {
    return 'I'm a ninja!';
  }
  return hiddenNinja();
}
```

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

Having functions contained in other functions might even raise some tricky questions regarding scopes and identifiers resolutions, but save them for now, because we'll revisit this case in details in the next chapter.

Now that you've met function declarations, let's meet their siblings: function expressions.

FUNCTION EXPRESSIONS

As we have already mentioned multiple times, in JavaScript functions are first-class citizens, 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. Since functions are such fundamental constructs, the JavaScript language enables us to treat them as any other expressions. So just like 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.* Let's study listing 4.3 for the differences between function expressions and function declarations.

Listing 4.3 Function declarations and function expressions

```
function myFunctionDeclaration(){           //#A
  function innerFunction() {}               //#B
}

var myFunc = function(){};                  //#C
myFunc(function(){                          //#D
  return function(){};                      //#E
});

(function namedFunctionExpression () {      //#F
})();                                       //#F

+function(){}();                            //#G
-function(){}();                            //#G
!function(){}();                            //#G
~function(){}();                            //#G
```

#A A stand-alone function declaration
#B An inner function declaration
#C A function expression as a part of a variable declaration assignment
#D A function expression as an argument of a function call
#E A function expression as a function return value
#F A named function expression as a part of a function call that will be immediately invoked
#G Function expressions that will be immediately invoked, as an argument to a unary operator

Our example code, from listing 4.3, opens up vanilla style, with a standard function declaration which contains another inner function declaration:

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

We've put them in the example to emphasize how function declarations are separate statements of JavaScript code, but which can be contained within the body of other functions.

In contrast to them, we have function expressions, which are always a part of another statement, that is, they are placed on expression level: as right-hand side of a variable declaration (or an assignment): `var myFunc = function(){};` as an argument to another function call, or as a function return value:

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

Besides where in the code they are placed, there's one more difference between function declarations and function expressions: for function declarations, the function name is mandatory, while with function expressions it is completely optional.

Function declarations must have a name defined because they stand on their own, and one of the basic requirements for a function is that it has to be invokable. To do this, there has to be 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. This means that there are 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();
```

If it is an argument to another function, we can invoke it within that function though 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 on the position where you would normally expect a function identifier. Let's stop for a minute and take a closer look at that construct (see figure 4.3).

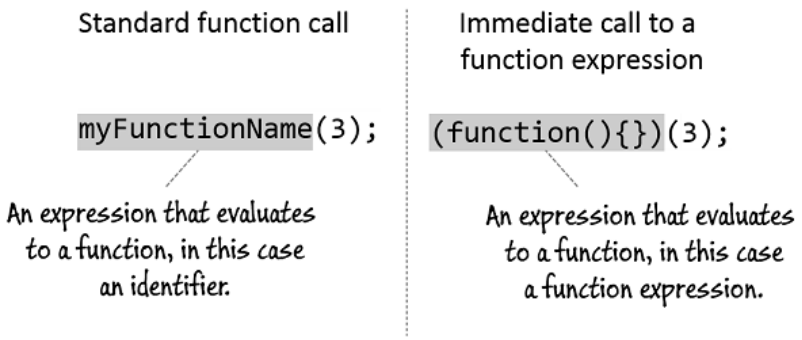


Figure 4.3 A comparison of a standard function call and an immediate call to a function expression.

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 some arguments. In the most basic function call, we simply put an identifier that evaluates to a function, as we did on the left side of figure 4.3. However, 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 just use a function expression. So on the right hand side of figure 4.3, 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, that we'll revisit later on, in chapter 11.

> **NOTE** There's one more thing that 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 engine has to be able to easily differentiate between function declarations and function expressions. If we leave out the parenthesis around the function expression, and put our immediate call as a separate statement: `function(){}(3);` the JavaScript engine will start processing it, and will conclude, because it is a separate statement starting with a the keyword function, that it is dealing with a function declaration. Since 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 engine that it is dealing with an expression (and not a statement).

The last four expressions in listing 4.1 are variations to the same theme of immediately invoked function expressions that you'll often find in JavaScript libraries:

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

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

Now that we've studied the ins and outs of 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.

4.2.2 Arrow functions

NOTE Arrow functions are a recent addition to the JavaScript standard (so recent, that they might not be supported by all browsers, see http://kangax.github.io/compat-table/es6/#arrow_functions).

Because in JavaScript we use *a lot* of functions, it made sense to add some syntactic sugar that enables us to create functions in a shorter, more succinct way, thus making our lives as developers a bit 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 value2 - value1;
});
```

In this example, we used a callback function expression sent to the sort method of the array object; a callback that will be invoked by the JavaScript engine to sort the values of our array in a descending order.

Now let's see how we can do the exact same thing by using arrow functions:

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

See how much more succinct this is? There is no clutter caused by the `function` keyword, the braces, nor the `return` statement. The arrow function simply states: here's a function that takes two arguments and returns their difference, in a much simpler way than a function expression could. Notice the introduction of a new operator `=>`, the so-called "fat arrow" operator (an equals sign immediately followed by a greater than sign), that is at the core of defining an arrow function.

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

```
param => expression
```

This arrow function simply takes a parameter and returns the value of some expression. We can use this syntax in the following way:

Listing 4.4 Comparing a lambda function and a function expression

```
var greet = name => "Greetings " + name;   //#A
assert(greet("Oishi") == "Greetings Oishi",
       "Oishi is properly greeted");

var anotherGreet = function(name){              //#B
  return "Greetings " + name;
}
```

```
assert(anotherGreet("Oishi") == "Greetings Oishi",
       "Once more, Oishi is properly greeted");
```

#A Defines a lambda function
#B Defines a function expression

Take a while and appreciate how arrow functions make our code more succinct, without sacrificing clarity.

That was the simplest version of the arrow function syntax, but in general, the arrow function can be defined in two different ways, as shown in figure 4.4.

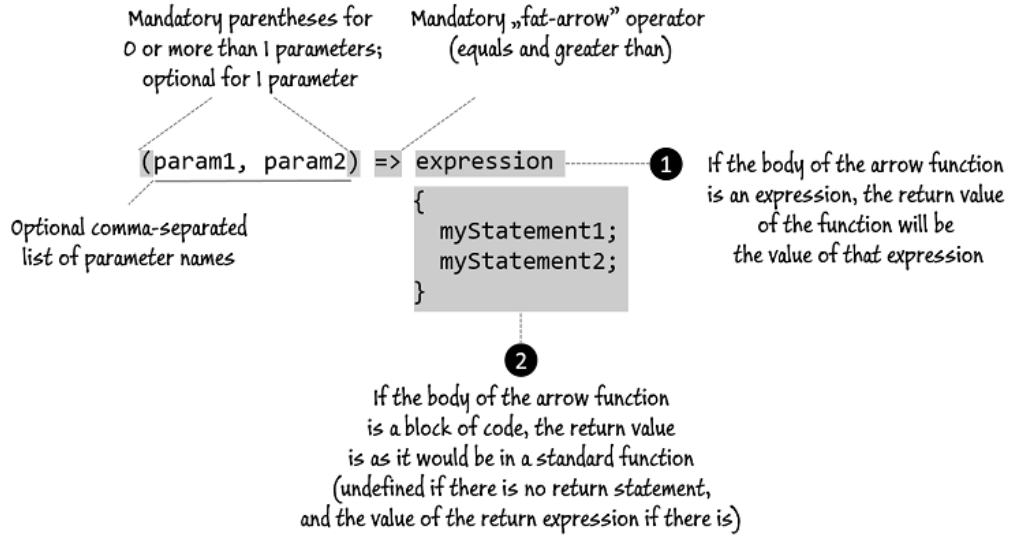


Figure 4.4 The syntax of an arrow function.

Figure 4.4 shows the syntax of an arrow function. The arrow function definition starts with an optional comma-separated list of parameter names. If there are no parameters, or if there is more than one parameter, this list must be enclosed within parentheses, but if there is only a single parameter, the parentheses are optional. This list of parameters is followed by a mandatory "fat-arrow" operator, which, in essence tells both programmers and the JavaScript engine that we are dealing with an arrow function.

After the fat arrow operator, we have two options. If it is a very simple function, we just 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 example, in our first arrow function example we had an arrow function:

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

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

In other cases, where our arrow functions are not that simple and when they require a bit more code, we can simply include a block of code after the arrow operator. For example:

```
var greet = name => {
  var helloString = 'Greetings ';
  return helloString + name;
};
assert('Greetings Oishi' === greet('Oishi'), 'Oishi is greeted');
```

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

This is it for now, but we'll revisit arrow functions multiple times throughout this book. Among other things, we'll show you some additional features of arrow functions that will help us evade some 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 study what happens with the values that we pass to a function.

4.3 From arguments to function parameters

Sometimes, when discussing code, 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 a bit more formal.

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

Figure 4.5 illustrates the difference.

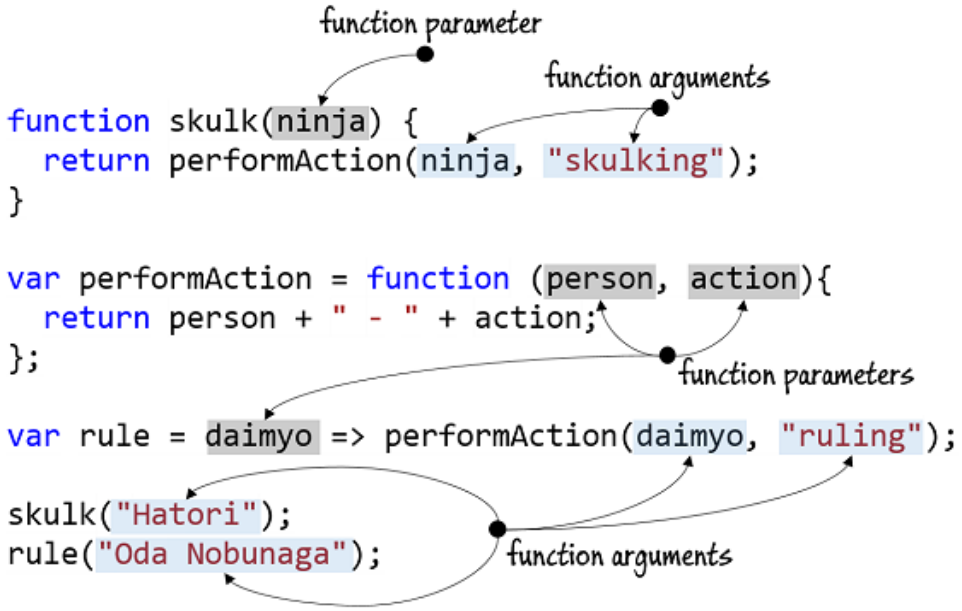


Figure 4.5 The difference between function parameters and function arguments.

As can be seen in figure 4.5, 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 are values passed to a function at the time of their invocation:

- The string "`Hatori`" 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 they are specified. The first argument gets assigned to the first parameter, the second argument to the second parameter, and so on.

If there are a different number of arguments than there are 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 are simply not assigned to parameter names. For example, see the following figure.

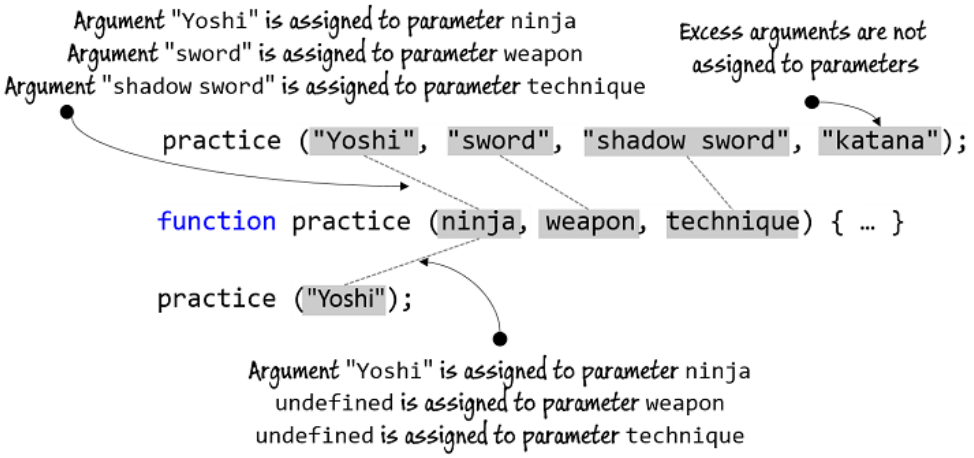


Figure 4.6 Arguments are assigned to function parameters in the order they are specified. Excess arguments are not assigned to any parameters.

Figure 4.6 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 parameters `ninja`, `weapon`, and `technique`, respectively.

The argument `"katana"`, is an excess argument, and would not be assigned to any parameter. We'll see in just a bit that even though some arguments aren't assigned to parameter names, we still have a way to access them.

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

In addition to these parameters, function invocations are, very interestingly, usually also passed two implicit parameters: `arguments` and `this`.

By *implicit*, we mean that these parameters aren't explicitly listed in the function signature, but they're silently passed to the function and are accessible within the function, and 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.

4.3.1 The arguments parameter

The `arguments` parameter is a collection of all of the arguments passed to the function. The collection has a property named `length` that contains the number of arguments, and 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 4.5 Using the arguments parameter

```
function whatever(a, b, c){                    //#A

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

  assert(arguments.length === 5,               //#C
    'We've passed in 5 parameters');           //#C

  assert(arguments[0] === a,                   //#D
    'The first argument is assigned to a');    //#D
  assert(arguments[1] === b,                   //#D
    'The second argument is assigned to b');   //#D
  assert(arguments[2] === c,                   //#D
    'The third argument is assigned to c');    //#D

  assert(arguments[3] === 4,                   //#E
    'We can access the fourth argument');      //#E
  assert(arguments[4] === 5,                   //#E
    'We can access the fifth argument');       //#E
}

whatever(1,2,3,4,5);                           //#F
```

#A Declare a function with three parameters: `a`, `b`, and `c`
#B Below we've called this function with arguments and here we test them for correct values
#C In all, the function was passed 5 arguments
#D Checks that the first three arguments match the function parameters
#E Checks that the excess arguments can be accessed through the `arguments` parameter
#F Call a function with 5 arguments

In listing 4.5, 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:

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

We can access the first three arguments through 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, which also includes the excess arguments that were not 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');
```

ARGUMENTS OBJECT AS AN ALIAS TO FUNCTION PARAMETERS

The `argument` parameter also has one curious feature. In essence, the `arguments` object aliases function parameters. This means that if we set a new value to, for example, `arguments[0]` the value of the first parameter will also be changed.

Listing 4.6 The arguments object aliases function parameters

```
function infiltrate(person) {
  assert(person === 'gardener',                    //#A
    'The person is a gardener');                   //#A
  assert(arguments[0] === 'gardener',              //#A
    'The first argument is a gardener');           //#A

  arguments[0] = 'ninja';                          //#B

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

  person = 'gardener';                             //#D

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

infiltrate("gardener");                            //#F
```

#A The `person` parameter has the value "gardener", sent as a first argument
#B Change the value of the first argument
#C Changing the arguments object will also change the matching parameter
#D Changes the value of the `person` parameter
#E The alias works both ways!
#F Calls the `infiltrate` function with the argument "gardener"

In listing 4.6, we demonstrate how the arguments object is just an alias for the function parameters. We've defined a function `infiltrate` that takes a single parameter `person`, and we've invoked it with the argument `"gardener"`. This means that we can access the value `"gardener"` through the function parameter `person`, or the `arguments` object:

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

As 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 thing holds for the other direction. If we change a parameter, the change can be observed in both the parameter and the `arguments` object.

AVOIDING ALIASES

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

Strict mode

Strict mode is an ES5 addition to JavaScript which 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 the strict mode changes is that there is no arguments aliasing in strict mode.

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

Listing 4.7 Using strict mode to avoid arguments aliasing

```
"use strict" #A

function infiltrate(person){
  assert(person === 'gardener',                    //#B
    'The person is a gardener');                   //#B
  assert(arguments[0] === 'gardener',              //#B
    'The first argument is a gardener');           //#B

  arguments[0] = 'ninja';                          //#C

  assert(arguments[0] === 'ninja',                 //#D
    'The first argument is now a ninja')           //#D

  assert(person === 'gardener',                    //#E
    'The person is still a gardener');             //#E
}

infiltrate("gardener");                            //#F
```

#A Enable strict mode
#B The person argument and the first argument start with the same value
#C Change the first argument
#D The first argument is changed
#E The value of the person parameter hasn't changed

We start listing 4.7 by placing a simple string `"use strict"` as the first line of our code. This will tell the JavaScript engine that we want to execute the following code in strict mode. In

this case, the 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');
```

However, unlike in non-strict mode, this time around the `arguments` object doesn't alias the parameters. If we changed 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');
```

Note that, throughout this section, we went 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, its entries can be fetched using array notation, and we can even iterate over it with a `for` loop. But it's *not* a JavaScript array, and if you try to use array methods on `arguments` (for example, the `sort` method, that we've used throughout this chapter), you'll find nothing but heartbreak and disappointment. Just think of `arguments` as an "array-like" construct, and exhibit restraint in its use.

The second implicit parameter, the `this` parameter, is even more interesting.

4.3.2 The this parameter: introducing the function context

When a function is invoked, in addition to the parameters that represent the explicit arguments that were provided on the function call, an implicit parameter named `this` is also passed to the function. The `this` parameter refers to an object that's implicitly associated with the function invocation and is termed the *function context*.

The function context is a notion that those coming from object-oriented languages such as Java will think that they understand; that `this` points to an instance of the class within which the method is defined. But beware! As we'll see, invocation as a *method* is only one of the four ways that a function can be invoked. And as it turns out, what the `this` parameter points to isn't, as in Java, defined only by how and where the function is defined, but it can also be heavily influenced by how it's *invoked*.

Yes, stop and ponder on that for a moment because your understanding of JavaScript functions greatly depends on understanding function contexts, and that what the function context consists of is *not* just a matter of how the function is declared.

Really. Now take a deep breath and we'll move on.

How value of the function context is determined does depend greatly, but not exclusively, on the way we've created our function. In the case of function declarations, function expressions, and function generators, the value of the `this` parameter depends *only* on the way the function was invoked, while in the case of arrow functions, the value of the *this* parameter depends on where the arrow function was defined (we'll see some examples in chapters 5 and 6).

We're about to look at how the four invocation mechanisms differ, and you'll see that one of the primary differences between them is how the value of `this` is determined for each type of invocation. 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; we'll be discussing `this` at great length.

Now let's see how functions can be invoked.

4.4 Invoking functions

We've all called JavaScript functions, but have you ever stopped to wonder what really happens when a function is called? In this section, we'll examine the various ways that functions can be invoked.

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 ninja level.

There are actually four different ways to invoke a function, each with its own nuances:

- As a function, in which the function is invoked in a straightforward manner
- As a method, which ties the invocation to an object, enabling object-oriented programming
- As a constructor, in which a new object is brought into being
- Via its `apply` or `call` methods, which is kind of complicated, so we'll cover that when we get to it

Listing 4.8 Function calling mechanism

```
skulk('Hatori');  #A

(function(who){ return who; })('Hatori');  #A

ninja.skulk('Hatori');    #B

var ninja = new Ninja('Hatori');    #C

skulk.call(ninja, 'Hatori');     #D

skulk.apply(ninja, ['Hatori']);     #E
```

#A Invoked "as a function"
#B Invoked as a method of `ninja`
#C Invoked as a constructor
#D Invoked via the `call` method
#E 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 with the simplest form, invoking functions as functions.

4.4.1 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 simply 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'll discuss that next.)

Here are some simple examples:

```
function ninja(){};
ninja();

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

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

null vs undefined

The difference between `null` and `undefined` is one of those small, but important conundrums in JavaScript, and we'll pay special attention to it in chapter 12. But for now, it's enough that we know that `undefined` is the value of missing members, or variables that haven't yet been assigned a value; while `null` is used to explicitly represent "empty" values.

Let's explore this difference in behavior between strict and non-strict mode, in the following listing.

Listing 4.9 Invocation as a function

```
function ninja() {
  return this;                          //#A
}

function samurai() {
  "use strict";                         //#B
  return this;
}

assert(ninja() === window,
  "In a 'non-strict' ninja function, " +
```

```
  "the context is the global window object");  //#C

assert(samurai() === undefined,
  "In a 'strict' samurai function, " +
  "the context is undefined");                 //#D
```

#A A function in non-strict mode
#B A function in strict mode
#C As expected, a non-strict function has `window` as the function context
#D The strict function, on the other hand, has an `undefined` context

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*.

4.4.2 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();
```

OK, so what? The function is called a "method" in this case, but what makes that interesting or useful?

Well, if you come from any 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 the function as the *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 be getting 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 4.10 Illustrating the differences between function and method invocations

```
function whatsMyContext() {                  //#A
  return this;                               //#A
}                                            //#A

assert(whatsMyContext() === window,          //#B
  "Function call on window");                //#B

var getMyThis = whatsMyContext;              //#C

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

var ninja1 = {                               //#E
  getMyThis: whatsMyContext                  //#E
};                                           //#E
```

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

var ninja2 = {                              //#G
  getMyThis: whatsMyContext                 //#G
};                                          //#G

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

#A A function that returns its function context. This will allow us to examine the function context of the function from outside of it, after it has been invoked.
#B When invoked "as a function" the function context is the window object.
#C The variable `getMyThis` gets a reference to the `whatsMyContext` function.
#D Invokes the function using the `getMyThis` variable. Even though we've used a variable, the function is still invoked "as a function" and the function context is the `window` object.
#E A `ninja1` object is created with a `getMyThis` property that reference the `whatsMyContext` function.
#F Invoking the functions through `getMyThis` calls it as a method of `ninja1`. The function context is now `ninja1`. That's object orientation!
#G Another object, `ninja2` also has a `getMyThis` property referencing `whatsMyContext` .
#H Invoking the function as a method of `ninja2`, shows that the function context is now `ninja2`!

In this test, we set 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'd expect that the function context would be the global context; in other words, `window`. We assert that this is so:

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

Then we create a reference to the function `whatsMyContext` in a variable named `getMyThis`: `var getMyThis = whatsMyContext`. Note that 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'd once again be invoking the function as a function. As such, we'd once again expect that the function context would be the `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 `skulk` that receives a reference to the `creep` function. By doing so, we say that we've created a *method* named `skulk` on the object. We don't say that `creep` has *become* a method of `ninja1`; it hasn't. We've already seen that `creep` 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 (the same thing holds for both the non-strict and the strict method):

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

> **NOTE** Invoking functions as methods is crucial to writing JavaScript in an object-oriented manner. It means that we can 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 our testing by creating yet another object, `ninja2`, also with a property named `skulk` that references the `creep` function. Upon invoking these methods through the `ninja2` object, we correctly assert that their function context is `ninja2`:

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

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

Note that even though the *same* function – `whatsMyContext` - is used throughout the example, the function context returned by `this` changes depending upon 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 upon, the object through which the method was invoked. This means that 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 we used it in this example has limitations. Foremost, when we created the two ninja objects, we were able to share the same function to be used as a method in each, but we had 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 be exploring those capabilities in depth in chapter 8. But for now, let's consider a part of that mechanism that relates to function invocations: the *constructor*.

4.4.3 Invoking functions as constructors

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. The difference is in how the function is invoked.

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

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

If we want to invoke the `creep` function as a constructor, we'd write this:

```
new creep();
```

But even though we can invoke `creep` as a constructor, that function isn't a particularly useful constructor, because it simply returns the newly created object. Let's find out why by discussing what makes constructors special.

THE SUPERPOWERS OF CONSTRUCTORS

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

Listing 4.11 Using a constructor to set up common objects

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

var ninja1 = new Ninja();                             //#B
var ninja2 = new Ninja();                             //#B

assert(ninja1.skulk() === ninja1,                     //#C
  "The 1st ninja is skulking");

assert(ninja2.skulk() === ninja2,
  "The 2nd ninja is skulking");                       //#C
```

#A Defines 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.
#B Creates two objects by invoking the constructor with `new`. The newly created objects are referenced by `ninja1` and `ninja2`.
#C 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 `new` keyword, an empty object instance will be created and passed to the function as `this`. 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 the following figure.

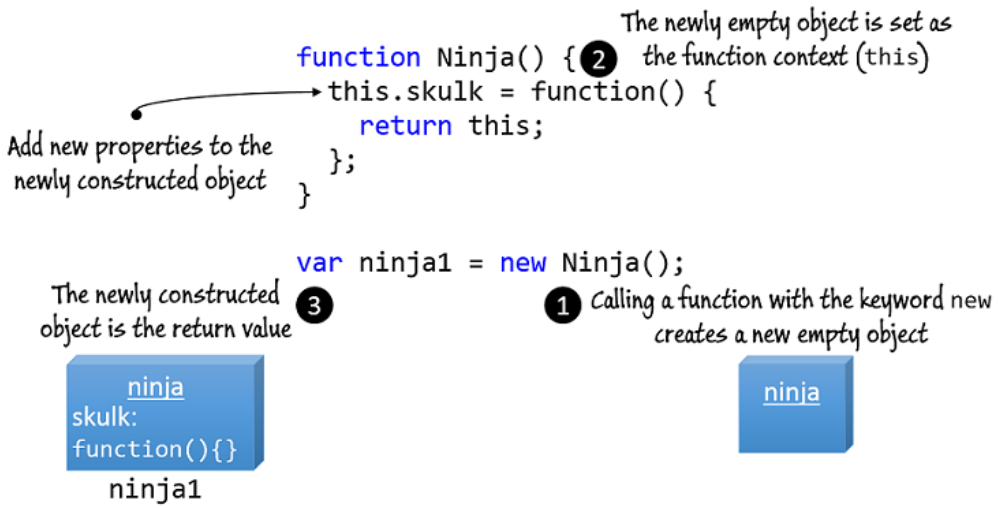


Figure 4.7 When calling a function with a keyword new a new empty object is created and set as the context of the constructor function.

As figure 4.6 shows, 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).

This last two points touch upon why `creep` in `new creep()` 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 functions intended for use as constructors.

Let's consider a more appropriate constructor function `Ninja` in the following listing, a constructor function that will set up the skulking ninjas of listing 4.8 in a more succinct fashion.

As we can see in listing 4.11, the `Ninja` constructor assigns a new method called `skulk` to the newly created objects:

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

The `skulk` method performs the same operation as `creep` 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 `Ninja`s:

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

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

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

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 a bit with the following listing.

Listing 4.12 Constructors returning primitive values

```
function Ninja() {                                          //#A
  this.skulk = function () {
    return true;
  };
  return 1;                                                 //#B
}
assert(Ninja() === 1,                                       //#C
  "Return value honored when not called as a constructor"); //#C

var ninja = new Ninja();                                    //#D

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

#A Defines a constructor function named Ninja.
#B The constructor returns a specific non-object value: the number 1.
#C The function is called "as a function" and its return value is returned as expected.
#D The function is called as a constructor via the `new` operator.
#E Tests verify that the return value of 1 is ignored, and that a new, initialized object has been returned from `new`.

If you run this listing you'll see that all is fine and well. The fact that our `Ninja` function return a simple number: `1`, has no significant influence on how our code behaves. If we call the `Ninja` function "as a function", it simply returns `1` (just as we would 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 4.13 Constructors explicitly returning object values

```
var puppet = {                                  //#A
  rules: false                                  //#A
};                                              //#A

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

var emperor = new Emperor();                    //#C

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

#A Create our own object with a known property.
#B Returns that object despite initializing the object passed as `this`.
#C Invoke the function as a constructor.
#D Tests show that the object returned by the constructor is assigned to the variable emperor (and not the object created by the `new` expression).

In listing 4.13 we take a slightly different approach. We start by creating a `puppet` object with a 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 `puppet` object:

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

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

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

We've set up an ambiguous situation: we get one object passed to us 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 tell us that the `puppet` object is returned as the value of `new`, and that the initialization that we performed on the function context in the constructor were all for naught.

Finally, the `puppet` has been exposed!

To 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 simply discarded. If however, a non-object is returned from the constructor, the returned value is simply ignored, and the newly-created object is returned, as expected.

Due to all these peculiarities, functions intended for use as constructors are generally coded differently from other functions. Let's explore that in greater depth.

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 while 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 very 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 effect would be for the `skulk` property to be created on `window` in non-strict mode, and for `window` to be returned and stored in `whatever`; not a particularly useful operation.

Thing go even more awry in strict more, as `this` would be undefined and our JavaScript application would crash. This is actually a good thing; if we make this mistake in non-strict mode, it might escape notice it (unless we had good tests), but there's no missing the mistake in strict mode. This is actually 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 pretty easy to see how a constructor makes it much easier to create multiple objects that conform to the same pattern without having to repeat the same code over and over again. The common code is written just once, as the body of the constructor. In chapter 8, we'll see much more about using constructors and about the other object-oriented mechanisms that JavaScript provides that make it even easier to set up object patterns.

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

4.4.4 Invocation with the apply and call methods

So far, we'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 is 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 "strictness"); for constructors, it's a newly created object instance.

But what if we wanted to make it whatever we wanted? What if we wanted 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 a bit ahead and consider that when an event handler is called, the function context is set to the bound object of the event. We'll see a concrete example in the next chapter, and we'll also examine event handling in detail in chapter 14, but for now just assume that the bound object is the object upon which the event handler is established. That's usually exactly what we want, but not always. For example, in the case of a method, we might want to force the function context to be the owning object of the method and not the object to which the event is bound. We'll see this scenario in chapter 14, but for now the question is, can we do that?

Well, yes we can.

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 `Function` constructor), functions can have properties just like any other object type, including methods.

To invoke a function 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 4.14 Using the `apply` and `call` methods to supply the function context

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

var ninja1 = {};                                        //#B
var ninja2 = {};                                        //#B

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

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

#A The function "juggles" the arguments and puts the result onto whatever object is the function context.
#B These object are initially empty and will serve as our test subjects.
#C Uses the `apply` method passing `ninja1` and an array of arguments.
#D Uses the `call` method passing `ninja2` and a list of arguments.
#E 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 number of other arguments:

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

Then we test!

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 in the passed array. Then we do the same for `ninja2`, which was called via `call`:

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

A closer look at what's going on listing 4.11 is shown in the following figure.

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

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

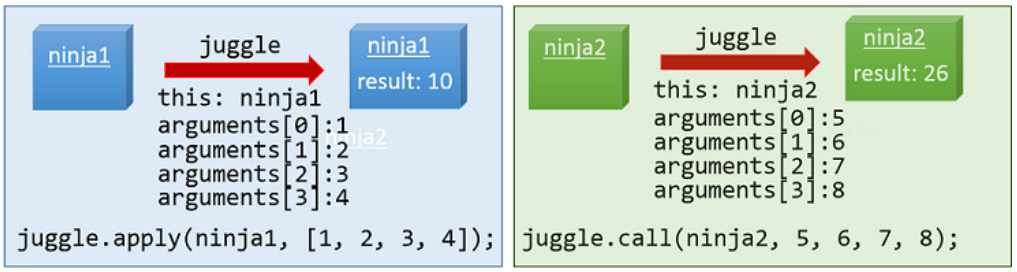


Figure 4.8 Manually set a function context by using built-in call and apply

We can also deconstruct the syntax of call and apply methods, in the following figure.

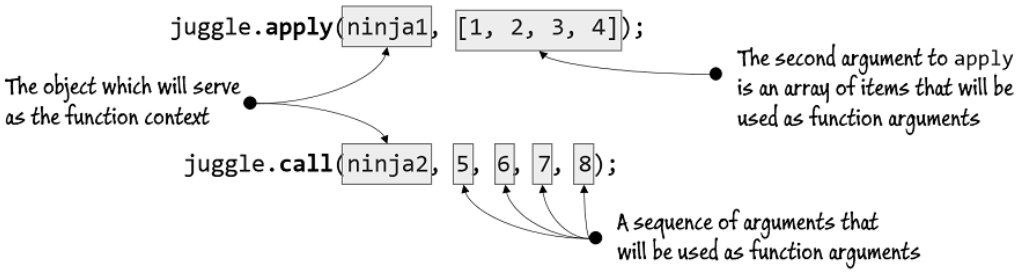


Figure 4.9 As first argument, both the call and apply methods take the object which will be used as function context. The difference is in the following arguments. Apply takes only one additional argument, an array of argument values; while call takes any number of arguments, which will be used as function arguments.

These methods can come in handy whenever it would be 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. Let's take a simple function that will perform an operation on every entry of an array.

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 would be 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 building blocks of the program rather than imperative statements.

You might think that it's all rather moot, and that all we're doing is moving the `for` loop out one level, but we're not done massaging this example yet.

In order 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*) once we've covered closures in chapters 5 and 6. Such an iteration function *could* simply pass the "current" element to the callback as a parameter, but most make the current element the function context of the callback.

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

Listing 4.15 Building a for-each function to demonstrate setting a function context

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

var weapons = ['shuriken','katana','nunchucks'];   //#C

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

#A Our iteration function accepts the collection to be iterated over, and a callback function.
#B The callback is invoked such that the current iteration item is the function context.
#C Our test subject!
#D Calls the iteration function and ensures that the function context is correct for each invocation of the callback.

Our 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 array `weapons` array and then 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]);
});
```

The following figure shows that our function works splendidly.

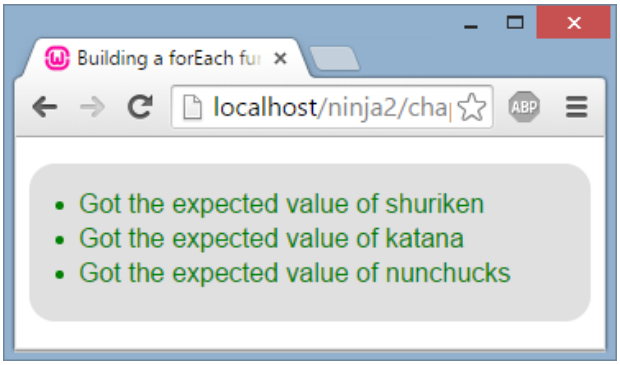


Figure 4.10 The test results show that we have the ability to make any object we please the function context of a callback invocation.

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 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 also pass an arbitrary number of arguments to the callback in addition to the iteration index.

Something you may have asked yourself at this point is that given that `apply` and `call` do pretty much the same thing, how do we decide which to use?

The high-level answer is the same answer as for many such questions: we'd use whichever one improves code clarity. A more practical answer would be 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.

4.5 Summary

In this chapter, we learned:

- 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. Just like 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
- Each object has a "super power" that distinguishes it from the rest; for functions it's the ability to be invoked.
- There are different types of functions: function declarations, function expressions, arrow functions, and function generators.
- The parameter list of a function and its actual argument list can be of different lengths:
  - Unassigned parameters evaluate as `undefined`.
  - Extra arguments are simply not bound to parameter names.
- A function invocation is passed two implicit parameters:
  - `arguments`, a collection of the actual passed arguments
  - `this`, a reference to the object serving as the function context
- Functions can be invoked in various ways, and the invocation mechanism determines the function context value:
  - When invoked as a simple function, the context is the global object (`window`) in non-strict mode, and `undefined` in strict mode.
  - When invoked as a method, the context is the object owning the method.

- When invoked as a constructor, the context is a newly allocated object.
- When invoked via the `apply` or `call` methods of the function, the context can be whatever the heck we want.

In all, we made a thorough examination of the fundamentals of function mechanics, and we'll continue this examination in the next chapter by investigating scopes – the mechanism of keeping track of identifiers and their values.

5

Identifier resolution and closures

This chapter covers:

- How the execution of JavaScript programs is tracked with execution contexts
- How identifier mappings are handled with lexical environments
- What are closures
- How closures work

In the previous chapter, we've seen how JavaScript is a programming language with significant functionally oriented characteristics. However, our JavaScript functions wouldn't be of much use without a way to store information through variables. So in this chapter, we'll make a through exploration of how JavaScript tracks variables throughout our program.

We'll start by seeing how the JavaScript engine keeps track of the execution of our programs, and then we'll study how it tracks the identifiers that are available at different points of application execution. We'll continue by exploring different types of variables in JavaScript, with a special focus on new additions to the JavaScript standard: blocked-scoped variables and constants. We'll also go through some of the common gotchas related to identifier resolution.

Finally, all concepts that we'll explore by learning about tracking application execution and identifiers, will come together in *closures, a fundamental* JavaScript concept, whose deep understanding will change the way your write your JavaScript code. Traditionally, closures have been a feature of purely functional programming languages. Having them cross over into mainstream development has been particularly encouraging, and it's not uncommon to find

closures permeating JavaScript libraries, along with other advanced code bases, due to their ability to drastically simplify complex operations.

Without further ado, let's start exploring!

5.1 Code execution

In JavaScript, the fundamental unit of execution is a function. You use them all the time, to calculate something, perform side-effects such as changing the UI, to achieve code reuse, or simply to make your code a bit easier to understand. In order to fulfill its purpose, a function can call another function, which can in turn call another function, and so on. And when a function does its thing, it has to return back to the position from which it was called. But have you ever wondered how the JavaScript engine keeps track of all these executing functions and return positions?

As we've already mentioned in chapter 2, there are two main types of JavaScript code: *global code*, placed outside of all functions, and *function code*, contained in functions. When you code is being executed by the JavaScript engine, each statement is executed in a certain *execution context. A*nd just as we have two different types of code, so do we have two different types of execution contexts: a *global execution context* and a *function execution context*. A significant difference between them is that there is only *one* global execution context, created when our JavaScript program starts, while a *new* function execution context is created on *each* function invocation. For example, consider the following figure in which we log the activity of a couple of sneaky ninjas.

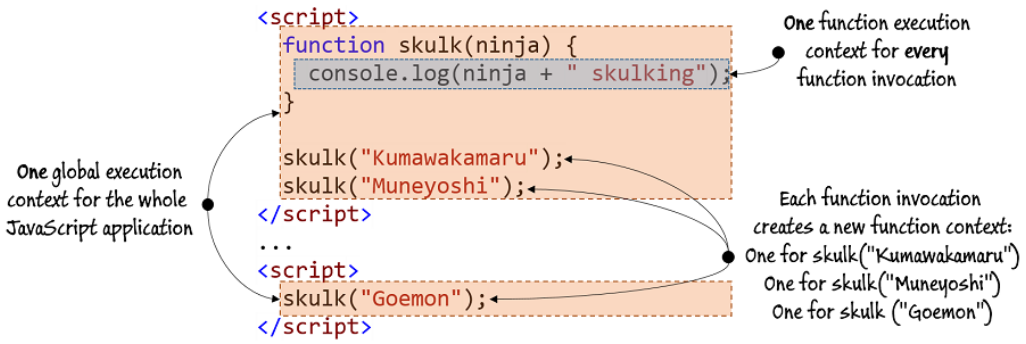


Figure 5.1 One global execution context is created per application, while one function execution context is created per function invocation.

Figure 5.1 shows an example application. When the JavaScript engine starts executing your JavaScript code, it creates *a* global execution context in which *all* your global JavaScript code will be executed in, even if it's spread across different script elements. In our example, even though there are two script elements, their global code is executed in the *same* global execution context, which is created before any JavaScript code is executed.

On the other hand, *every* time your code invokes a function, a new function execution context is created. In this case, this means that there is one execution context created when executing the `skulk` function with the argument `"Kumawakamaru"`, one when executing it with the argument `"Muneyoshi"`, and one when executing it with the argument `"Goemon"` – three different function execution contexts, one for each function invocation.

The mechanism of execution contexts is used by the JavaScript engine for two reasons: it allows us to keep track of the position in the application execution where the program flow is currently at, and it helps us deal with identifier resolution. Let's start by exploring how execution contexts are used to keep track of the application execution.

5.1.1 Tracking application execution with execution contexts

As you already know, JavaScript is based on a single threaded execution model, which means that at a certain point in time, only one piece of code can be executing. You also know that when your JavaScript application starts, a global execution context is created in which all your global JavaScript code is evaluated.

Now, let's continue. During the course of program execution, we call *a lot* of functions. And each time a function is invoked, since only once piece of code can be executed at a time, we have to stop the current execution context, and create a new function execution context in which the function code will be evaluated. Once the function performs its task, its function execution context is discarded, and the caller execution context restored. So we have this need to keep track of all these execution context, both the one that is currently executing, as well as the ones that are patiently waiting. The easiest way to do this is by using a *stack*, the so called *Execution Context stack.*

> **NOTE** A stack is one of the fundamental data structures in which you can put new items only to the top, and when you want to take existing items you also have to do it from the top. Think of a stack of trays in a cafeteria: when you want to take one for yourself, you simply pick the one from the top, and when the cafeteria staff have a new clean one, they simply put it on the top.

Let's take a look at the following code, where we report the activity of two skulking ninjas.

Listing 5.1 The creation of execution contexts

```
<script>
  function skulk(ninja) {                #A
    report(ninja + " skulking");
  }

  function report(message) {             #B
    console.log(message);
  }

  skulk("Kuma");                         #C
  skulk("Yoshi");                        #C
</script>
```

#A – a function that calls another function
#B – a function that reports a message
#C – two function calls from global code

The code in listing 5.1 is straightforward, we define a `skulk` function #A that calls the `report` function #B, which outputs a message. Then, from global code, we make two separate calls to the `skulk` function #C. Now, by using this code as a basis, we'll explore the creation of execution contexts, see the following figure.

```
<script>
  function skulk(ninja) {
    report(ninja + " skulking");
  }

  function report(message) {
    console.log(message);
  }

  skulk("Kuma");
  skulk("Yoshi");
</script>
```

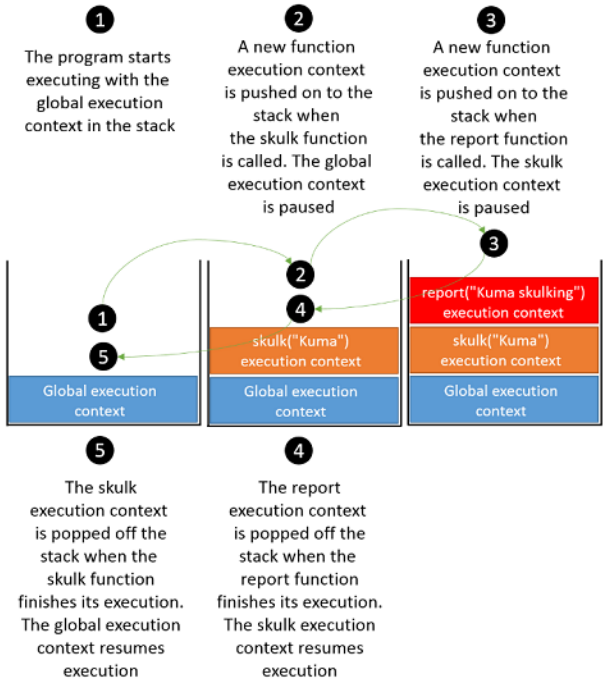


Figure 5.2 The behavior of the execution context stack

As figure 5.2 shows, in the beginning, the execution context stack starts with the global execution context #1, which is the active execution context when executing global code. In the global code, the program first defines two functions: `skulk` #A and `report` `#B`, and then it calls the `skulk` function #C. Since 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 execution context stack #2. The `skulk` function, in turn, calls the `report` function with the argument `"Kuma skulking"`. Again, since only one piece of code can be executed at once, 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 on to the stack #3. Once the `report` function logs the message and finishes its execution, we have to go back to the `skulk` function. This is done by simply popping the `report` function execution context from the execution context stack #4. When this has been done, the `skulk` function execution context is reactivated, and the execution of the `skulk` function continues. A similar thing happens when the `skulk` function finishes its execution – the function execution context of the `skulk` function is removed from the execution context stack, and the global execution context, that has been patiently waiting this whole time, is restored as the active execution context #5. With this, 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 respected functions are called. These execution contexts are also popped off the stack, when the program returns from the matching function. Figure 5.3 shows the state of the execution context stack when logging: `"Yoshi skulking"`.

Figure 5.3 The state of the execution context stack when logging the message "Yoshi skulking"

Even though the execution context stack is an internal JavaScript concept, you can explore it in any JavaScript debugger (see figure 5.4 for how it is accessible in Chrome's dev tools).

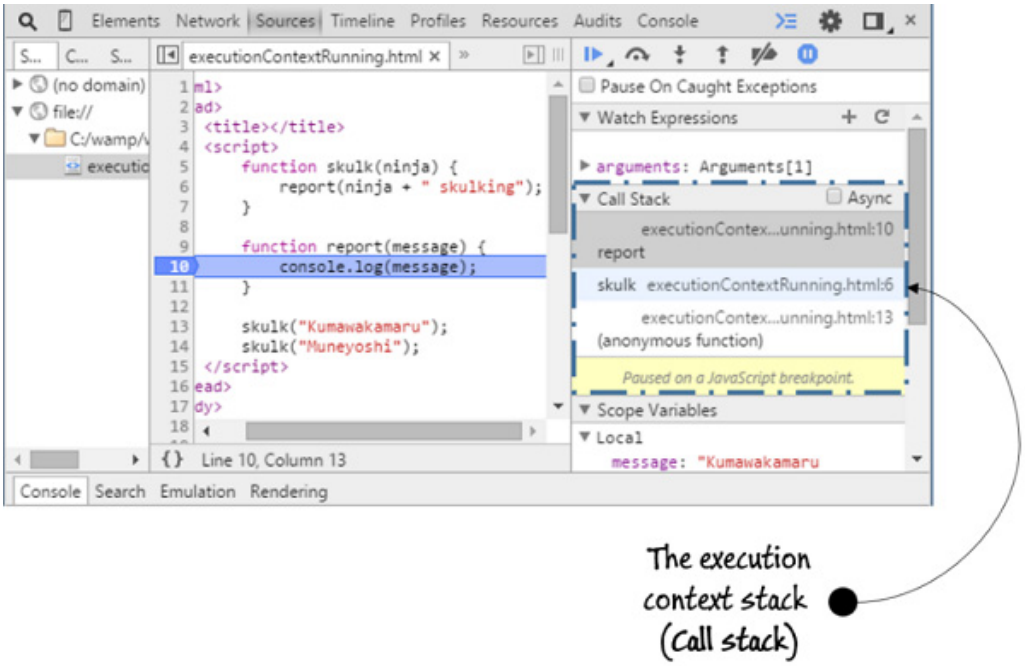


Figure 5.4 The current state of the execution context stack in Chrome Dev tools.

Besides keeping track of the position in the application execution, the execution contexts are vital for identifier resolution. The execution context does this via another internal JavaScript engine concept called the *Lexical Environment.*

5.2 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 values. For example, in the following code:

```
var ninja = "Hatori";
console.log(ninja);
```

The lexical environment is consulted when our code is accessing the `ninja` variable in the `console.log` statement.

> **NOTE** You're probably familiar with the concept of scopes(just in case, a scope refers to the visibility of identifiers in certain parts of a program). You can think of lexical environments as an internal implementation of the JavaScript scoping mechanism.

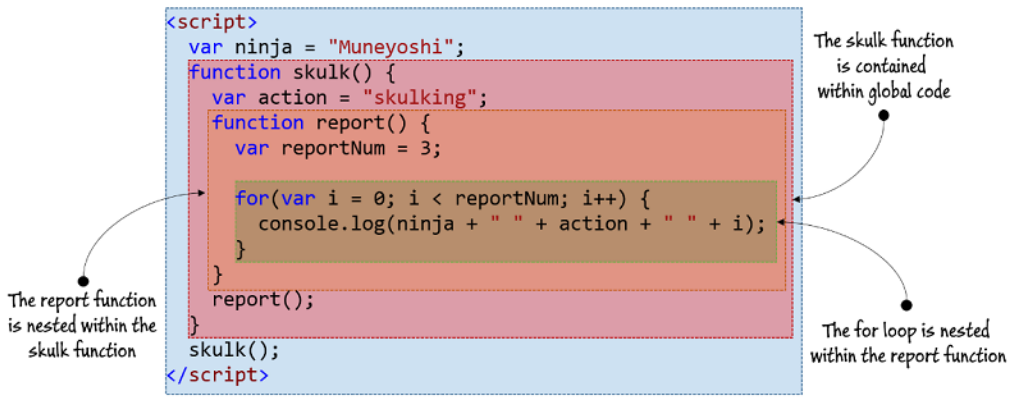
Usually, a lexical environment is associated with a specific structure of JavaScript code. It can be associated with a function, a block of code, or the catch part of a try-catch statement.

This means that each of these structures: functions, blocks, and catch parts can have their own separate identifier bindings.

> **NOTE** In earlier versions of JavaScript (pre-ES6), a lexical environment could only be associated with a function. This meant that variables could only be function scoped. This was a source of many confusions. Since JavaScript is a C-like language, people coming from other C-like languages (such as C++, C#, or Java) are naturally expecting that some low level concepts, such as the existence of block scopes, will be the same. With ES6 this is finally fixed.

5.2.1 Code nesting

Lexical environments are heavily based on the idea of code nesting, that one code structure can be contained within another code structure. For example, let's take a look at figure 5.4 which shows different 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>
```

Figure 5.5 Different types of code nesting

In figure 5.4, the `for` loop is nested within the `report` function, the `report` function within the `skulk` function, and the `skulk` function within the global code. In terms of scopes, each of these code structures gets an associated lexical environment *every* time such code is evaluated. For example, on every invocation of the `skulk` function, a new function lexical environment is created, and on each `for` loop iteration, a new block lexical environment is created. If you're wondering, in this case, there will be three different block lexical environments created while looping to the `reportNum` (don't worry, we'll see a more detailed example later on).

Now let's go through the implications of code nesting for identifier resolution. Consider a similar example in the following listing.

Listing 5.2 Nested environments

```
<script>
  var ninja = "Muneyoshi";

  function skulk() {
    var action = "skulking";

    function report() { #A
     assert(ninja == "Muneyoshi", "Muneyoshi, we can see you!"); #B
     assert(action == "skulking", "We know that you're skulking!");#C
    }

    report();
  }

   skulk();
</script>
```

#A – The report function is defined within the skulk function
#B – Check if we can see the global ninja variable
#C – Check if we can see the outer action variable, defined in the skulk function

Listing 5.2 shows an example in which a function `report` is defined within a function `skulk` #A. The interesting thing happens in the body of the `report` function. In the first assert statement #B we access the global `ninja` variable. This isn't anything weird, you can do the exact same thing in almost any other programming language. But now comes the interesting part. Consider the second test, where we access the variable `action` #C. If you look closely, you'll notice that the `action` variable is neither the variable of the `report` function (there is no variable declaration nor a function parameter with that name within the `report` function), nor a global variable. It is in fact, a variable of the `skulk` function. In JavaScript, variables that are accessed in the body of a function but that are neither local variables, nor function parameters are called *free* variables.

This code works perfectly fine, see figure 5.5. The `report` function has access to both `ninja` and `action` variables.

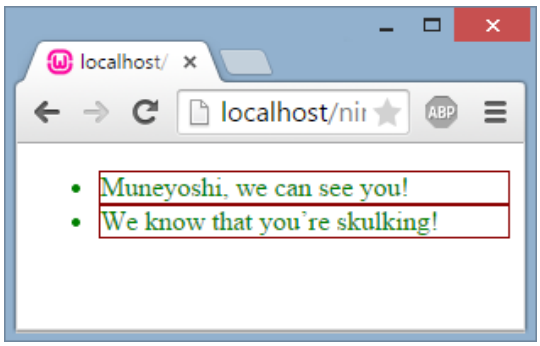


Figure 5.6 In an inner function, we can access variables from outer environments.

In terms of nesting, the code of the `skulk` function is nested within the global code, while the code of the `report` function is nested within the code of the `skulk` function. Since all of our code is nested within global code, from that perspective, it makes perfect sense to allow access to global variables from any part of code – that makes them global. But consider the case of the relationship: the `report` function contained within the `skulk` function. Just as we can access global variables from any part of the code, so it makes perfect sense to be able to access variables from outer functions within inner functions, regardless of the depth of nesting, since these variables are in effect "global" from the perspective of inner functions. There's nothing special about it, we've all probably written code such as this.

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.

CODE NESTING AND LEXICAL ENVIRONMENTS

In addition to keeping track of local variables, function declarations, and function parameters, since in inner functions we can access the variables of outer functions, the lexical environment also has to keep track of its *outer* (parent) lexical environment. The idea being that if the identifier cannot be found in the current lexical environment that the outer lexical environment is searched for. This stops either when the matching variable is found in this chain of lexical environments, or with a reference error if we've reached the global lexical environment and the identifier hasn't been found. Let's see an example in the following figure.

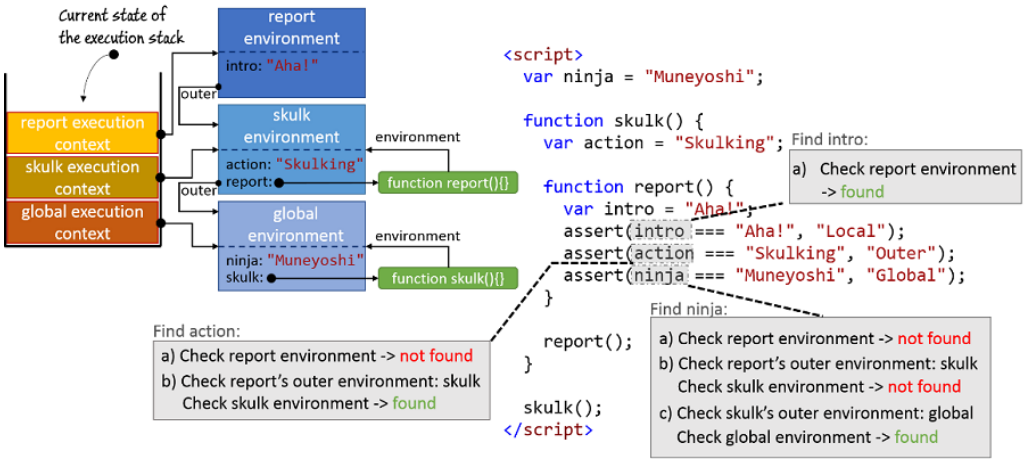


Figure 5.7 Resolving identifiers when executing the report function

We'll examine how identifiers `intro`, `action`, and `ninja` are resolved when executing the `report` function. The `report` function is called by the `skulk` function, which is in turn called by global code. Each of those execution contexts 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 environment associated with the execution of the `skulk` function, mapping for the identifiers `action` and `report`, and the environment associated with the execution of the `report` function, mapping for the `intro` identifier (the left-hand side of figure 5.6).

In a particular execution context, besides accessing identifiers defined directly in the matching lexical environment, our programs often access other variables defined in outer lexical 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 cannot access or manipulate it directly) property named `[[Environment]]` (this is the notation that we'll use to mark these internal properties). In our case, the `skulk` function will keep a reference to the global environment, and the `report` function to the `skulk` environment.

> **NOTE** This might seem a bit odd at first – why don't we just traverse the whole stack of execution contexts and search their matching environments for identifier mappings? Technically, this would work in our current example, but remember, in JavaScript, functions can be passed around as any other objects, which means that the position of function definition and the position from where the function is executed, are generally not related. We'll look into this case with all the benefits and shortcomings later in this chapter.

Whenever a function is called, a new function execution context is created and pushed onto the execution context stack. With it a new associated lexical environment is created. Now comes the crucial part, as 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` lexical environment becomes the global environment, since this was the environment in which the `skulk` function was created in. Similarly, when calling the `report` function, the outer environment of the newly created `report` lexical environment is set to the `skulk` environment.

Now take a look at the `report` function. When the first `assert` statement is being evaluated, we have to resolve the identifier `intro`. To do this, the JavaScript engine starts by checking the environment of the currently running execution context – the `report` environment. Since the `report` environment contains the `intro` binding, the identifier is resolved. Next, the second `assert` statement has to resolve the binding for the `action` identifier. Again, the environment of the currently running execution context is checked. However, this time around, the `report` environment does not contain the binding for the `action` identifier, so the JavaScript engine checks the outer environment of the `report`

environment – the `skulk` environment. The binding exists and the identifier is resolved. A similar process is followed when trying to resolve the `ninja` identifier (a little hint, the binding can be found in the global environment).

Now that you understand the fundamentals of identifier resolution, let's study different ways a variable can be declared.

5.3 Understanding different types of JavaScript variables

In JavaScript, we can use three different keywords for defining variables: `var`, `let`, and `const`. They differ in two aspects: *mutability* and their relationship towards the lexical environment. In this section, we are going to show you how to create immutable variables by using the `const` keyword, and how to create block-scoped variables (a new addition to JavaScript). We'll also deeply explore how identifiers are registered and we'll study some of the common gotchas of identifier resolution.

We'll start with variable mutability.

5.3.1 Variable mutability

When dividing variable declaration keywords by mutability, we can put `const` on one side and `var` and `let` on the other side. All variables defined with `const` are immutable – their value can be set only once, and if you try to set it another time, the JavaScript engine will either silently ignore it, if the current code is in non-strict mode, or it will throw a type error if it is in strict mode. On the other hand, variables defined with keywords `var` and `let` are you typical run-of-the-mill variables, whose value you can change as many times as you like. We'll go a bit deeper into how `const` variables behave.

CONST VARIABLES

A const variable is very similar to a normal variable, with the exception that you have to provide an initialization value once you declare it, and that you can't change its value afterwards.

Const variables are most often used to reference some fixed value, for example, the maximum number of ronin in a squad: `MAX_RONIN_COUNT`, by name, instead of using a number, for example 234. This makes your programs easier to understand and change. Your code is not filled with seemingly arbitrary literals (`234`), but with meaningful variables (`MAX_RONIN_COUNT`) whose value you have to change in only one place, if necessary. At the same time, since these values are not meant to be changed during program execution, you've safeguarded your code against unwanted or accidental modifications and you've even made it possible for the JavaScript engine to do some performance optimizations.

Now let's study the behavior of `const` variables in strict and non-strict mode.

Listing 5.3 The behavior of constant variables

```
<script>
  const firstConst = 3; #A
  assert(firstConst == 3, "firstConst is 3");    #A
```

```
  firstConst = 4; #B
  assert(firstConst == 3, "firstConst is still 3!"); #B

  function myStrictFun {
    "use strict"; #C
    const secondConst = 3;  #D
    assert(secondConst == 3, "secondConst is 3");        #D

    try {
      secondConst = 4;                                   #E
      assert(false, "This shouldn't be reached!");       #F
    }
    catch(e){
      assert(true, "An exception was thrown"); #F
    }
    assert(secondConst == 3, "The value hasn't changed!"); #G
  }

  myStrictFun();

</script>
```

#A – define a const variable in non-strict mode and verify that the value was assigned
#B – assign a new value to a const variable and verify that the assignment was silently ignored
#C – the function code is in strict mode
#D – define a const variable in strict mode and verify that the value was assigned
#E – try to modify the const variable
#F – check that the exception was raised
#G – and that the value of the variable hadn't changed

In listing 5.3, we define and initialize two `const` variables: `firstConst` #A, in non-strict mode and `secondConst` #D, in strict mode. We first check that our `firstConst` variable has the assigned value #A and then we modify it #B. Since our code is in non-strict mode, this assignment is simply silently ignored #B. On the other hand, the `myStrictFun` function is executed in strict mode #C, so any attempt of assigning a value to a const variable will result with an exception #F. This will also leave our variable unmodified #G. Check out the following figure.

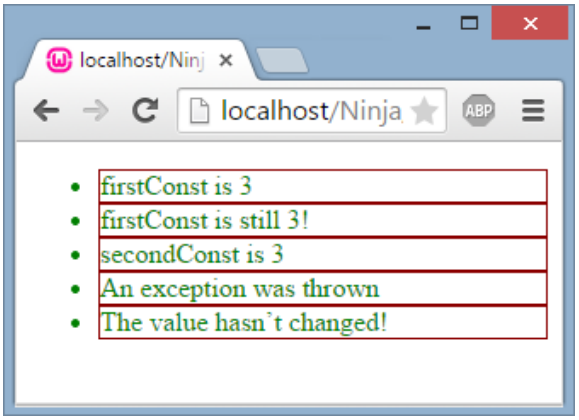


Figure 5.8 Check the behavior of const variables

Now we'll study the minutia of the relationship between different types of variables and lexical environments.

5.3.2 Variable definition keywords and lexical environments

The three types of variable definitions: `var`, `let`, and `const` can also be divided by their relationship with the lexical environment. In that case, we can put `var` on one side, and `let` and `const` on the other. The difference is when you use `var`, the variable is defined in the closest function or global lexical environment (blocks are ignored!), and when you use `let` and `const`, the variable is defined in the closest lexical environment. This effectively means that you use `var` to define either global scoped or function scoped variables, and `let` and `const` to define block scope, function scope, and global scope variables, depending on where you define them. This might feel a bit vague, so let's take a look at a simple example.

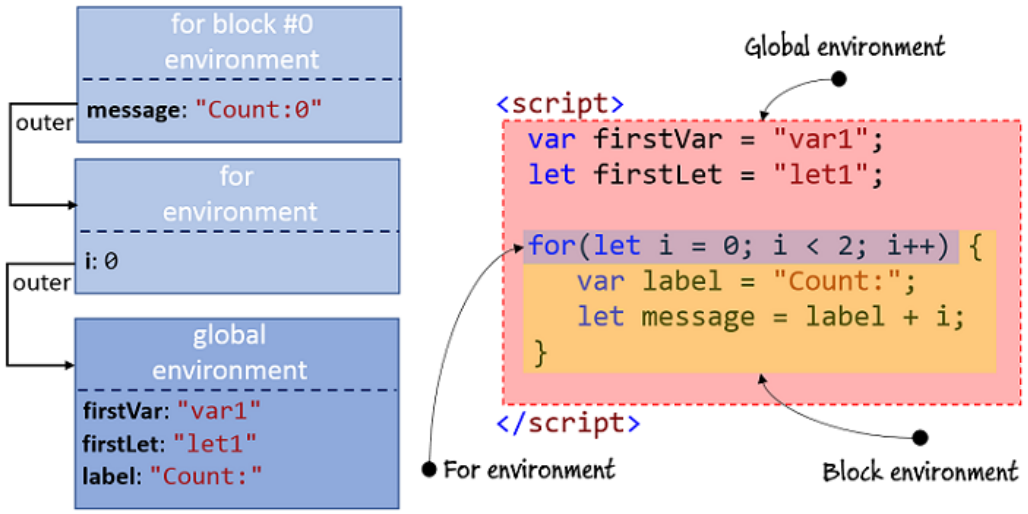


Figure 5.9 The behavior of block and global scope variables depending on the variable declaration keyword

The left side of figure 5.9 shows the state of the lexical environments when finishing the first iteration of the for loop. In this case, we have three lexical environments: the global environment, for global code outside all functions and blocks, the environment bound to the `for` loop, and the block environment for the `for` loop body. As we've already mentioned, the `var` keyword is used to define global scoped or function scoped variables, and the `let` keyword (as well as the `const` keyword) is used to define variables in the closest lexical environment. For these reasons, the global environment, which is created only once per program execution, contains identifier bindings for variables: `firstVar`, defined with the keyword `var` outside of all functions and blocks; `firstLet`, defined with the keyword `let`, but outside any blocks; and `label`, defined within the `for` loop block, *but* with the keyword `var`!

The `for` lexical environment, on the other hand, is created whenever a `for` statement is reached. In this case, it contains one identifier binding: `i`, because the initialization part of the `for` statement contains a variable defined with the keyword `let`.

Finally, since the body of our `for` statement is a block of code, every time the program execution enters a new iteration of the loop, a new lexical environment is created. At the end of first iteration, that environment will contain a single identifier binding for the variable `message`, because it is the only variable defined within that block with the keyword `let`.

Now that we've seen how block lexical environments work in terms of variables, let's study how they are tracked during program execution.

PROGRAM EXECUTION AND BLOCK LEXICAL ENVIRONMENTS

In figure 5.6 we've already gone through how function lexical environments are tracked. In a nutshell, whenever a function is created, it also gets a reference to the lexical environment in

which it was created. Then, when that function is invoked, a new function lexical environment is created, whose outer environment is set to the environment in which the called function was created. But now, we'll take a deeper look into how we keep track of block environments, in the next figure.

```
<script>
  var firstVar = "var1";
  let firstLet = "let1";

  for(let i = 0; i < 2; i++) {
     var label = "Count:";
     let message = label + i;
  }

</script>
```

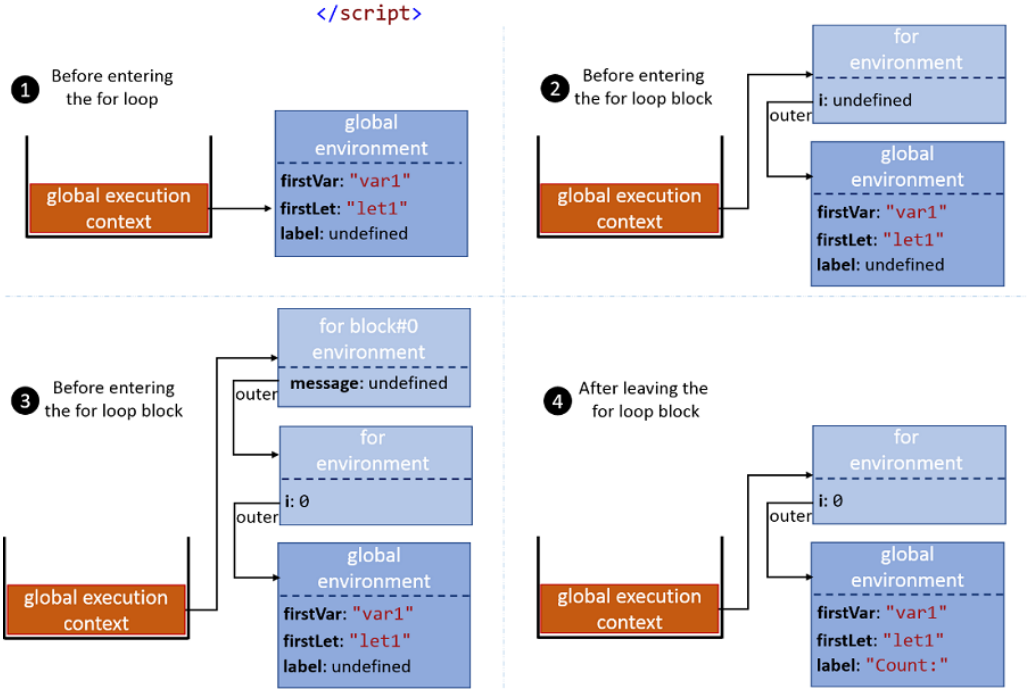


Figure 5.10 Keeping track of block lexical environments during program execution

Figure 5.10 revisits the same simple program, but now from the perspective of program execution. When our JavaScript program starts executing, a new global execution context is created, and with it a new global environment is created #1. Next, when the program execution reaches the `for` loop, a new `for` environment is created. Things work a bit differently with block lexical environments than with function lexical environments. When a

new block environment is created, as its outer environment, the JavaScript engine puts the currently active lexical environment. In our case, this means that the outer environment of the new `for` environment will be the global environment #2. Next, the JavaScript engine has to set the newly created for environment as the current, active lexical environment (in the figure, the arrow now goes from the global execution context to the `for` environment #2). A similar thing happens each time the `for` block is evaluated: a new `for` block environment is created and set as the current active environment, and its outer environment is set to the previously active lexical environment, the `for` environment #3.

An interesting thing happens when the program execution exits a block. For example, after leaving the `for` loop block, the active lexical environment is reset to its outer environment, the `for` environment, and the `for` loop environment is usually simply discarded #4.

Now that you understand the mechanism of how identifier bindings are kept within lexical environments and how lexical environments are linked to program execution, let's discuss the exact process by which identifiers are registered with lexical environments. This will help you better understand some commonly occurring bugs.

5.3.3 Registering identifiers with lexical environments

One of the driving principles behind the design of JavaScript as a programming language was the ease of use – that's one of the main reasons behind not specifying function return types, function parameter types, variable types and so on. Now consider this example. You already know that JavaScript code is executed line by line, in a straight-forward fashion, for example:

```
firstRonin = "Kiyokawa";
secondRonin = "Kondo";
```

First the value "`Kiyokawa`" is assigned to the identifier `firstRonin`, and then value "`Kondo`" to the identifier `secondRonin`. Nothing weird about that, right? But now let's take a look at another example:

```
var firstRonin = "Kiyokawa";
check(firstRonin);
function check(ronin) {
  assert(ronin == "Kiyokawa", "The ronin was checked! ");
}
```

In this case, first 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 our code is executed line by line, should we be able to call the `check` function – our program execution hasn't reached it yet and the JavaScript engine shouldn't know about it?

But if you check the next figure you'll see that all is fine and well. JavaScript isn't too picky about where we define our functions and we can choose to place function declarations before, or even after their respective calls – this isn't something that the developer should fuss about.

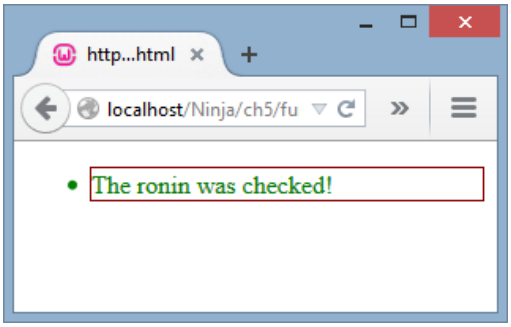


Figure 5.11 The function is indeed visible, even before the execution reaches its definition.

Ease of use aside, if code is executed line by line, how did the JavaScript engine know that a function named check exists?

THE PROCESS OF REGISTERING IDENTIFIERS

The truth is that the JavaScript engine "cheats" a little and that there are actually two phases in the execution of JavaScript code. The first phase is activated whenever a new lexical environment is created. In that phase, the code is not executed, and the JavaScript engine simply visits and registers all declared variables and functions with the current lexical environment. The second phase, JavaScript execution, starts after this is done. The exact behavior depends on the type of variable (`let`, `var`, `const`, function declaration) and the type of the current lexical environment (global, function, or block).

The process goes according to the following steps:

1. If we are creating a function lexical environment, the implicit `arguments` identifier is created, along with all formal function parameters and their values. If we are dealing with some other environment, this step is skipped.
2. If we are creating a global or a function lexical environment, the current code is scanned (without going into the body of other functions) for function *declarations (but not function expressions or lambdas!)*. For each found function declaration a new function is created and bound to an identifier with the exact function name, in the current lexical environment. If that identifier name already exists, its value is overwritten. If we are dealing with block environments, this step is skipped.
3. The current code is scanned for variable declarations. In the case of function and global environments, all variables declared with the keyword `var` defined outside other functions (but they *can* be placed within blocks!), and all variables declared with the keywords `let` and `const` declared outside other functions *and* blocks are found. In the case of block environments, the code is scanned only for variables declared with keywords `let` and `const`, directly in the current block. For each found variable, if the identifier doesn't exist in the current lexical environment, the identifier is registered

and its value initialized to undefined, but if the identifier exists, then it is left with its current value.

Now we'll go through the implications of these rules to some common JavaScript conundrums that can sometimes lead to weird bugs that are easy to detect, but can be tricky to understand. We'll start with why you are able to call a function before it is even declared.

CALLING FUNCTIONS BEFORE THEIR DECLARATIONS

One of the features that makes JavaScript really pleasant to use is the fact that the order of function definitions does not matter – we can call a function even before it is technically declared. Check out the following listing.

Listing 5.5 Accessing the function before its declaration

```
<script>
  assert(typeof fun == "function", "The fun refers to a function even though the
      function is not yet reached!"); #A

  assert(typeof myFunExp == "undefined", "But we cannot access function
      expressions"); #B
  assert(typeof myLamda == "undefined", "Nor lambda functions"); #B


  function fun(){} #C

  var myFunExpr = function(){}; #D
  var myLambda = (x)=>x; #D
</script>
```

#A – We can access a function that isn't yet defined, if the function is defined as a function declaration!
#B – We cannot access functions that are defined as function expressions or lambda functions
#C – The fun function is defined as a function declaration
#D – myFunExp points to a function expressions, and myLambda to a lambda function

In listing 5.5, you can see that you can access the function `fun` #A even before we've defined it #C. We can do this because the function is defined as a function declaration #C, and the second step of the process says 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 #A, the `fun` function exists. The JavaScript engine does this to make things easier for you; it doesn't burden you with the exact order in which you have to place your functions – they exist at the time your code starts executing. But notice that this holds only for function declarations. Function expressions and lambda functions are not a part of this process, and are created when the program execution reaches their definition. This is why you cannot access the `myFunExp` and `myLamda` functions #B.

OVERRIDING FUNCTIONS

The next conundrum that we'll tackle is the problem of overriding function identifiers. Let's take a look at another example:

Listing 5.6 Overriding the function identifier

```
<script>
assert(typeof fun == "function", "We access the function"); #A

var fun = 3; #B

assert(typeof fun == "number", "Now we access the number"); #C

function fun(){} #D

assert(typeof fun == "number", "Still a number"); #E
</script>
```

#A – fun refers to a function
#B – Define a variable fun and assign a number to it
#C – fun refers to a number
#D – A fun function declaration
#E – fun still refers to a number

In listing 5.6, we have a variable declaration #B and a function declaration #D with the same name: `fun`. If you run this code, you'll see that both asserts pass – in the first `assert` #A the identifier `fun` refers to a function and in the second #C and third #E to a number. This behavior follows directly from the steps that are taken when registering identifiers. In the second step, 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 our case, since the identifier `fun` has been encountered in the second step when function declarations are registered, the value `undefined` is not assigned for variable `fun`. This is why the first assertion passes #A. After it, we have an assignment statement #B, in which the number 3 is assigned to the identifier `fun`. When we do this, we lose the reference to the function, and from now on, the identifier `fun` refers to a number #C. During the act of program execution, function declarations are simply skipped, so definition of the `fun` function doesn't have any impact on the value of the `fun` identifier #E.

VARIABLE HOISTING

If you've read a bunch of JavaScript blogs or books explaining identifier resolution, you've probably run into the term *hoisting* – that variable and function declarations are hoisted (or lifted) to the top of function or global scope. As you've seen so far, things are a bit deeper than that, and variables and function declarations technically are not "moved" anywhere – they are simply visited and registered before any code is executed. While understanding *hoisting*, as it is most often defined, is enough to get a basic understanding of how JavaScript scoping works, we've gone much deeper than that, making another step on the path of becoming a true JavaScript ninja.

In the next section, all of the concepts that we've gone through so far will join in closures – an important concept whose deep understanding will forever change you as a JavaScript developer.

5.4 Closures

Succinctly put, a *closure* is a mechanism that allows a function to access all variables, as well as other functions, that are in scope when the function itself is created.

That may seem rather 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, so let's start small with the following listing.

Listing 5.7 A simple closure

```
<script type="text/javascript">
  var outerValue = 'ninja';                                    #A
  function outerFunction() {
    assert(outerValue == "ninja","I can see the ninja.");      #B
  }
  outerFunction();                                             #C
</script>
```

#A Defines a value in global scope
#B Declares a function in global scope
#C Executes the function

In this code example, we declare a variable #A and a function #B in the same scope—in this case, the global scope. Afterwards, we cause the function to execute #C.

As can be seen in figure 5.11, 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!

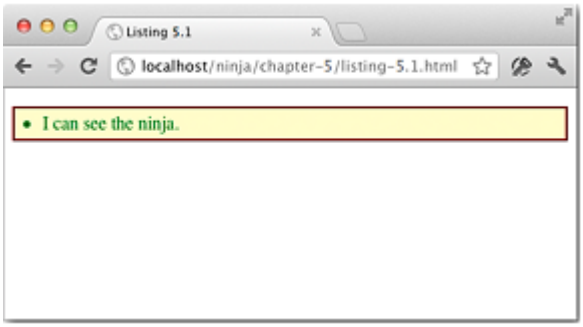


Figure 5.12 Our function has found the ninja, who was hiding in plain sight.

Not impressed? Guess that's not surprising. Because both the `outerValue` and the `outerFunction` are declared in global scope, that scope (which is actually a closure) never goes away (as long as the page is loaded), and it's not surprising that the function can access the variable because it's still in scope and viable. Even though the closure exists, its benefits aren't yet clear.

Let's spice it up a little in the next listing.

Listing 5.8 A not-so-simple closure

```
<script type="text/javascript">
  var outerValue = 'ninja';
  var later;                                          #A
  function outerFunction() {
    var innerValue = 'samurai';                       #B
    function innerFunction() {                        #C
      assert(outerValue,"I can see the ninja.");
      assert(innerValue,"I can see the samurai.");
    }                                                 #C
    later = innerFunction;                            #D
  }
  outerFunction();                                    #E
  later();                                            #F
</script>
```

#A Declares an empty variable that we'll use later.
#B Declares a value inside the function. This variable's scope is limited to the function and cannot be accessed from outside the function.
#C Declares an inner function within the outer function. Note that innerValue is in scope when we create this function.
#D Stores a reference to the inner function in the later variable. Because later is in the global scope, it will allow us to call the function later.
#E Invokes the outer function, which causes the inner function to be created and its reference assigned to later.
#F Invokes the inner function through later. We can't invoke it directly because its scope (along with innerValue) is limited to within outerFunction().

Let's over-analyze the code in `innerFunction()` and see if 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?

We're executing the inner function *after* the outer function has been executed via the trick of copying a reference to the function to a global reference (`later`) #D. When the inner function 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 the `assert` to fail, as `innerValue` is sure to be `undefined`. Right?

But when we run the test, we see the display shown in figure 5.12.

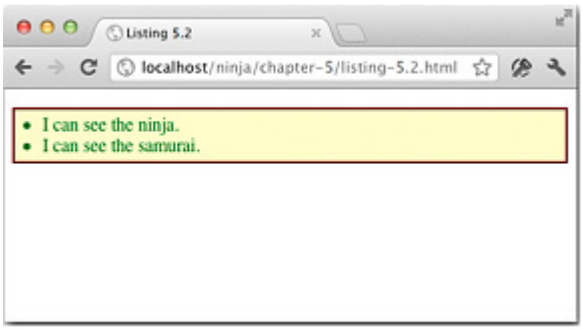


Figure 5.13 Despite trying to hide inside a function, the samurai has been detected!

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, of course, is closures.

When we declared `innerFunction()` inside the outer function, not only was the function declaration defined, but a closure was also created that encompasses not only the function declaration, but also all variables that are in scope *at the point of function definition*.

When `innerFunction()` eventually 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.

Let's take a deep look at what exactly is going on during the execution of the program in the next figure.

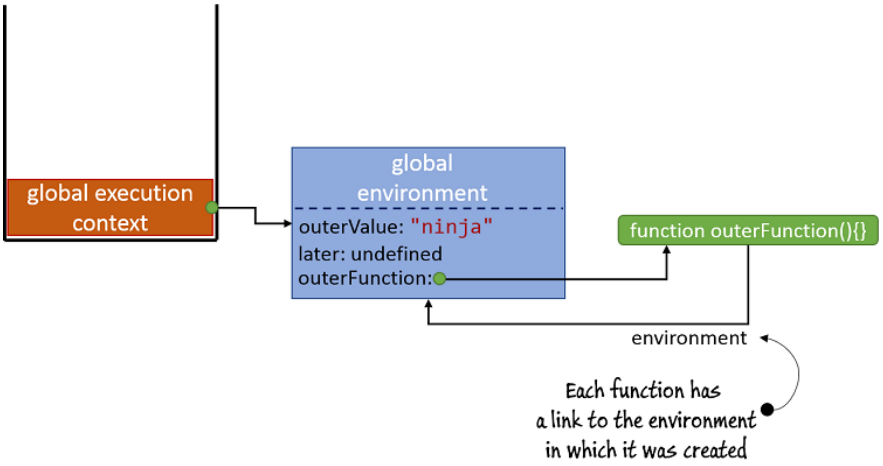


Figure 5.14 The state of the execution stack and the global environment before executing the outerFunction.

Every time when a JavaScript application starts its execution, a global execution context, in which all global code will be executed is created. With it, the JavaScript engine creates the matching global environment, which keeps track of global identifiers. In this case, we have three global identifiers: two standard variables `outerValue` and `later`, and a function called `outerFunction`. Figure 5.13, shows the values of these identifiers before calling the `outerFunction`. The `outerValue` variable references a string "ninja", the `later` variable is `undefined`, and the `outerFunction` identifier references the function `outerFunction`.

We've already discussed how every created function has an internal `[[Environment]]` property that references the lexical environment in which it was created. So in global code, we've defined the `outerFunction`, which as its environment has the global environment.

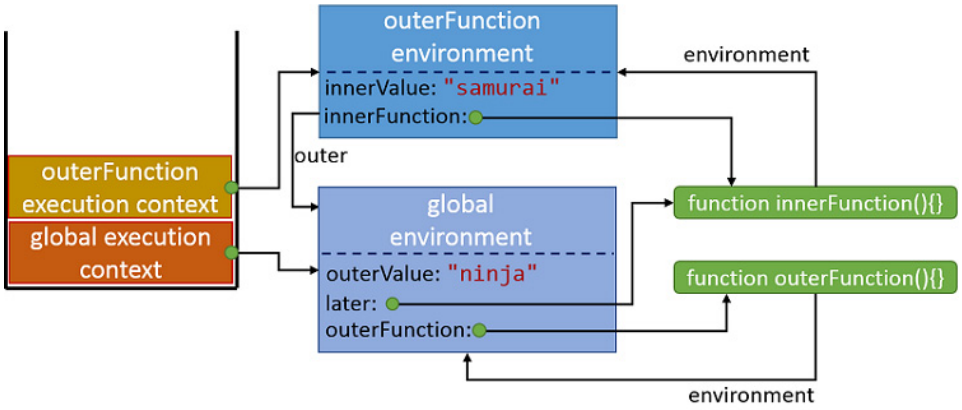


Figure 5.15 The state of the execution stack before returning from the outerFunction.

Whenever a function is called, a new function execution context is created and pushed to the top of the execution context stack. With it, a new function lexical environment that keeps track of function's local identifiers: `innerValue` and `innerFunction`, is also created – the `outerFunction` environment. As we've already mentioned, every function lexical environment has a reference to its outer environment (to implement nesting), and the reference to the outer environment is obtained by checking the `[[Environment]]` property of the called function. In this case, we are calling the `outerFunction`, so its internal `[[Environment]]` property is used as the outer environment of the `outerFunction` environment. Nothing special to it, yet.

During the execution of the `outerFunction`, a new `innerFunction` is created, and its `[[Environment]]` is set to the `outerFunction` environment (since this is the environment in which the function was created). The last assignment in the `outerFunction`, assigns the `innerFunction` to the variable `later`. Referencing the variable `later`, triggers the search for it. First the current lexical environment – outerFunction environment is searched. Since this

environment doesn't have a mapping for that environment, its outer lexical environment is checked – the global environment. The global environment has a mapping for the `later` identifier and a reference to the `innerFunction` is stored in it (see figure 14).

An interesting thing happens when we exit the `outerFunction`, see the following figure.

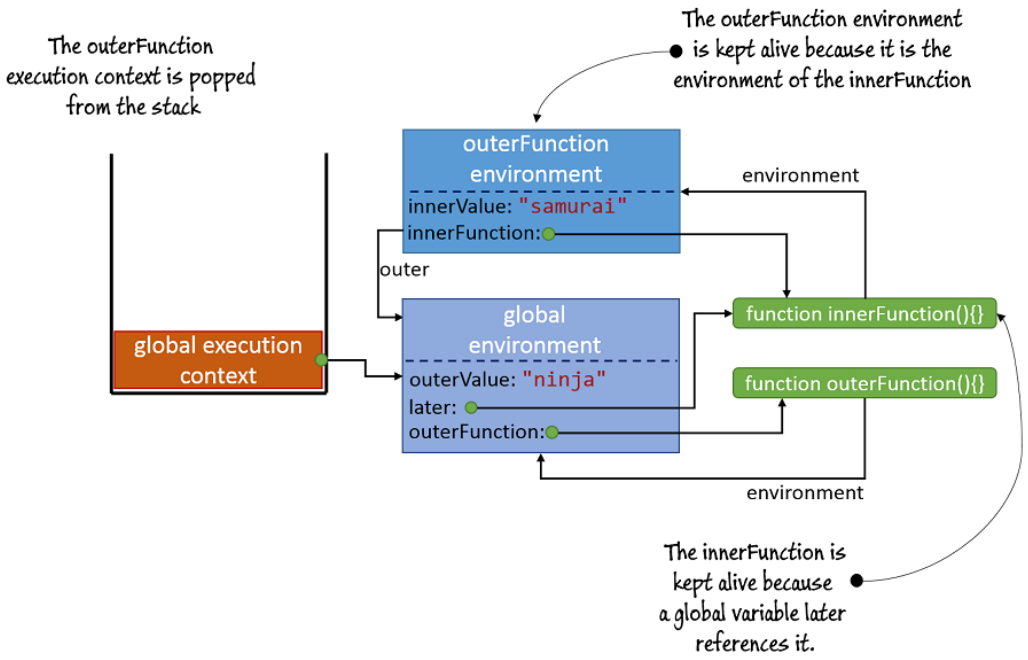


Figure 5.16 When we exit the outerFunction, its execution context is popped from the stack. Normally, this would also lead to the removal of the outerFunction environment. However, the environment is not removed because it is kept alive through the innerFunction.

Every time when the program flow exits a function, the execution context of that function is popped from the execution context stack, so that the execution of the caller function can be continued. Normally, this also leads to the removal of the matching lexical environment. In our case, the removal of the `outerFunction` execution context would lead to the removal of the `outerFunction` environment. However, this will not happen, because the `outerFunction` environment is kept alive because it is the environment of the `innerFunction`, and the `innerFunction` is kept alive because it is referenced by the global `later` variable.

Another interesting thing happens when we call the function referenced by the `later` global variable – the `innerFunction`, see the following figure.

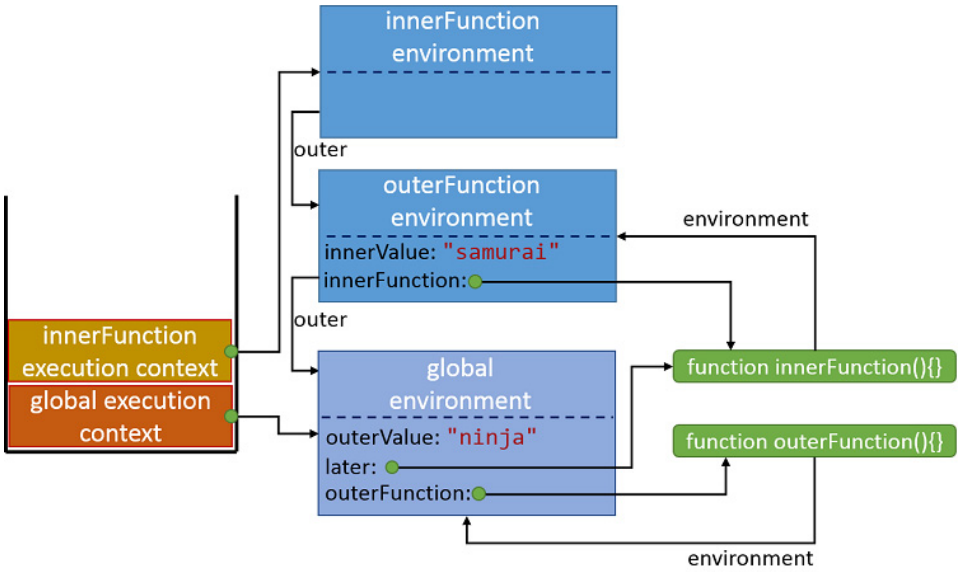


Figure 5.17 The state of the application when executing the innerFunction (through the later() call).

When we call the `innerFunction` through the reference to the variable `later`, a new function execution context is created and pushed to the top of the execution context stack, the innerFunction execution context. With it, a new lexical environment, the innerFunction environment is created. Since we are calling the `innerFunction`, as the outer environment of the new `innerFunction` environment, we set the environment in which the `innerFunction` was created – the `outerFunction` environment.

Now we have everything we need to perform identifier resolution within the body of the `innerFunction`. The first identifier is the `outerValue` identifier. Since there is no identifier with that name in the innerFunction environment, first its outer environment is searched – the outerFunction environment. This environment also doesn't have a mapping to the identifier `outerValue`, so its outer environment – the global environment – is also searched. This time around, the global environment does contain the mapping to the `outerValue` and the value `"ninja"` is returned. The next referenced identifier is the `innerValue` identifier. Again, first we check the current environment – the innerFunction environment. Since it doesn't contain the mapping, its outer environment is searched and the mapping for the `innerValue` identifier and the value "samurai" is found in the `outerFunction` environment.

And that's all there is to the concept of closures – they are a mechanism that make sure that a function can access all variables that are in scope in the moment the function is created. Since in JavaScript, scoping is implemented through lexical environments, a closure simply means that a lexical environment, and all variables defined in it is kept alive as long as

someone references it, either another environment through its outer reference, or a function created in the lexical environment through its internal `[Envirnonment]` property.

In the next chapter we'll take a closer look to all wonderful, practical usages of closures, but for now we would like you to notice one downside of closures. With closures, the whole environment accessible at the moment when a function is created (yes, that even includes the environments accessible by following the outer references) is kept alive for as long as the function itself is alive! Sure, some JavaScript engines could optimize a bit and save only the variables explicitly referenced by the function, and not the whole environment; but that's not something to count on. For example, figure 5.14 also shows the state of the application when the program flow exits the `innerFunction`. Notice how the outerFunction environment is kept alive even after we've exited the `innerFunction`, because the `innerFunction` is still accessible through the global variable `later`. The only way to allow the JavaScript engine to free up the memory reserved for the `outerFunction` environment is to remove the reference to the `innerFunction` by reassigning (or deleting) the global variable `later`. This is something that you should keep in mind when developing complex JavaScript applications.

5.5 Summary

In this chapter we've made a deep exploration of the process of program execution, identifier resolution, and closures in JavaScript. We've seen how:

- The program execution is tracked through execution contexts organized in an execution context stack.
- Your global code is executed in a global execution context that is created once for the whole application, and all your function code is executed in function execution contexts, which are created every time you invoke a function. In order to keep track of program execution, the execution contexts are organized in an execution context stack, where new execution context are pushed on the top, and when a function finishes with its execution, the matching execution context is popped from the top of the stack.
- Lexical environments are used to keep track of identifier bindings. In the new ES6 version of JavaScript, a lexical environment can be bound to functions and blocks of code, which means that modern JavaScript has both function scope and block scope.
- Lexical environments are based on the idea of code nesting, that one function can be defined within another. In that case, the inner function has access to all variables accessible in the outer function.
- In JavaScript, there are three different keywords for defining variables: `var`, `let`, and `const`:
  - Keyword `var` can be used to define typical variables in global or function scope.
  - Keywords `const` and `let` are used to define variables in the closest scope. Can be used to define block scoped, function scoped, and global scoped variables.
  - Unlike `var` and `let`, which are used to define normal, run-of-the-mill variables, the `const` keyword is used to create immutable variables whose value can be set only once.

- When a new lexical environment is created, before any code is executed, the JavaScript engine visits all variables declared in that environment, creates functions defined through function declarations, and sets the value of declared variables to `undefined`. When this is performed, the execution of JavaScript code starts.
- We've seen how when a function is created it gets a reference to the lexical environment in which it was created (the `[[Environment]]` internal property). This enables us to access variables that are in scope when the function is created – the function "closes over" these variables and keeps them alive for as long as the function itself is alive.

Now that you have a deep understanding on how identifiers are handled in JavaScript, in the next chapter, you'll see how take the knowledge you gained in this and the previous chapter and put it to use.

6

Wielding functions

This chapter covers

- Using closures to simplify development
- Using the function context to get our way
- Treating functions as objects
- Dealing with variable-length argument lists

In the previous two chapters, first we focused on how JavaScript treats functions as first-class objects enabling a functional programming style, and then we moved onto the intricacies of identifier resolution and closures. In this chapter, we'll use that knowledge to explore advanced function techniques that can significantly simplify your code, improve its performance, and avoid some common bugs.

We'll start with advance usages of function closures, by showing you how to use closures to mimic a language feature strangely missing from JavaScript – private object variables. We'll continue by taking another step into the land of closures by exploring the relationship between closures and callbacks that can significantly simplify your code. By the time you're done with these, you'll have a deep understanding of all there's to know about closures.

Next, we'll go on to explore how to bind functions to certain objects to make sure that our function context is what we want it to be, in the process learning to avoid a common source of bugs.

We'll also show you how to take advantage of functions as first class objects, for example, by increasing their performance through a process known as function memoization.

Finally, we'll focus on dealing with variable-length argument lists, with a special focus on new ES6 features that make our lives as developers a bit more pleasant.

The examples in this chapter were purposefully chosen to expose secrets that will help you to truly understand JavaScript functions. Many are simple in nature, but they expose important concepts that will be broadly applicable to the dilemmas we're bound to run into in future coding projects.

Without further ado, let's take the functional JavaScript knowledge that we now possess in our two hands and wield it like the mighty weapon that it is.

6.1 Putting closures to work

In Chapter 5, we've seen, on a theoretical level, how in JavaScript functions are closures that keep alive the variables that were in scope when the function itself was created. But now, let's see how we can put that knowledge to work to achieve more elegant code!

6.1.1 Mimicking private variables

In many programming languages, there exists a concept of private variables, where properties of the object that are hidden from outside parties. This is a useful feature, since we don't want to overburden the users of our objects (usually ourselves) with unnecessary implementation details when accessing those objects from other parts of the code. Unfortunately, JavaScript as a language doesn't have support for private variables. But by using the concept of a closure, we can achieve an acceptable approximation, as demonstrated by the following code.

Listing 6.1 Using closures to approximate private variables

```
<script type="text/javascript">
  function Ninja() {                                    #A
    var feints = 0;                                     #B
    this.getFeints = function(){                        #C
      return feints;
    };                                                  #C
    this.feint = function(){                            #D
      feints++;
    };                                                  #D
  }
  var ninja1 = new Ninja();                             #E
  ninja1.feint();                                       #F
  assert(ninja1.getFeints() == 1,                       #G
         "We're able to access the internal feint count.");     #G
  assert(ninja1.feints === undefined,                           #H
         "And the private data is inaccessible to us.");        #H

  var ninja2 = new Ninja();      #I
  assert(ninja2.getFeints() == 0,
         "The second ninja object gets it's own feints variable."); #I

</script>
```

#A Defines the constructor for a Ninja.
#B Declares a variable inside the function (constructor). 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.
#C Creates an accessor method for the feints counter. As the variable is not accessible to code outside the constructor, this is a common way to give read-only access to the value.

#D Declares the increment method for the value. Because the value is private, no one can screw it up behind our backs; they are limited to the access that we give them via methods.
#E Now for testing; first we construct an instance of Ninja.
#F Calls the feint() method, which increments the count of the number of times that our ninja has feinted.
#G Verifies that we can't get at the variable directly.
#H Shows that we've caused the value to increment to 1, even though we had no direct access to it. We can affect the feints value because, even though the constructor in which it's declared has finished executing and gone out of scope, the feints variable is bound into the closure (think protective bubble) created by the declaration of the feint() method, and is available to that method.
#I When we create a new ninja2 object with the Ninja constructor, the ninja2 objects gets it's own feints variable

In listing 6.1, we create a function `Ninja` that is to serve as a constructor. We introduced the concept of using a function as a constructor in chapter 4, and we'll be taking an in-depth look at it again in chapter 8. For now, just 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 (the variable is stored in the lexical environment associated with this constructor call). 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.")

An implementation method, `feint()`, is then created to give us control over the value of the variable in a controlled fashion. In a real-world application, this might be some business method; in this example, it merely increments the value of `feints`.

After the constructor has done it's duty, we can call the `feint()` method, on the newly created `ninja1` object.

Our tests show that we can use the accessor method to obtain the value of the private variable, but that we cannot access it directly. This effectively prevents us from being able to make uncontrolled changes to the value of the variable, just as if it were a private variable in a fully object-oriented language.

Figure 6.1 shows the state of the application after creating the first `Ninja` object.

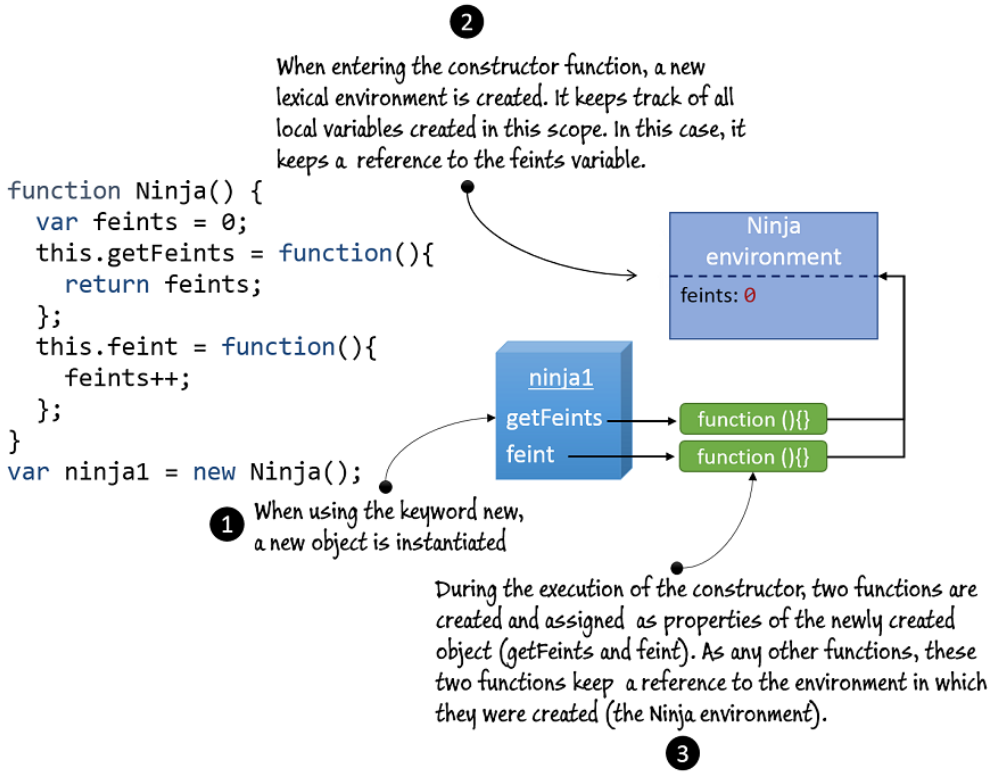


Figure 6.1 Private variables are realized as closures created by object methods created in the constructor.

Now let's use our knowledge of intricacies of identifier resolution from the previous chapter to better understand how closure come into play here: JavaScript constructors are simply functions invoked with the keyword `new`, so *every* time we invoke a constructor, a new lexical environment, which keeps track of variables local to the constructor, is created. In our example, a new `Ninja` environment is created, that keeps track of the all variables defined in the constructor, in this case the `feints` variable. You'll also remember that whenever a function is created, through an internal [[Environment]] property, it keeps a reference to the lexical environment in which it was created. This is used so that you can access the current variables whenever the function is called. In our case, the functions associated with the `getFeints` and `feint` properties are created in the lexical environment of the constructor, and the current constructor's lexical environment is set as their environment. Since these two functions are accessible from outside of the constructor function (they can be called as object methods), they've created a *closure* around the `feints` variable.

When we create another `Ninja` object #9, the whole process is repeated again. Figure 6.2 shows the state of the application after creating the second `Ninja` object.

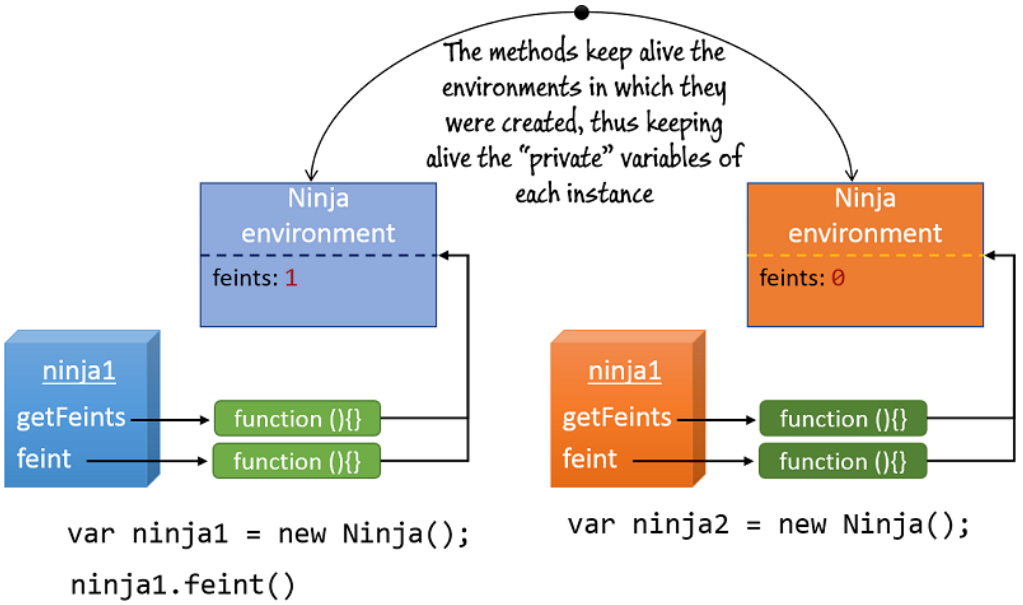


Figure 6.2 The methods of each instance create closures around the "private" instance variables.

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! However, as you might've figured out by now, it's not really that these "private" variables are private properties of the object, they are simply variables kept alive by the object methods created in the constructor. Let's take a look at one interesting side-effect of this.

CAVEAT

In JavaScript, there is nothing stopping you from adding properties to the object after the constructor function has returned. For example, you can easily rewrite the code from Listing 6.1, into something like along the following lines.

Listing 6.2 Private variables are accessed through functions, not through objects!

```
<script type="text/javascript">
  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;                #A

  assert(imposter.getFeints () == 1,                    #B
        "We can access the 'private' variable of ninja1 through the imposter!")#B
</script>
```

#A – Make the getFeints function of ninja1 accessible through object ninja2
#B – Verify that we can access the supposedly private variable of ninja1.

The listing modifies the source code in a way that it assigns the `ninja1.getFeints` method to a completely new object `imposter`. Then, when we call the `getFeints` function on the `imposter` object, we test that we can access the value of variable `feints` created when `ninja1` was instantiated, thus proving that we are actually faking this whole "private" variable thing, see the following figure.

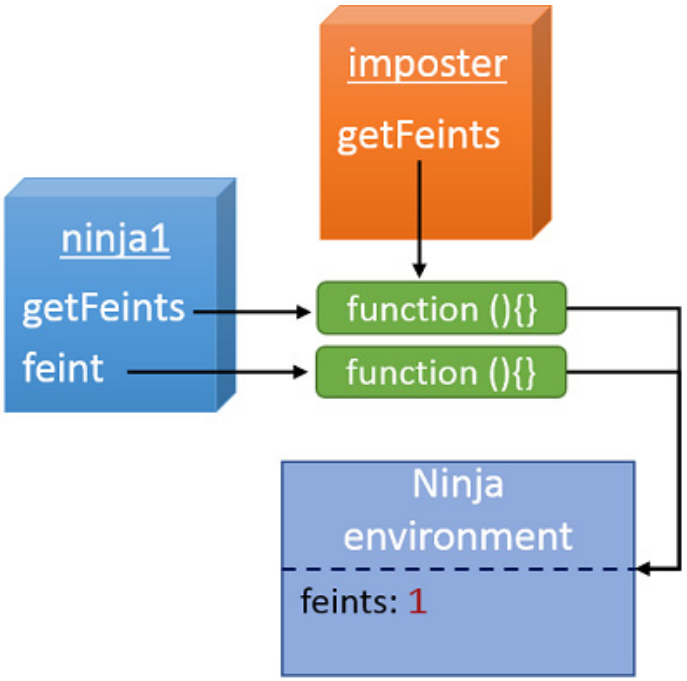


Figure 6.3 We can access the "private" variables through functions, even if that function is attached to another object!

We've used this example just to illustrate that there aren't really any private object variables in JavaScript, but that we can use closures created by object methods to have a "good-enough" alternative to private variables. Still, even though it isn't the "real-thing", lots of developers find this way of information hiding useful.

This was a glimpse into the world of object-oriented JavaScript, which we'll explore in much greater depth in chapter 8. For now, let's focus on another common use of closures.

6.1.2 Closures and callbacks

Another one of the most common areas in which we can use closures is when dealing with callbacks – when a function is asynchronously called at an unspecified later time. Often, within such functions we frequently need to access outside data. Let's look at an example that creates a simple animation with callback timers, in the next listing.

Listing 6.3 Using a closure in a timer interval callback

```
<div id="box1">First Box</div>                                          #1
      <div id="box2">Second Box</div>                                    #1
<script type="text/javascript">
  function animateIt(elementId) {
    var elem = document.getElementById(elementId);                       #2
    var tick = 0;                                                        #3
    var timer = setInterval(function(){                                  #4
      if (tick < 100) {                                                  #A
        elem.style.left = elem.style.top = tick + "px";
        tick++;
      }                                                                  #A
      else {
        clearInterval(timer);
        assert(tick == 100,                                              #B
               "Tick accessed via a closure.");
        assert(elem,
               "Element also accessed via a closure.");
        assert(timer,
               "Timer reference also obtained via a closure." );
      }
    }, 10);                                                              #C
  }
  animateIt("box1");                                                     #D
  animateIt("box2");                                                     #D
</script>
```

#1 Creates the elements that we're going to animate.
#2 Inside the animateIt() function, we get a reference to that element.
#3 Establishes a counter to keep track of animation ticks (steps).
#4 A built-in function that creates and starts an interval timer given a callback.
#A The timer callback will be invoked every 10 milliseconds. For 100 ticks it will adjust the position of the element
#B After 100 ticks we stop the timer and perform tests to assert that we can see all relevant variables needed to perform the animation.
#C the setInterval duration, the callback will be called every 10ms
#D Now that it's all set up, we set it in motion!

What's especially important about the code in listing 6.3 is that it uses a single anonymous function #4 to accomplish the animation of the target element #1. That function accesses three variables, via a closure, to control the animation process.

The three variables (the reference to the DOM element #2, the tick counter #3, and the timer reference #4) all must be maintained *across* the steps of the animation. And we need to keep them out of the global scope.

Why do you think we need to do that?

The answer is, as you might guess – closures. 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 upon the closures to make them available to the timer callback invocations, each animation gets its own private set of variables, as shown in figure 6.4.

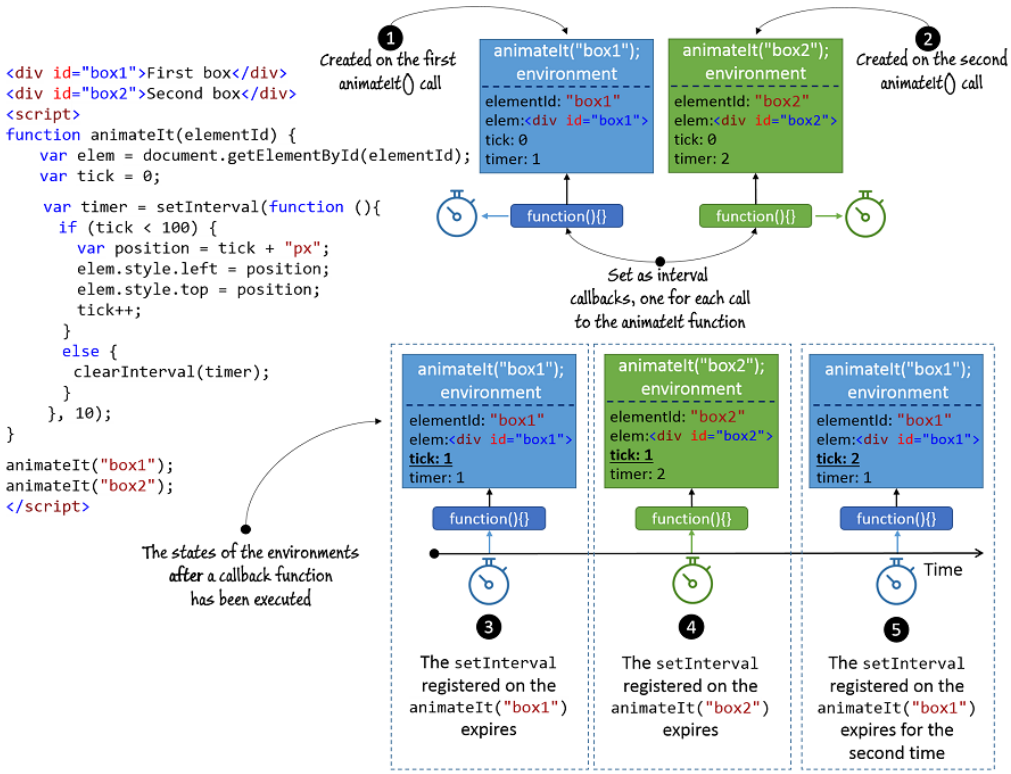


Figure 6.4 By creating multiple closures, we can do many things at once. Every time when an interval expires, the callback function reactivates the environment that was active at the time of callback creation. The closure of each callback automatically keeps track of its own set of variables.

Every time when we call the `animateIt` function, a new function lexical environment is created #1 #2 that keeps track of the set of variables important for that animation (the elementId, the element that is being animated, the current number of ticks, and the id of the timer doing the animation). That environment will be kept alive as long as there is at least one function that makes use of its variables (through closures). In this case, the browser will keep alive the `setInterval` callback up until we call the `clearInterval` function. What's even more interesting is that every time the `setInterval` callback is called by the browser, the environment that was active when the callback function was created, is reactivated.

In this example, we have two calls to the `animateIt` function. This means that two different lexical environments will be created #1 #2, one for each call. 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 – keeping us from the trouble of manually mapping the callback and the active variables #3 #4 #5.

Without closures, doing multiple things at once, whether event handling, animations, or even Ajax requests, would be incredibly difficult. If you've been waiting for a reason to care about closures, this is it!

There's another important concept that this example makes clear. Not only do we see the values that these variables had at the time the closure was created, but we can also update them within the closure while the function within the closure executes. In other words, the closure isn't simply a snapshot of the state of the scope at the time of creation, but an active encapsulation of that state that can be modified as long as the closure exists.

This example is a particularly good one for demonstrating how the concept of closures is capable of producing some surprisingly intuitive and concise code. By simply including the variables in the `animateIt()` function, we create a closure without needing any complex syntax.

Now that we've seen closures used in various contexts, let's take a look at some of the other, advanced usages of functions, starting with how to bend function contexts to our wills.

6.2 Binding function contexts

During our discussion of function contexts in the previous chapter, we saw how the `call()` and `apply()` methods could be used to manipulate the context of a function. While this manipulation can be incredibly useful, it can also be potentially harmful to object-oriented code.

Consider the following listing, in which a function that serves as an object method is bound to a DOM element as an event listener.

Listing 6.4 Binding a specific context to a function

```
<button id="test">Click Me!</button>                    #1
<script type="text/javascript">
  var button = {                                          #2
    clicked: false,
    click: function(){                                     #3
      this.clicked = true;
```

```
      assert(button.clicked,"The button has been clicked");          #4
    }
  };
  var elem = document.getElementById("test");                        #5
  elem.addEventListener("click",button.click,false);                 #5
</script>
```

#1 Creates a button element to which we'll assign event handler.
#2 Defines an object to retain state regarding our button. With it, we'll track whether the button has been clicked or not.
#3 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.
#4 Within the method, we test that the button state has been correctly changed after a click.
#5 Establishes the click handler on the button.

In this example, we have a button #1, and we want to know whether it has ever been clicked or not. In order to retain that state information, we create a backing object named `button #2`, in which we'll store the clicked state. In that object, we'll also define a method that will serve as an event handler #3 that will fire when the button is clicked. That method, which we establish as a click handler for the button #5, sets the clicked property to `true` and then tests #4 that the state was properly recorded in the backing object.

When we load the example into a browser and click the button, we see by the display of figure 6.5 that something is amiss; the stricken text indicates that the test has failed.

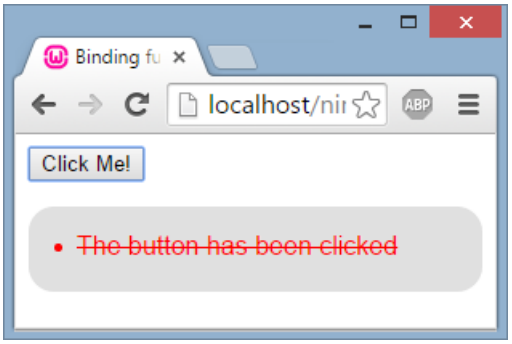


Figure 6.5 Why did our test fail? Where did the change of state go? Usually, the event callback's context is the object raising the event (in our case the HTML button, and not our button object).

The code in listing 6.4 fails because the context of the `click` function is *not* referring to the `button` object as we intended.

Recalling the lessons of chapter 4, if we had called the function via

```
button.click()
```

the context *would* indeed have been the button. But in our 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! Often we don't want such behavior, let's see how we can fix it!

USING THE BIND METHOD

As you've already seen in chapter 4, a JavaScript function, being a special kind of object, can have its own properties, and we've already met two of these method properties: `call` and `apply`, that we can use to have greater control over the context and arguments of our function invocations. In addition to these methods, every function also has access to the `bind` method that, in short, creates a new function with the same function body, a function that is always bound to certain object, regardless of the way we invoke it.

Setting the context to the target element when an event handler is invoked is a perfectly reasonable default, and one that we can, and *will*, count on in many situations. But in this instance, it's in our way. Luckily, the `bind` function gives us a way around this. Take a look at the following code, which updates the code of listing 6.4 with additions (in bold) to bend the function context to our wills.

Listing 6.5 Binding a specific context to an event handler

```
<button id="test">Click Me!</button>
<script type="text/javascript">
  var button = {
    clicked: false,
    click: function(){
      this.clicked = true;
      assert(button.clicked,"The button has been clicked");
      console.log(this);
    }
  };
  var elem = document.getElementById("test");
  var boundClick = button.click.bind(button);
  elem.addEventListener("click", boundClick ,false);      #1
  assert(button.click !== boundClick,                     #2
        "When calling bind, a new function is created");
</script>
```

#1 Uses the bind function to create a new function with the same body, but a function that will always have button object as context, regardless of how it is called.
#2 The bound function, while having the same code as the button.click function, is a different object!

The secret sauce that we've added here is the `bind()` method #1. This method, available to all functions, is designed to *create* and return a *new* function that is bound to the passed in object (in our 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 (it has the same code in its body). It is important to emphasize that calling the `bind` method creates a completely new function #2!

Now, when the button is clicked, that bound function will be invoked, and the context will be the `button` object.

6.3 Fun with functions as objects

As we've consistently harped on throughout this book, functions in JavaScript aren't like functions in many other languages. JavaScript gives functions many capabilities, not the least of which is their treatment as first-class objects.

We've seen that functions can have properties, can have methods, can be assigned to variables and properties, and generally enjoy all the abilities of plain vanilla objects, but with an amazing superpower: they're callable.

In this section, we'll examine some ways that we can exploit the similarities that functions share with other object types. One capability that may have surprised you is that, just as with any other object, we can attach properties to a function:

```
var obj = {};
var fn = function(){};
obj.prop = "hitsuke (distraction)";
fn.prop = "tanuki (climbing)";
```

This aspect of functions can be used in a number of different ways throughout a library or general on-page code, and this is especially true when it comes to topics like event callback management. Let's look at a couple of the more interesting things that can be done with this capability; first we'll look at storing functions in collections and then at a technique known as "memoization."

6.3.1 Storing functions

There are times when we may want to store a collection of related but unique functions, event callback management being the most common example (and one that we'll be examining in excruciating detail in chapter 13). When adding functions to such a collection, a challenge we can face is determining which functions are actually new to the collection and should be added, and which are already resident and shouldn't be added.

An obvious, but naïve, technique would be to store all the functions in an array and loop through the array checking for duplicate functions. Unfortunately, this performs poorly, and as ninjas we want to make things work *well*, not merely work.

We can make use of function properties to achieve this with an appropriate level of sophistication, as shown in in the next listing.

Listing 6.6 Storing a collection of unique functions

```
<script type="text/javascript">
  var store = {
    nextId: 1,                                          #1
    cache: {},                                          #2
    add: function(fn) {                                 #3
      if (!fn.id) {
        fn.id = store.nextId++;
        store.cache[fn.id] = fn;
        return true;
      }
    }
```

```
                                                          #3
  };
  function ninja(){}
  assert(store.add(ninja),                                #4
         "Function was safely added.");
  assert(!store.add(ninja),
         "But it was only added once.");                  #4
</script>
```

#1 Keeps track of the next available id to be assigned.
#2 Creates an object to serve as a cache in which we'll store functions.
#3 Adds functions to the cache, but only if they're unique.
#4 Tests that all works as planned.

In this listing, we create an object assigned to 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 #1, and one within which we'll cache the stored functions #2. Functions are added to this cache via the `add()` method #3.

Within `add()`, we first check to see if an `id` property has been added to the function, and if so, 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 #4, the function is only added once, as shown in figure 6.6.

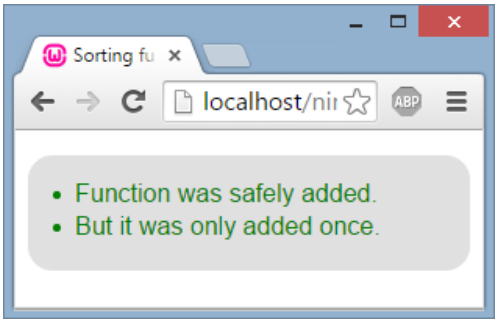


Figure 6.6 By tacking a property onto a function, we can keep track of it. In that way, we are sure that our function has been added only once.

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

6.3.2 Self-memoizing functions

Memoization (no, that's not a typo) is the process of building a function that's capable of remembering its previously computed values. This can markedly increase performance by avoiding needless complex computations that have already been performed.

We'll take a look at this technique in the context of storing the answer to expensive computations.

MEMOIZING EXPENSIVE COMPUTATIONS

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

From the outside, the function will appear 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 6.7 Memoizing previously computed values

```
<script type="text/javascript">
  function isPrime(value) {
    if (!isPrime.answers) isPrime.answers = {};                    #1
    if (isPrime.answers[value] != null) {                           #2
      return isPrime.answers[value];
    }                                                               #2
    var prime = value != 1; // 1 can never be prime

    for (var i = 2; i < value; i++) {
      if (value % i == 0) {
        prime = false;
        break;
      }
    }
    return isPrime.answers[value] = prime;                          #3
  }
  assert(isPrime(5), "5 is prime!" );                               #4
  assert(isPrime.answers[5], "The answer was cached!" );             #4

</script>
```

#1 Creates the cache
#2 Checks for cached values
#3 Stores the computed value
#4 Tests that it all works

Within the `isPrime()` function, we start by checking to see if the `answers` property that we'll use as a cache has been created, and if not, we create it #1. The creation of this initially empty object will only occur on the first call to the function; after that, the cache will exist.

Then we check to see if the answer for the passed value has already been cached in `answers` #2. Within this cache, we'll store the computed answer (`true` or `false`) using the value as the property key. If we find a cached answer, we simply 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 #3.

Some simple tests #4 show that the memoization is working!

This approach has two major advantages:

- The end user enjoys performance benefits for function calls asking for a previously computed value.
- It happens completely seamlessly and behind the scenes; neither the end user nor the page author need 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; there are disadvantages that may need to be weighed against the advantages:

- Any sort of caching will certainly sacrifice memory in favor of performance.
- Purists may consider that caching is a concern that should not be mixed with the business logic; a function or method should do one thing and do it well.
- 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.

The ability to possess properties, just like the other objects in JavaScript, isn't the only superpower that functions have. Much of a function's power is related to its context, and we'll explore an example of that next.

6.4 Variable-length argument lists

JavaScript, as a whole, is very flexible in what it can do, and much of that flexibility defines the language as we know it today. One of these flexible and powerful features is the ability for functions to accept an arbitrary number of arguments. This flexibility offers developers great control over how their functions, and therefore their applications, can be written.

Let's take a look at a few prime examples of how we can use flexible argument lists to our advantage. We'll see

- How to supply multiple arguments to functions that can accept any number of them
- How to use variable-length argument lists to implement function overloading
- How to understand and use the `length` property of argument lists

Because JavaScript has no function overloading (the JavaScript engine simply takes the lastly defined function), the flexibility of the argument list is key to gaining similar advantages that overloading gives us in other languages.

Let's start with using `apply()` to hand off a variable number of arguments.

6.4.1 Supplying variable arguments

With any language, there are often things we need to do that seem to have been mysteriously overlooked by the developers of the language, and JavaScript is no exception. One of these

odd vacuums involves finding the smallest or the largest values contained within an array. It seems like that would be done often enough to warrant inclusion in JavaScript, but if we poke around, the closest thing we'll find is a set of methods on the `Math` object named `min()` and `max()`.

At first we might think that these methods are the answer to our problem, but on examination, we'll see that each of these methods expects a variable-length argument list, and not an array. How silly not to have provided both.

That means calls to `Math.max()`, for example, could look like this:

```
var biggest = Math.max(1,2);
var biggest = Math.max(1,2,3);
var biggest = Math.max(1,2,3,4);
var biggest = Math.max(1,2,3,4,5,6,7,8,9,10,2058);
```

When it comes to arrays, we can't very well resort to something like this:

```
var biggest = Math.max(list[0],list[1],list[2]);
```

Unless we know exactly how big the array is, how would we know how many arguments to pass? And even if we did know the array size, that's far from a satisfactory solution.

Before abandoning `Math.max()` and resorting to looping through the contents ourselves to find the minimum and maximum values, let's pull on our ninja hoods and ponder whether there's an easy and supported way to use an array as a variable-length argument list.

Eureka! The `apply()` method!

USING APPLY() TO SUPPLY VARIABLE ARGUMENTS

You may recall, the `call()` and `apply()` methods exist as methods of *all* functions—even of the built-in JavaScript functions. Let's see how we can use that ability to our advantage in defining our array-inspecting functions, as shown in the next listing.

Listing 6.8 Generic `min()` and `max()` functions for arrays

```
<script type="text/javascript">
  function smallest(array){                    #1
    return Math.min.apply(Math, array);
  }                                           #1
  function largest(array){                      #2
    return Math.max.apply(Math, array);
  }                                             #2
  assert(smallest([0, 1, 2, 3]) == 0,          #3
         "Located the smallest value.");
  assert(largest([0, 1, 2, 3]) == 3,
          "Located the largest value.");          #3
</script>
```

#1 Implements a function to find the smallest value
#2 Implements a function to find the largest value
#3 Tests the implementations

In this code we define two functions: one to find the smallest value within an array #1, and one to find the largest value #2. Notice how both functions use the `apply()` method to supply the value in the passed arrays as variable-length argument lists to the `Math` functions.

A call to `smallest()`, passing the array `[0,1,2,3]` (as we did in our tests #3), results in a call to `Math.min()` that's functionally equivalent to

```
Math.min(0,1,2,3);
```

Also note that we specify the context as being the `Math` object. This isn't necessary (the min() and max() methods will continue to work regardless of what's passed in as the context), but there's no reason not to be tidy in this situation.

Now we know how to *use* variable-length argument lists when calling functions. This is an ability that's been available in JavaScript for a long time, but now; let's take a look at a recent addition to JavaScript – rest parameters.

6.4.2 Rest parameters

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 is 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 result of using the `Math.max()` function (which we've already become familiar with) on the remainder of the argument values. In the old versions of JavaScript, this would require some workarounds – we would have to use the `slice()`array method on the `arguments` object to create a new array of arguments, but this time leaving out the first element. Luckily, in ES6, we don't need to jump through all these hoops. We can use *rest parameters,* see the following listing.

Listing 6.9 Using the rest parameters

```
<script type="text/javascript">
  function multiMax(first, ...remainingNumbers){    #1
    return first * Math.max.apply(Math, remainingNumbers);
  }
  assert(multiMax(3, 1, 2, 3) == 9,                              #2
        "3*3=9 (First arg, by largest.)");
   </script>
```

#1 The rest parameters are prefixed with ...
#2 The function is called just like any other function

If you prefix the last named argument of a function with ellipsis ("...") #1, that argument is called the *rest* arguments, and, very cleverly, contains the rest of the arguments. It is an array that contains the values sent by the function call that weren't assigned to the standard, named arguments. For example, in our case, the `multiMax` function is called with four arguments #2, and in the body of the `multiMax` function, the value of the first argument – `3`

will be assigned to the first `multiMax` function parameter – `first`. Since the second parameter of the function is the rest parameter, this means that all remaining arguments (`1, 2, 3`) will be placed in a new array – `remainingNumbers`.

DIFFERENCE BETWEEN REST PARAMETERS AND THE ARGUMENTS OBJECT

As we've seen in chapter 4, whenever we call a function, besides the formal function parameters specified in the function's signature, there is also an implicit parameter available in the function's body – the `arguments` object, an array-like object that enables us to access all arguments sent through a function call by using array-like index notation (for example `arguments[0]` accesses the value of the first passed-in argument). So in a way, the implicit `arguments` parameter and the rest parameter play a similar role – they enable you to access the excess arguments passed in the function call. However, it is also important to emphasize their differences:

- The rest parameters are the excess arguments that haven't been given their own, separate name. The `arguments` object, on the other hand, contains all arguments sent trough the function call.
- The arguments object is not an array, even though it makes an effort to fool you: it has the length property with which you can get the total number of passed in arguments, and you access the arguments using index notation. This means that you cannot directly use your favorite array methods on it, such as `sort`, `splice`, `indexOf`, and so on. We'll explore these later array methods later, in chapter 10, and we'll also show you how to use array methods on the arguments object. On the other hand, the rest parameter is a true array, and has direct access to all array methods.
- In the next section, we'll continue improving your JavaScript tool-belt with additional ES6 functionality – default parameters.

6.4.3 Default parameters

Sometimes, as you develop your own functions, you end up with a situation that some of the function calls require additional, configurable parameters. In that case, in most other programming languages, people resort to function overloading (specifying additional functions, with the same name, but with a different set of parameters). As you'll see later, function overloading can be a bit tricky in JavaScript, so in the past, when faced with this situation, people have most often resorted to something along the code in the following listing.

Listing 6.10 Tackling default parameters in pre ES6

```
<script>
  function performAction(ninja, action){
     action = typeof action == "undefined" ? "skulking" : action; #1
     return ninja + " " + action;
  }

  assert(performAction("Yagyu","sneaking") === "Yagyu sneaking",    #2
         "The passed in value of the second parameter is used");    #2
  assert(performAction("Fuma") === "Fuma skulking",                 #3
```

```
          "The default value of the second parameter is used");     #3
</script>
```

#1 if the action parameter is undefined, we use a default value "skulking", and if it defined, then we keep the passed in value
#2 As the value of the action parameter we pass a string, that will be the value of that parameter throughout the function body.
#3 We haven't passed in a second argument, the value of the action parameter, after executing the first function body statement will default to "skulking"

In listing 6.10, we define a `performAction` function, that checks if the value of the `action` parameter is `undefined` (by using the `typeof` operator), and if it is then it sets the value of the `action` variable to `"skulking"`. In the case where the `action` parameter was sent through a function call (that is, it is not `undefined`), we simply keep the value.

This is a commonly occurring pattern in JavaScript development and therefore the new ES6 standard, supports *default* parameters, see the following listing.

Listing 6.11 Tackling default parameters in ES6

```
<script>
  function performAction(ninja, action = "skulking"){  #1
     return ninja + " " + action;
  }

  assert(performAction("Yagyu","sneaking") === "Yagyu sneaking",   #2
         "The passed in value of the second parameter is used");   #2
  assert(performAction("Fuma") === "Fuma skulking",                #3
         "The default value of the second parameter is used");     #3
  assert(performAction("Fuma", undefined) === "Fuma skulking",     #4
         "The undefined is passed in, the default value is used"); #4

</script>
```

#1 In ES6, it is possible to assign a value to a function parameter
#2 The passed value is used
#3 If the value is not passed in, the default value is used
#4 If undefined is explicitly passed in, the default value is also used.

Listing 6.11 presents the syntax of default function parameters in JavaScript. To create a default parameter, you simply assign a value to a function parameter #1. Then, when we make a function call and the matching argument value is explicitly defined #2, that value is used in the function body. If it is left out #3, or explicitly set to undefined #4, then the default value is used.

You 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, you can even reference previous default parameters.

6.4.4 Function overloading

When it comes to function overloading—the technique of defining a function that does different things based upon what's passed to it—it's easy to imagine that such a function could be easily implemented by using the mechanisms we've learned so far to inspect the argument list, and to perform different actions in `if-then` and `else-if` clauses. Often, that approach will serve us well, especially if the actions to be taken are on the simpler side.

But once things start getting a bit more complicated, lengthy functions using many such clauses can quickly become unwieldy. In the remainder of this section, we're going to explore a technique by which we can create multiple functions—seemingly with the same name, but each differentiated from the others by the number of arguments they expect—that can be written as distinct and separate anonymous functions rather than as a monolithic `if-then-else-if` block.

All of this hinges on a little-known property of functions that we need to learn about first.

THE FUNCTION'S LENGTH PROPERTY

There's an interesting property on all functions that isn't very well known, but that gives us an insight into how the function was declared: the `length` property. This property, not to be confused with the `length` property of the `arguments` parameter, equates to the number of named parameters with which the function was declared.

Thus, if we declare a function with a single formal parameter, its `length` property will have a value of 1. Examine the following code:

```
function makeNinja(name){}
function makeSamurai(name, rank){}
assert(makeNinja.length == 1, "Only expecting a single argument");
assert(makeSamurai.length == 2, "Two arguments expected");
```

As a result, within a function, we can determine two things about its arguments:

- How many named parameters it was declared with, via the `length` property
- How many arguments were passed on the invocation, via `arguments.length`

Let's see how this property can be used to build a function that we can use to create overloaded functions, differentiated by argument count.

OVERLOADING FUNCTIONS BY ARGUMENT COUNT

There are any number of ways that we can decide to overload what a function does based upon its arguments. One common approach is to perform different operations based upon the type of the passed arguments. Another could be switching based upon whether certain parameters are present or absent. Still another is based upon the *count* of the passed arguments. We'll be looking at this approach in this section.

Suppose we want to have a method on an object that performs different operations based upon argument count. If we want to have long, monolithic functions, we could do something like the following:

```
var ninja = {
```

```
  whatever: function() {
       switch (arguments.length) {
         case 0:
           /* do something */
           break;
         case 1:
           /* do something else */
           break;
         case 2:
           /* do yet something else */
           break;
       //and so on ...
       }
     }
}
```

In this approach, each case would perform a different operation based upon the argument count, obtaining the actual arguments through the `arguments` parameter. But that's not very tidy, and certainly not very ninja, is it?

Let's posit another approach. What if we wanted to add the overloaded method using syntax along the following lines:

```
var ninja = {};
addMethod(ninja,'whatever',function(){ /* do something */ });
addMethod(ninja,'whatever',function(a){ /* do something else */ });
addMethod(ninja,'whatever',function(a,b){ /* yet something else */ });
```

Here we create the object and then add methods to it using the same name (`whatever`), but with separate functions for each overload. Note how each overload has a different number of parameters specified. This way, we actually create a separate anonymous function for each overload. Nice and tidy!

But the `addMethod()` function doesn't exist, so we'll need to create it ourselves. Keep your arms in the cart at all times, as this one's going to be a bit of a short but wild ride.

Take a look at the following listing.

Listing 6.12 A method-overloading function

```
function addMethod(object, name, fn) {
  var old = object[name];                                    #1
  object[name] = function(){                                 #2
    if (fn.length == arguments.length)                       #3
      return fn.apply(this, arguments)                       #3
    else if (typeof old == 'function')                      #4
      return old.apply(this, arguments);                     #4
  };
}
```

#1 Stores the previous function because we may need to call it if the passed function doesn't have a matching number of arguments
#2 Creates a new anonymous function that becomes the method
#3 Invokes the passed function if its parameter and argument counts match
#4 Invokes the previous function if passed function isn't a match

Our `addMethod()` function accepts three arguments:

- An object upon which a method is to be bound
- The name of the property to which the method will be bound
- The function to be bound

Look again at our usage example:

```
var ninja = {};
addMethod(ninja,'whatever',function(){ /* do something */ });
addMethod(ninja,'whatever',function(a){ /* do something else */ });
addMethod(ninja,'whatever',function(a,b){ /* yet something else */ });
```

The first call to `addMethod()` will create a new function that, when called with a zero-length argument list, will call the passed `fn` function. Because `ninja` is a new object at this point, there's no previously established method to worry about.

On the *next* call to `addMethod()`, we store a reference to the function that we created in the previous invocation in the variable `old #1`, and we proceed to create another function that becomes the method #2. This newer method will check to see if the number of passed arguments is `1`, and if so, will invoke the function passed as `fn` #3. Failing that, it will call the function stored in `old` #4, which as you'll recall, will check for zero parameters and call the version of `fn` with zero parameters.

On the third call to `addMethod()`, we pass an `fn` that takes two parameters, and we go through the process again: creating yet another function that becomes the method, and calling the two-parameter `fn` when two arguments are passed, and deferring to the previously created one-argument function.

It's almost as if we're winding the functions around each other like the layers of an onion, each layer checking for a matching number of arguments and deferring to a previously created layer if no match is found. Notice how the inner anonymous function accesses variables `old` and `fn`, through closures.

Let's test our new function in the next listing.

Listing 6.13 Testing the `addMethod()` function

```
<script type="text/javascript">
  var ninjas = {                                                    #1
   values: ["Dean Edwards", "Sam Stephenson", "Alex Russell"]
  };
  addMethod(ninjas, "find", function(){                             #2
    return this.values;
  });
  addMethod(ninjas, "find", function(name){                         #3
    var ret = [];
    for (var i = 0; i < this.values.length; i++)
      if (this.values[i].indexOf(name) == 0)
        ret.push(this.values[i]);
    return ret;
  });
  addMethod(ninjas, "find", function(first, last){                  #4
    var ret = [];
    for (var i = 0; i < this.values.length; i++)
```

```
      if (this.values[i] == (first + " " + last))
        ret.push(this.values[i]);
    return ret;
  });
  assert(ninjas.find().length == 3,                          #5
         "Found all ninjas");
  assert(ninjas.find("Sam").length == 1,
         "Found ninja by first name");
  assert(ninjas.find("Dean", "Edwards").length == 1,
         "Found ninja by first and last name");
  assert(ninjas.find("Alex", "Russell", "Jr") == null,
         "Found nothing");
</script>
```

#1 Declares an object to serve as the base, preloaded with some test data
#2 Binds a no-argument method to the base object
#3 Binds a single-argument method to the base object
#4 Binds a dual-argument method to the base object
#5 Tests the bound methods

Loading this page to run the tests shows that they all succeed, as shown in figure 6.7.

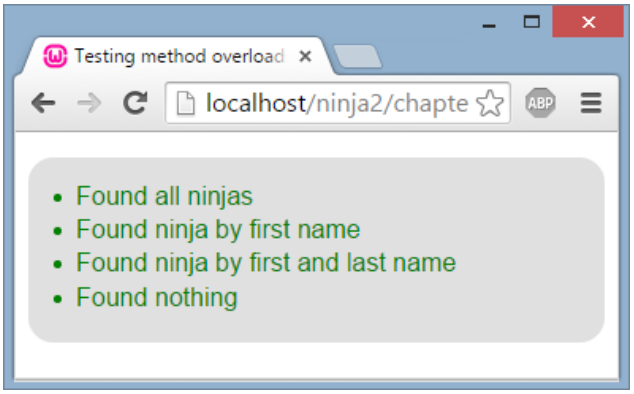


Figure 6.7 Ninjas found, all using the same overloaded method name `find()`

To test our method-overloading function, we define a base object containing some test data consisting of well-known JavaScript ninjas #1, to which we'll bind three methods, all with the name `find`. The purpose of all these methods will be to find a ninja based upon criteria passed to the methods.

We declare and bind three versions of a `find()` method:

- One expecting no arguments that returns all ninjas #2
- One that expects a single argument and that returns any ninjas whose name starts with the passed text #3
- One that expects two arguments and that returns any ninjas whose first and last names match the passed strings #4

This technique is especially nifty because these bound functions aren't actually stored in any typical data structure. Rather, they're all saved as references within closures.

It should be noted that there are some caveats to be aware of when using this particular technique:

- The overloading only works for different numbers of arguments; it doesn't differentiate based on type, argument name, or anything else. Which is frequently exactly what we'll want to do.
- Such overloaded methods will have some function call overhead. We'll want to take that into consideration in high-performance situations.

Nonetheless, this function provides a good example of some functional techniques, as well as an opportunity to introduce the `length` property of functions.

6.5 Summary

In this chapter, you learned:

- Advanced usages of function closures, how to:
- Mimic private object variables, by closing over constructor variables through method closures.
  - Deal with callbacks, in a way that significantly simplifies your code.
- By binding a function to a particular object, we can make sure that our function is operating on the targeted context, thereby evading some surprisingly common bugs.
- Functions can have properties and those properties can be used to store any information we might wish to use, including
- Storing functions in function properties for later reference and invocation.
  - Using function properties to create a cache (memoization).
- Functions can perform different operations based upon the arguments that are passed to it (function overloading). We can inspect the `arguments` list to determine what it is we'd like to do given the type or number of the passed arguments.

If you take a look back at Chapter 4, you'll see that in JavaScript there are different types of functions, and that the new ES6 standard specifies a new type of functions, *generator* functions which are quite different than your normal run-of-the-mill functions. In the next chapter, we'll take a closer look at what makes generator functions special, and we'll closely examine a related new concept, *Promises* that will help you deal with asynchronous code.

7

Generator functions and promises

This chapter covers

- Pausing and resuming the execution of generator functions
- Exchanging data with generator functions
- Handling asynchronous tasks with promises
- Working with multiple promises
- Achieving elegant asynchronous code by combining generator functions and promises

Imagine that you're a developer working at www.freelanceninja.com, a popular freelance ninja recruitment site where customers can hire ninjas for stealth missions. Your task is to implement a functionality that will enable the users to get the 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. 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. Unfortunately, this code has a great problem. Getting data from the server is a long-running operation, and since 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 usually 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
    });
  });
});
```

While this code will be much better received by our users, you'll probably agree that it's not the easiest code to write, the error handling is messy, and it is plain ugly.

In this chapter, we'll go deep into two new ES6 concepts: *generator functions* and *promises*, which will help you turn the non-blocking, but awkward callback code into something much more elegant:

```
async(function*() #A
{
  try {
    var ninjas = yield getJSON("ninjas.json");
    var missions = yield getJSON(ninjas[0].missionsUrl);
    var missionDescription = yield getJSON(missions[0].detailsUrl);
    //Study the mission details
  }
  catch(e) {
    //Oh no, we weren't able to get the mission details
  }
});
```

#A A generator function is defined by putting a star right after the `function` keyword. You can use the new `yield` keyword in generator functions
#B The promises are hidden within the getJSON method, more on that later

Don't worry if you don't recognize some of the constructs from the example. That's what this chapter is for. We are going to devote it to fully understanding all the mechanisms that enable you to write code that combines the ease of understanding of synchronous code with the non-blocking aspects of asynchronous code. To do this, we'll need *generators* and *promises*. For now, it's enough to know that *generators* are a special type of functions whose execution can be paused and resumed, and that *promises* are objects that represent values which will be known at a later point in time.

Let's start with *generators*.

7.1 Generator functions

The ES6 version of JavaScript introduced a completely new type of function – *generator functions*, which are significantly different than your run-of-the-mill functions. Simply put, a *generator function is a function that generates a sequence of values, but not all at once, like a standard function would, but on a per-request basis. You have to explicitly ask the generator for a new value, and the generator will either respond with the value, or it will notify you that it has no more values to give.* Here's a really simple code example:

Listing 7.1 A very simple generator function

```
function* NinjaGenerator(){  #A
  yield "Hatori";                #B
     yield "Yoshi";              #B
}

var ninjaIterator = NinjaGenerator();  #C

var result1 = ninjaIterator.next();    #D
assert(typeof result1 == "object"           #E
    && result1.value === "Hatori"  #E
    && result1.done === false,   #E
    "The first result received!");    #E

var result2 = ninjaIterator.next();    #F
assert(typeof result2 == "object"       #F
    && result2.value === "Yoshi"   #F
    && result2.done === false,   #F
    "The second result received!");    #F

var result3 = ninjaIterator.next();   #G
assert(typeof result3 == "object"         #G
    && result3.value === undefined  #G
    && result3.done,    #G
    "There are no more results!");    #G
```

#A – Define a generator function. Notice the star * after the function keyword
#B – Use the `yield` keyword to return a value from the generator. You can yield as many times as you want
#C – Calling a generator function doesn't execute it. Instead an iterator object through which we control the execution of a generator is created
#D – Use the iterator's `next` method to request a value from the generator
#E – The result is an object which contains the returned value and an indicator that tells us whether the generor is done
#F – By calling `next` again, we get another value from the generator
#G – Once there's no more code to execute, the generator returns the value undefined and idicates that it is done.

As listing 7.1 shows, defining a generator function is simple, you just add '`*`', a star after the `function` keyword. This enables you to use the new `yield` keyword within the generator body that returns intermediary results. In our case, we have two such return values: `"Hatori"` and `"Yoshi"`.

Unlike with standard functions, calling a generator doesn't really execute its body. Instead, a new object, called an *iterator* object, is created.

```
var ninjaIterator = NinjaGenerator();
```

The iterator is used to control the execution of the generator, and calling the `next` method on the iterator requests the next value from the generator.

```
var result1 = ninjaIterator.next();
```

As a response to that call, the generator executes its code until it has an intermediary result (or until there's no more code to execute), and returns a *new* object that encapsulates that result and tells us whether it's work is now done. In our case, the first call to the iterator's `next` method executes the generator code up to the first yield expression:

```
yield "Hatori";
```

and returns an object with the value `"Hatori"`, and `done` set to `false`, signaling that there's more work to do. As soon as the value is produced, the generator pauses its execution, without blocking, and patiently waits. This is an incredibly powerful feature that normal functions don't have, a feature that will utilize later, to great effect.

Later, when another call to the `next` method is made:

```
var result2 = ninjaIterator.next();
```

the generator function continues where it has last paused, and again, executes until either it runs out of code, or until there's another intermediary value. In our case, the generator reaches: `yield "Yoshi"`, so an object carrying `"Yoshi"` will be returned, and the generator will again be paused.

Finally, when we call the `next` method for the third time, the generator again continues its execution from where it has last stopped. However, now there's no more code to execute, and the generator returns an object with a `value` set to `undefined`, and `done` to `true`, signaling that it's done with its work.

THE LIFECYCLE OF A GENERATOR

As you might've guessed by this description, a generator function acts like its own little program that based on some trigger (calls to the matching iterator's `next` method) moves between different states:

- *Suspended start* – When the generator is created, it starts in this state. None of the generator's code is actually executed.
- *Executing* – the state in which the actual code of the generator is executed. The execution continues either from beginning or from where it has last paused. 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 the *completed* state.
- The following figure describes the example from listing 7.1 as the movement through different generator states.

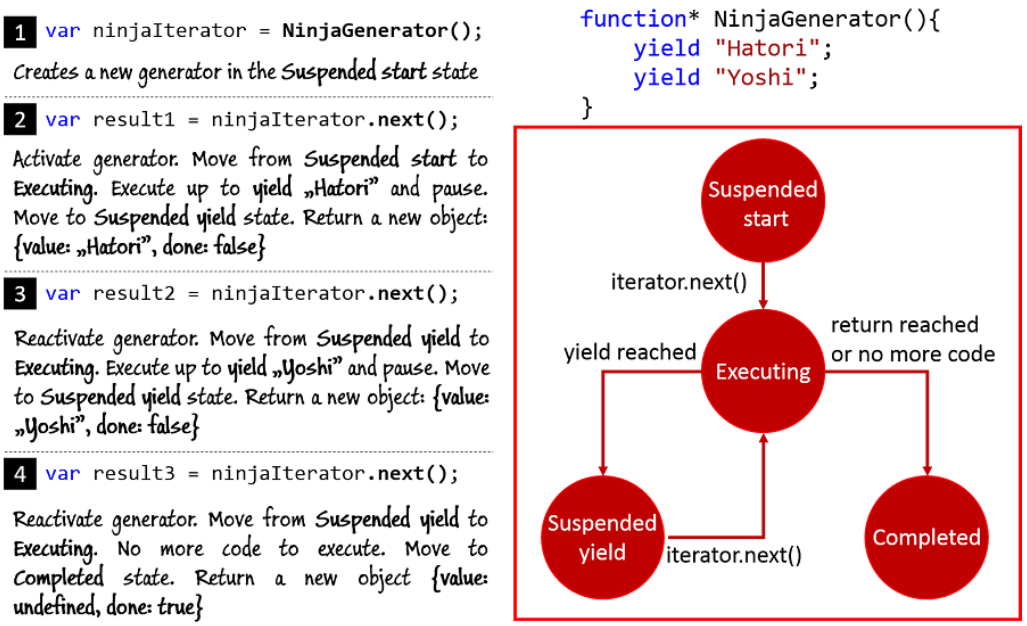


Figure 7.1 During execution, the generator moves between different states triggered by the calls to the matching iterator's `next` method.

Another, a bit more simpler way of looking at generators is to simply view them as producers of sequences of values, meaning that, among other things it should make sense that we could loop over those values. Let's see how to do this.

7.1.1 Iterating over a generator

So far, we now that each call to the iterator's `next` method produces an object with properties `value` and `done`, and when there are no more results, the returned object will have its `done` property set to `true`. Now by using this knowledge, let's implement a loop through the generated values. See the following listing:

Listing 7.2 Iterating over generator results with a while loop

```
function* NinjaGenerator(){
  yield "Hatori";
  yield "Yoshi";
  yield "Hatano";
}
var iterator = NinjaGenerator();
var res;
while(!(res = iterator.next()).done) {
  assert(true, res.value)
}
```

If you execute this code, you'll see that it outputs: `"Hatori", "Yoshi", "Hatano"`. We simply use a while loop to pull out and print values out of a generator, and we do this until the generator notifies us that its work is done. Since one of the fundamental applications for iterators is, well… iterating over certain collections of items, it feels clunky to have to resort to this rather ugly `while` loop. For this reason, ES6 introduced a special kind of for loop, the `for...of` loop that enables you to easily iterate over generator produced values. Take a look at the following listing.

Listing 7.3 Iterating over a generator by using the for…of loop

```
function* NinjaGenerator(){
  yield "Hatori";
     yield "Yoshi";
  yield "Hatano";
}

for(var item of NinjaGenerator()){  #A
  assert(true, item);
}                                   #A
```

#A – the for…of loop enables us to iterate over generator values

The for…of loop is really just syntactic sugar that helps us deal with iterators, and from a functional perspective, is very similar to the foreach loop from C# or Java.

> **TIP** In ES6, a lot of data structures, old ones such as array or strings, and new ones such as maps and sets can be iterated over. This means that you can also use the for…of loop on them. For more information, visit: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Iteration_protocols

7.1.2 Understanding generator functions

Now that you have achieved a certain comfort level with generators, let's take a deeper look at how they really work, by overanalyzing the following, very simple code fragment:

```
function* NinjaGenerator(action) {
  yield "Hatori " + action;
  return "Yoshi " + action;
}
```

```
var ninjaIterator = NinjaGenerator("skulk");
var result1 = ninjaIterator.next();
var result2 = ninjaIterator.next();
```

In this example, we define a generator function that, in this case, generates two values `"Hatori skulk"` and `"Yoshi skulk"`.

Now, we'll explore the state of the application, the execution context stack at various points in the application execution. We'll start with the following figure.

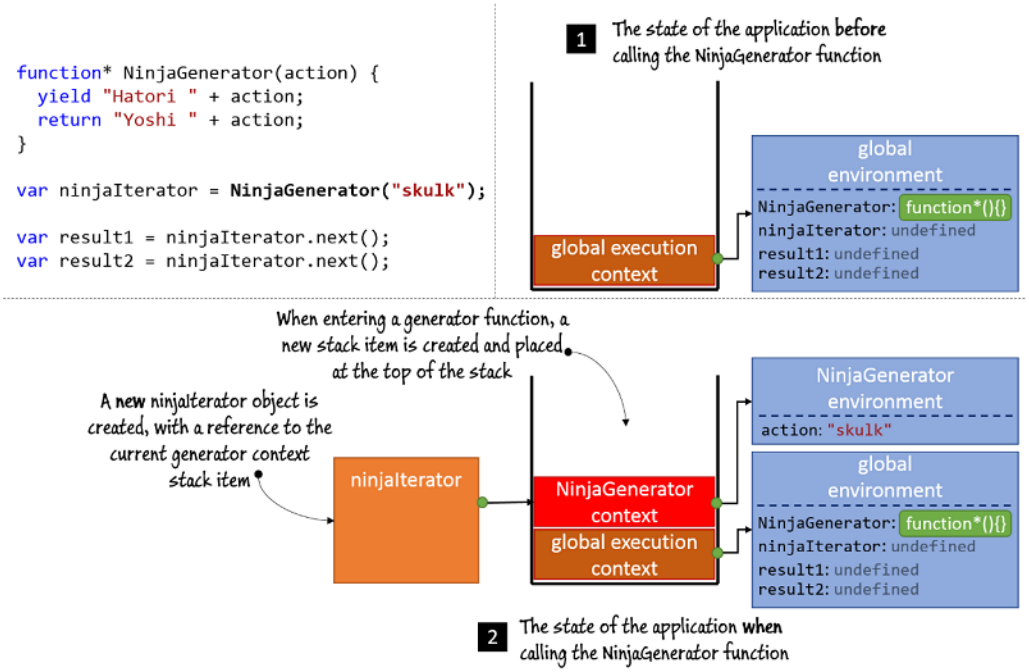


Figure 7.2 The state of the execution context stack **before** calling the `NinjaGenerator` function (1), and **when** calling the `NinjaGenerator` function (2).

Figure 7.2 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 (1). Since we are executing global application code, in the execution context stack there's only the global execution context that references the global environment in which our identifiers are kept. Currently, only the `NinjaGenerator` identifier references a function, while the values of all other identifiers are `undefined`.

When we call the `NinjaGenerator` function (2):

```
var ninjaIterator = NinjaGenerator("skulk");
```

the control-flow enters the generator function, and, as it happens when we enter any other function, a new `NinjaGenerator` execution context stack item is created and pushed onto the stack. However, since generator functions are special, **none** of the function code is currently executed. Instead, a new iterator object, `ninjaIterator`, is created and returned. Since the iterator is used to control the execution of the generator function, the iterator gets a reference to the execution context in which it was created.

A really interesting thing happens when the control-flow of the application leaves the generator function, as shown in figure 7.2.

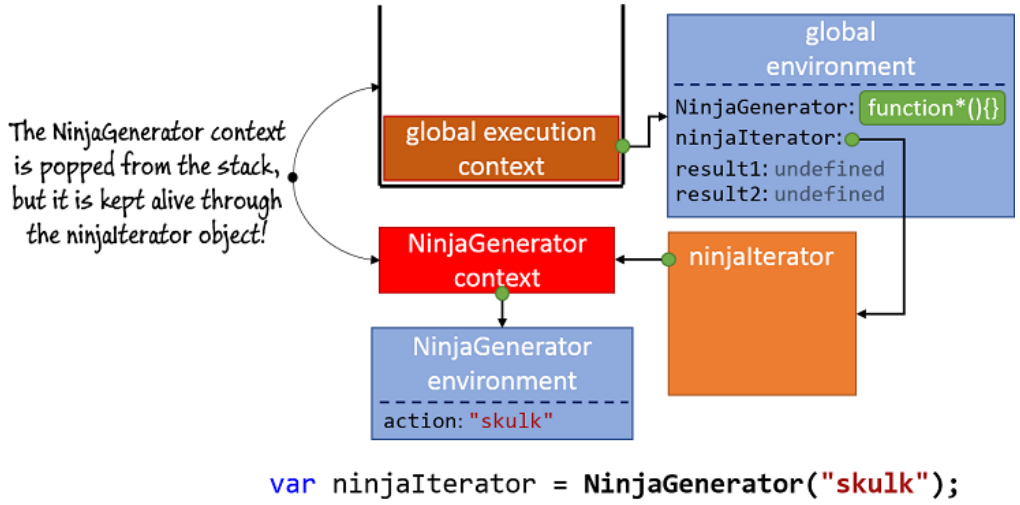


Figure 7.3 The state of the application when returning from the `NinjaGenerator` call

Figure 7.3 shows the state of the application when we return from the call to the `NinjaGenerator` function. Normally, when we return from a standard function, the matching execution context stack item is popped from the stack and completely discarded. However, this is not the case with generators. The matching `NinjaGenerator` stack item *is* still popped from the stack, but it is *not* discarded, since the iterator object keeps a reference to it. You can see it as an analog to closures. In closures we need to keep alive the variables that are alive at the moment when the function closure is created, so our functions keep a reference to the environment in which they were created, in that way making 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. Since the generator execution is handled by the execution context, the iterator object keeps a reference to it, so that is alive for as long as the iterator needs it.

Another interesting thing happens when we call the `next` method on the iterator:

```
var result1 = ninjaIterator.next();
```

If this was your normal, straight-forward function, this would cause the creation of a *new* `next()` execution context stack item which would be placed on the execution context stack. However, as you might've noticed so far, generator functions are anything but standard, and the call to the `next()` method of an iterator object behaves differently. It reactivates the matching execution context stack item, the `NinjaGenerator` context, and places it on top of the stack, continuing the execution where it has left off, see the following figure.

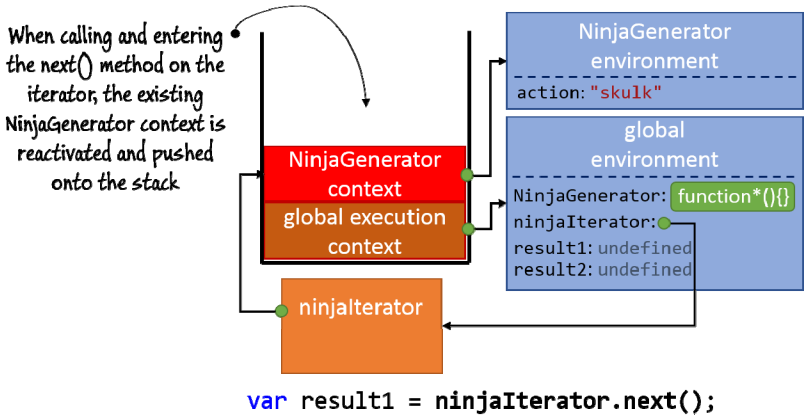


Figure 7.4 Calling the iterator's `next` method reactivates the execution context stack item of the matching generator, pushes it on the stack, and continues where it has left off the last time.

Figure 7.4 illustrates a crucial difference between standard functions and generator functions. Standard functions can only be called anew, and each call creates a *new* execution context stack item. In contrast, the execution context stack item of a generator function can be temporarily paused and resumed, at will.

In our example, since this is the first call to the `next` method, and the generator function hasn't really started executing yet, this simply means that the generator function starts its execution.

The next interesting thing happens when our generator function reaches:

```
yield "Hatori " + action
```

The generator determines that the expression equals to `"Hatori skulk"` and the evaluation reaches the `yield` keyword. This means that `"Hatori skulk"` is the first intermediary result of our generator function and that we want to pause the execution of the generator and return that value. In terms of the application state, a similar thing happens as before, the `NinjaGenerator` context is taken off the stack, but it's not completely discarded since the `ninjaIterator` keeps a reference to it – the execution state of the generator function is suspended, without blocking, and the program execution resumes in the global code.

Back in the global code, we simply store the yielded value to `result1`, and continue execution. Immediately, we reach another iterator call

```
var result2 = ninjaIterator.next();
```

and we go through the whole procedure once again: we reactivate the `NinjaGenerator` context referenced by the `ninjaIterator`, push it onto the stack, and continue the execution where we last left off. In this case, the generator function evaluates the expression:

```
"Yoshi " + action
```

and the program execution continues similarly to the end.

Now that you have a deeper understanding on how generators work under the hood, let see some additional generator concepts, starting with how to send data to a generator.

7.1.3 Sending data to a generator

In the examples presented so far, you've seen that you can return multiple values *from* the generator by using the `yield` expression. But generators are even more powerful than that, and you can also send data into the generator, thereby achieving two-way communication. This essentially means that, with a generator, you can produce some intermediary result, use that result to calculate something else, and then, whenever you're ready, send some completely new data to the generator and resume its execution.

Check out the following example:

Listing 7.4 Sending data to and receiving data from a generator

```
function* NinjaGenerator(action) {                      #A
  try {
    var imposter = yield ("Hatori " + action); #B

    assert(imposter === "Hanzo",                        #C
           "The generator has been infiltrated");

    yield ("Yoshi (" + imposter + ") " + action);

    assert(false, "Should not come here!");
  }
  catch(e){
    assert(true, "We've sent an exception!");   #D
  }
}

var ninjaIterator = NinjaGenerator("skulk"); #E

var result1 = ninjaIterator.next();
assert(result1.value == "Hatori skulk", "Hatori is skulking");

var result2 = ninjaIterator.next("Hanzo");         #F
assert(result2.value === "Yoshi (Hanzo) skulk",
       "We have an imposter!");                                  #G

ninjaIterator.throw("Error");           #H
```

#A – You can send function arguments as with standard functions
#B – This is where the magic happens, by yielding a value, the generator returns an intermediary calculation, by calling the iterator's `next` method with an argument, you can send data to the generator
#C – The value sent over `next` becomes the value of the yielded expression, so our imposter is Hanzo
#D – Check if we got an exception
#E – Normal argument passing
#F – Sending an argument through `next`
#G – Check that the `next` argument was successfully processed
#H – We can cause exceptions from the outside of the generator function!

The easiest way to send data to a generator is by treating it like any other function and using function call arguments, like we did when making the `NinjaGenerator("skulk")` call. This is nothing special, your plain old functions do it all the time. But remember, generators have this amazing power that they can be paused and resumed at will, and they can even receive data after their execution has started.

USING THE NEXT METHOD

Another way of sending data, specific to generator functions, is to use the `next` method of a matching iterator. You've already seen that you can use the `next` method to continue the execution of a generator, but we didn't tell you that, at the same time, you can pass in a value that the generator will use for the value of the `yield` expression, where the generator was suspended. In our example, the call:

```
ninjaIterator.next("Hanzo")
```

resumes the execution of the generator that was patiently waiting after yielding control on:

```
yield "Hatori " + action.
```

This means that the value of the whole `yield` expression, and subsequently, the value of the `imposter` variable will be set to `"Hanzo"`.

THE FIRST CALL TO THE NEXT METHOD

Notice how we said that the `next` method supplies the value to the waiting `yield` expression. This means that if there is no `yield` expression waiting, there is nothing to supply the value to. For this reason, you *cannot* supply values over the first call to the `next` method. At that moment, the generator function hasn't yet started its execution, and there is no waiting `yield` expression. But remember, if you need to supply something to the generator, you can also do it when calling the generator function itself, like we did with `NinjaGenerator("skulk")`.

THROWING EXCEPTIONS

There's also another way, a slightly less orthodox way, to supply a value to a generator, by throwing an exception. Each iterator object, in addition to having a `next` method, also has a `throw` method with which we can throw an exception to the generator. In our example, calling:

```
ninjaIterator.throw("Error")
```

throws an exception that can be caught in the generator's body. This might be a bit strange at first, but later in the chapter we'll show you an interesting usage of this generator feature.

7.1.4 Delegating generators

In certain cases, just like we often call other one standard function from another standard function, so too we want to be able to delegate the execution of one generator to another, for example:

Listing 7.5 Using yield* to delegate to another generator

```
function* WarriorGenerator(){
  yield "El Cid";
  yield* NinjaGenerator(); #A
  yield "Genghis Khan";
}

function* NinjaGenerator(){
  yield "Hatori";
  yield "Yoshi";
}

for(var warrior of WarriorGenerator()){
  assert(true, warrior);
}
```

#A - Use `yield*` to delegate to another generator

If you run the code, you'll see that the output is: `"El Cid"`, `"Hatori"`, `"Yoshi"`, `"Genghis Khan"`. Generating `"El Cid"` will probably not catch you off guard, it's simply the first value yielded by the `WarriorGenerator`. However, the second output: `"Hatori"`, deserves some explanation. By using the `yield*` operator on an iterator object, we basically yield to another generator. In our case, from a `WarriorGenerator` we are yielding to a new `NinjaGenerator`. This effectively means that all calls to the current `WarriorGenerator` iterator's `next` method will be rerouted to the `NinjaGenerator`. This will hold until the `NinjaGenerator` is done with its work. So in our example, after `"El Cid"`, the program will generate `"Hatori"` and `"Yoshi"`. Only when the `NinjaGenerator` has done its work, the execution of the original iterator will continue by outputting `"Genghis Khan"`. Notice how this is happening entirely transparent to the code that calls the original generator, the `for…of` loop doesn't care that the `WarriorGenerator` yields to the `NinjaGenerator`, it just keeps calling `next` until it's done.

Now that you have a grasp on how delegating to another generator works, let's see its usage on a practical example of traversing the DOM.

DOM TRAVERSAL WITH A GENERATOR

As you've seen in chapter 2, the layout of a web page is based on the DOM, a tree structure of HTML nodes, in which every node, except the root one, has exactly one parent, and can have zero or more children. Since 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 7.6 Recursive DOM traversal

```
<div id="subTree">
  <form>
    <input type="text"/>
  </form>
  <p>Paragraph</p>
  <span>Span</span>
</div>
<script>
  function traverseDOM(element, callback) {
    callback(element);                          #A
    element = element.firstElementChild;
    while (element) {                           #B
      traverseDOM(element, callback);
      element = element.nextElementSibling;
    }                                       #B
  }
  var subTree = document.getElementById("subTree");
  traverseDOM(subTree, function(element) {                    #C
    assert(true, element.nodeName);
  });                                           #C
</script>
```

#A Process the current node with the callback
#B Traverse the DOM of each child element.
#C Start the whole process by calling the `traverseDOM` function for our root element.

In the example from listing 7.6, 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 our 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 a bit differently, see the following code.

Listing 7.7 Iterating over a DOM tree with generators

```
function* DomTraversal(element){
  yield element;
  element = element.firstElementChild;
  while (element) {
    yield* DomTraversal(element); #A
    element = element.nextElementSibling;
  }
}

var subTree = document.getElementById("subTree");
for(var element of DomTraversal(subTree)) {  #B
  assert(true, element.nodeName);
}                                       #B
```

#A – `yield*` keyword is used to yield the iteration control to another instance of the `DomTraversal` generator.
#B – Iterate over all elements in the node.

Listing 7.7 shows that we can achieve DOM traversals with generators, just as easily as with standard recursion, but with an aditional benefit of not having to use the slightly ackward 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 is conceptually recurisive code, in iterabile fashion; we can even consume it with a for…of loop. Using iterations is sometimes more natural than recursion, so it's always good to have your options open.

We've explored generators quite a bit, but now we'll move onto the second key ingredient required for writing elegant asynchronous code – promises.

7.2 Promises

In JavaScript, we rely a lot on asynchronous computations, computations whose results we don't have yet, but will at some later point in time. So ES6 has introduced a new concept that makes dealing with asynchronous tasks easier – *Promises*. In short, a promise is a placeholder for a value that we yet don't have, but that we will have, at some later point in time; it's a guarantee that we'll eventually know the result of some asynchronous computation. If we make good on our promise, that result will be some value, and if a problem occurs, it will be an error, an excuse on why we couldn't deliver. Creating a new promise is easy, just take a look at the following example.

Listing 7.8 Creating a simple promise

```
var ninjaPromise = new Promise(function(resolve, reject){       #A
  resolve("Hatori");  #B
  //reject("An error resolving a promise!");    #B
});

ninjaPromise.then(function(ninja) {                    #C
  assert(ninja === "Hatori", "We were promised Hatori!");
}, function(err){                                              #D
  assert(false, "There shouldn't be an error")
});
```

#A – a promise is constructed by calling a built-in Promise constructor and passing it a function with two parameters: resolve and reject
#B – a promise is successfully resolved by calling the passed-in resolve function (and rejected by calling the reject function)
#C – By using the then() method on a promise, we can pass in two callbacks, the first will be called if a promise is successfully resolved
#D – and the second will be called if an error occurs

As listing 7.8 shows, to create a promise, you just use the built-in `Promise` constructor to which you pass a function, the so called *executor* function, which has two parameters: `resolve` and `reject`. The executor function will be called when constructing the Promise

object with two built-in functions: `resolve`, which you have to manually call if you want the promise to resolve successfully, and `reject` that you call if an error occurs.

Our code then, makes use of the promise by calling the built-in `then` method on a `Promise` object, a method that takes in two callback functions: a *success* callback and a *failure* callback. The former will be called if the promise is resolved successfully, if the `resolve` function was called on the promise; and the latter if there was a problem, either an unhandled exception has occurred or the `reject` function was called on a promise.

In our example code, we create a promise and immediately resolve it by calling the `resolve` function with the argument `"Hatori"`. This means that, when we call the `then` method, the first, success callback will be executed and the test that outputs: `"We were promised Hatori!"` will pass.

Now that you have a general idea on what promises really are and how they work, let's take a step back and see some of the problems that promises tackle.

7.2.1 The problems with simple callbacks

We deal with asynchronous code because we don't want to block the execution of our application (thereby disappointing our users) while long-running tasks are executed. Currently, we solve this problem with callbacks: to a long-running task we provide a function, a callback, that will be called when the task is finally done.

For example, fetching a JSON file from the 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 called when the task is done:

```
getJSON("ninjas.json", function(err, ninjas) {
  /*Handle results*/
});
```

Naturally, during this long-running tasks, errors can happen. And the problem with callbacks is that you cannot really use the built-in language constructs, such as the try-catch statements, in the following way:

```
try {
  getJSON("ninjas.json", function(err, ninjas) {
  //Handle results
  });
} catch(e) {/*Handle errors*/}
```

The reason being that the code calling the callback is usually not executed in the same step of the event loop as the code that starts the long-running task. This means that errors usually get lost. For this reason, a lot of libraries define their own conventions for reporting errors. For example, in the node.js world, it is a custom that callback functions take two arguments `err` and `data`, where `err` will be a non-null value if an error occurs somewhere on the way. This leads us to the first problem with callbacks: *Difficult error handling*.

Often, once we've performed a long-running task, we 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; in essence leading to a series of interdependent, asynchronous, callback processed steps. For example, if we wanted to execute a sneaky plan in which we want to find all ninjas at our disposal, get the location of the first ninja, and send him some orders, we would end up with something like this:

```
getJSON("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've noticed how this code is difficult to understand, it is a pain to insert new steps, 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 the final result, don't depend on each other, so we don't have to make them in sequence, instead, in order to save precious milliseconds, we can do them in parallel. For example, if we wanted to set up a plan in motion that requires us to know which ninjas we have at our disposal, what's the plan, and the location in which our plan will play out, we would 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 the plan in parallel, since these actions don't really depend on each other. We simply care that, in the end, we have all the data at our disposal. Since we don't really know the order in which our data is received, every time we get some data, we have to check if 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 how you have to write a lot of boiler-plate 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've shown you how with callbacks you cannot really use some of the fundamental language constructs, like try-catch statements. A similar thing holds with loops also – if you want to perform 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 this 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 that will be available in all browsers.

Now that you understand most of the reasons behind creating promises, as well as having a basic understanding of them, let' take it up a notch.

7.2.2 Diving into promises

A promise is an object that servers as a placeholder for a result of an asynchronous task. It represents a value that we currently don't have, but that we hope to have sometime in the future. For this reason, during its lifetime, a promise can go through a couple of different states, as shown in the following figure.

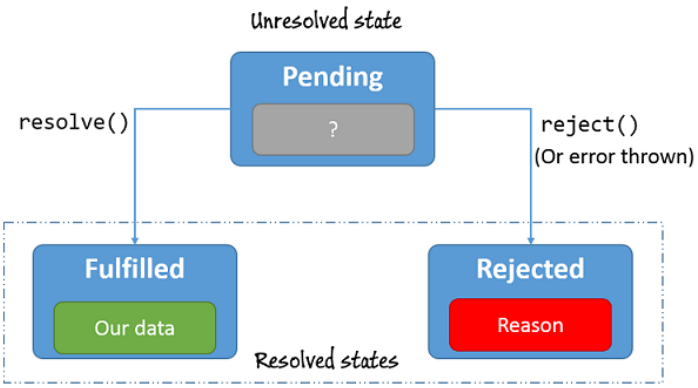


Figure 7.5 Different states of a promise.

Figure 7.5 shows different states of a promise. 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 promise moves onto 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 onto the *rejected* state, in which we weren't able to obtain the promised value, but in which we at least know why we weren't able to deliver on our promise. Once a promise has reached either the *fulfilled* state or the *rejected* state, it cannot switch (go from fulfilled to rejected or vice versa), and it always stays in that state. A promise is *resolved* (either successfully or not).

Now we'll take a closer look at what's exactly going on when you use promises. For this, we'll take a look at the following code example.

Listing 7.9 A closer look at promise order of execution

```
assert(true, "At code start");

var ninjaDelayedPromise = new Promise(function(resolve, reject){
  assert(true, "ninjaDelayedPromise executor");                  #A
  setTimeout(function(){                                                #B
    assert(true, "Resolving ninjaDelayedPromise");               #B
    resolve("Hatori");    #B
  }, 500);                                                              #B
});

assert(ninjaDelayedPromise, "After creating ninjaDelayedPromise");

ninjaDelayedPromise.then(function(ninja) {                              #C
  assert(ninja === "Hatori",    #C
        "ninjaDelayedPromise resolve handled with Hatori");  #C
});                                                             #C

var ninjaImmediatePromise = new Promise(function(resolve, reject){   #D
```

```
  assert(true, "ninjaImmediatePromise executor. Immediate resolve."); #D
     resolve("Yoshi");    #D
});                       #D

ninjaImmediatePromise.then(function(ninja) {     #E
  assert(ninja === "Yoshi",   #E
          "ninjaImmediatePromise resolve handled with Yoshi");  #E
});      #E

assert(true, "At code end");
```

#A – Calling the promise constructor immediately invokes the passed in function
#B – We'll resolve this promise as successful after a 500ms timeout expires
#C – 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
#D – Create a new promise that gets immediately resolved
#E – Sets up a callback to be called when the promise resolves. But, our promise is already resolved!?

The code in listing 7.9 outputs the results shown in the following figure.

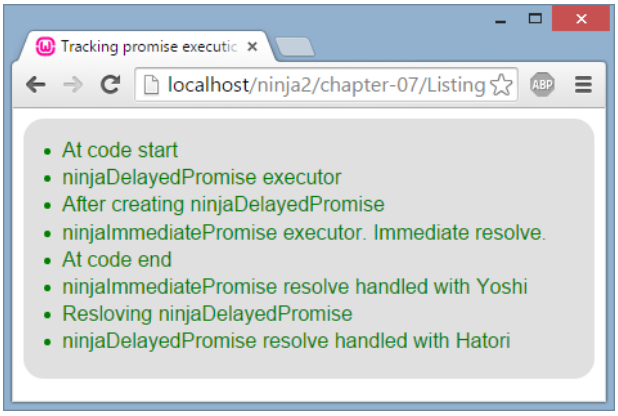


Figure 7.6 The result of executing listing 7.9

As you can see in figure 7.6, when we call the `Promise` constructor, the passed in executor function is immediately invoked. During the construction of our first promise, the `ninjaDelayedPromise`, we set up a timeout:

```
setTimeout(function(){
    assert(true, "Resolving ninjaDelayedPromise");
    resolve("Hatori");
  }, 500);
```

that will resolve the promise after 500 ms. This could've been any other asynchronous action, but we chose the humble timeout, due to 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` will be 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:

```
ninjaDelayedPromise.then(function(ninja) {
  assert(ninja === "Hatori",
         "ninjaDelayedPromise resolve handled with Hatori");
});
```

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. This means that, unlike the `ninjaDelayedPromise`, which after construction is in *pending* state, the `ninjaImmediatePromise` finishes construction in the *resolved* state – the promise already has the value `"Yoshi"`.

Afterwards, we use the `ninjaImmediatePromise 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 simply be ignored? The answer is: *neither*.

Promises are designed to deal with asynchronous actions, so the JavaScript engine *always* resorts to asynchronous handling, simply to make the promise behavior predictable. The engine does this by executing the `then` callbacks after all of the code in the current step of the event loop is executed. For this reason, if you study the output in figure 7.5, you'll see that we first log "At code end" and then we log that the `ninjaImmediatePromise` was resolved.

In the end, after our 500 ms timeout expires, we resolve the `ninjaDelayedPromise`, which cause the execution of its `then` callback.

In this example, for the sake of simplicity, we've only dealt with the rosy scenario in which everything goes great. But, in the real-world, it's not only sunshine and rainbows, so now we'll see how to deal with all sorts of crazy problems that can occur.

7.2.3 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. See the following listing:

Listing 7.10 Rejecting promises

```
var firstPromise = new Promise(function(resolve, reject){
  reject("Explicitly reject a promise!");              //#A
});
firstPromise.then(
  function() { assert(false, "Happy path, won't be called!"); },
  function(error) {                                          //#B
    assert(true, "First promise threw an explicit reject!");
```

```
  }                                                                    //#B
);

var secondPromise = new Promise(function(resolve, reject){
  undeclaredVariable++; //#C
});
secondPromise.then(
  function() { assert(false, "Happy path, won't be called!"); },
  function(error) {
    assert(true, "Second promise threw an implicit reject!");
  }
);

var thirdPromise = new Promise(function(resolve, reject){
  reject("Error!");
});
thirdPromise.then(function(){
  assert(false, "Happy path, won't be called!")
}).catch(function(){                                  //#D
  assert(true, "Third promise was also rejected");
});                                    //#D
```

#A – A promise can be explicitly rejected by calling the passed-in reject function
#B – If a promise is rejected, the second, error callback is invoked
#C – A promise is implicitly rejected if an unhandled exception occurs when processing the promise
#D – 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.

Listing 7.10 shows different ways of rejecting a promise. When processing the `firstPromise`, we explicitly reject the promise by calling the passed in `reject` method. This means that when using the `then` method, the second, error callback will be invoked. A similar thing happens, if an unhandled exception occurs, for example, in our case, an exception occurs when we try to increment the `undeclaredVariable`. Again, the error callback is invoked. This is really handy, regardless of how the promise was rejected, all errors are directed to our rejection callback function, making our lives easier.

You can also use an alternative error-handling syntax, by chaining a `catch` method call after the `then` method call. This gives the code a try-catchy look that some people find more natural. This method of error handling has another purpose that we'll show you later on.

Now that you understand how promises work, how to schedule success and failure callbacks, let's take a real-world scenario, getting JSON data from a server, and promisify it.

7.2.4 Creating your first real-world promise

One of the most common asynchronous actions on the client is fetching some data from the server. As such, this is an excellent little case-study on the usage of promises. For the underlying implementation we'll use the built-in `XMLHttpRequest` object.

Listing 7.11 Creating a getJSON promise

```
function getJSON(url) {
  return new Promise(function (resolve, reject) { #A
    var request = new XMLHttpRequest();  #B
```

```
    request.open("GET", url);          #C

    request.onload = function () {             #D
      try {
        if(this.status == 200 ){                          #E
          resolve(JSON.parse(request.response));          #F
        }
           else{
             reject(this.status + " " + this.statusText); #G
        }
      }
      catch(e){
        reject(e.message);                                      #G
      }
    };

    request.onerror = function () {                             #H
      reject(this.status + " " + this.statusText);
    };                                            #H

    request.send();                                         #I
  });
}

getJSON("ninjas.json").then(function(ninjas){           #J
  assert(ninjas, "Ninjas obtained!");
}).catch(function(e){
  assert(false, "Shouldn't be here:" + e);
});                                                     #J
```

#A – Create and return a new promise
#B – Creates an `XMLHttpRequest` object
#C – Initializes the request
#D – Registers an `onload` handler which will be called if the server has responded
#E – Even if the server has responded, it doesn't mean that everything went as we expected. Use the result if the server responds with status 200 (everything OK)
#F – Try to parse the JSON string, if it succeeds, resolve the promise as successful with the parsed object
#G – If the server responds with a different status code, or if there's an exception parsing the JSON string, reject the promise
#H – If there's an error while communicating with the server, reject the promise
#I – Send the request
#J – Use the promise created by the `getJSON` function to register a resolve and reject callbacks

Our goal, in listing 7.11., is to create a `getJSON` function that returns a promise that will enable us to register success and failure callbacks for asynchronously getting `JSON` objects from the server. For the underlying implementation we'll use the `XMLHttpRequest` object that offers two events: `onload` and `onerror`. The `onload` event is triggered when the browser receives a response from the server, while the `onerror` is triggered when an error in communication occurs. 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 JSON objects 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 the response and see what the exact situation is. Without going into too much detail, a server can respond with a number of things, but in our case, we only care if that response was successful (status 200). If it isn't, again we have to reject the promise.

Even if the server has successfully responded with some content, this still doesn't mean that we are out in the clear. Since our goal was to get JSON objects from the server, it can always happen that our JSON code has some syntax errors. This is why, when calling the `JSON.parse` method we surround the code with a try-catch statement, and if an exception occurs while parsing the server response, we also reject the promise. This takes care of all the bad scenarios that can happen.

In the case where everything goes according to plan, and we successfully obtain JSON objects, we can safely resolve the promise with the obtained JSON objects. Finally, we can use the `getJSON` function to fetch a number of ninjas from the server:

```
getJSON("ninjas.json").then(function(ninjas){
  assert(ninjas, "Ninjas obtained!");
}).catch(function(e){
  assert(false, "Shouldn't be here:" + e);
});
```

Notice how, in this case, we have three potential sources of errors: errors in establishing the communication between the server and the client; 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 really care about the specific of error sources, we only supply a callback that gets triggered if everything went ok and the data was properly received, and a callback that gets triggered if any error occurred. This makes our lives as developers so much easier.

To be honest, this probably doesn't look as much; ok it gives you a standard way of dealing with callback, but you could probably achieve similar results with a wide variety of JavaScript libraries and regular callbacks. But now, we're going to take it up a notch, and show you the main advantage of promises – their elegant composition. We'll start by chaining a number of promises in a series of distinct steps.

7.2.5 Chaining promises

You've already seen how handling a sequence of interdependent steps leads to the pyramid of doom, a deeply nested, difficult to maintain sequence of callbacks. Promises are a step towards solving that problem.

One way of solving the problem of a sequence of interdependent steps, where consecutive steps utilize the data obtained in the previous steps is to simply chain promises.

You've already seen how, by using the `then` method on a promise, you 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 you from chaining as many `then` methods as you want, see the following code.

```
getJSON("ninjas.json").then(function(ninjas){
  return getJSON(ninjas[0].missionsUrl);
}).then(function(missions){
  return getJSON(missions[0].detailsUrl);
}).then(function(mission){
  assert(plan, "Ninja mission obtained!");
});
```

As usual, we'll stop here a bit and study more deeply what's exactly going on, by taking a closer look at the following listing.

Listing 7.12 Chaining promises

```
var getNinjasPromise = getJSON("ninjas.json");

var getOneNinjaPromise = getNinjasPromise.then(function(ninjas){  #A
  return getJSON(ninjas[0].location); #B
});

var ninjasPromise = getNinjasPromise.then(function(ninjas){ #C
  return ninjas; #2     #C
});

assert(getNinjasPromise != getOneNinjaPromise #D
    && ninjasPromise != getOneNinjaPromise,
       "Three different promises!");           #D

getOneNinjaPromise.then(function(ninja){                        #E
  assert(ninja, "Ninja recieved");
});                                                  #E

ninjasPromise.then(function(ninjas){                            #F
    assert(ninjas && ninjas.length > 0, "Ninjas recieved");
});                                                             #F
```

#A – Calling the `then` method on a promise creates a new promise
#B – When this promise is resolved, we return a new promise!
#C – Create another promise, but this time, return the result that is not a promise!
#D – Test that these are all different promises
#E – As the return value of the callback of the `then` method that has created `getOneNinjaPromise` is another promise, the `getOneNinjaPromise` will be resolved when the callback returned promise is resolved.
#F – As the return value of the callback of the `then` method that has created `ninjasPromise` is not a promise, the `ninjasPromise` is immediately resolved with that value.

In listing 7.12, we create a `getNinjasPromise` by calling our getJSON method, and we call the `then` method on that promise. This creates a completely new promise that we store in the variable `getOneNinjaPromise`. We also register a success callback on the `getNinjasPromise`.

Next, we again call the `then` method on the `getNinjasPromise`, in the process creating another promise that we store in the variable `ninjasPromise`. Notice how there's nothing

special about calling the `then` method multiple times on the same promise. Each call creates a completely new promise, and registers a callback that will be executed when the promise is successfully resolved.

It's important to note that the eventual values of the promises created by invoking `then` depend on the return value of the callback passed in to that `then` call. For example, if the return value of the callback is another promise:

```
var getOneNinjaPromise = getNinjasPromise.then(function(ninjas){
  return getJSON(ninjas[0].location);
});
```

the `getOneNinjaPromise` will be resolved or rejected in the same fashion as `getJSON(ninjas[0].location)`. If the return value of the callback is a non-promise object, as is with `ninjasPromise`:

```
var ninjasPromise = getNinjasPromise.then(function(ninjas){
  return ninjas;
});
```

then the `ninjasPromise` succeeds with that value as a result. In our case the `ninjasPromise` succeeds with the array of ninjas that were returned by the callback.

This means that, in your `then` handler, you can always count on that the sent in value will be a concrete value, and not a promise.

CATCHING ERRORS IN CHAINED PROMISES

When dealing with sequences of asynchronous steps, an error can occur in either of those steps. We already know that we can either provide a second, error callback to the `then` call, or that we can simply chain in a `catch` call that takes an error callback. In cases where we only care about success/failure of the entire sequence of steps, it might be a bit tedious to supply each step with special error handling. So, we'll take advantage of the `catch` method. See the following code.

```
getJSON("ninjas.json").then(function(ninjas){
  return getJSON(ninjas[0].location);
}).then(function(ninja){
  return getJSON(ninja.plan);
}).then(function(plan){
  assert(plan, "Ninja plan obtained!");
}).catch(function(err){
    alert("An error has occurred:" + err);
});
```

If a failure occurs in any of the previous promises, the `catch` method catches it, and if no error occurs, the program flow will continue through it, onward.

Dealing with a sequence of steps is much nicer with promises than with regular callbacks, wouldn't you agree? But it's still not exactly easy as we would like it to be. Don't worry, we'll get to that, but first let's see how to use promises to take care of a number of parallel, asynchronous steps.

7.2.6 Waiting for a number of promises

In addition to helping you deal with sequences of interdependent, asynchronous steps, promises significantly reduce the burden of waiting for a number of independent asynchronous tasks. Let's revisit our little example, in which we want to, in parallel, gather the information about the ninjas at our disposal, the intricacies of the plan, as well as the map of the location where the plan will be set in motion. With promises, this is as simple as:

Listing 7.13 Waiting for a number of promises

```
Promise.all([getJSON("ninjas.json"),                              #A
             getJSON("mapInfo.json"),
             getJSON("plan.json")]).then(function(results)  #A
{
  var ninjas = results[0], mapInfo = results[1], plan = results[2]; #B

  assert(ninjas && mapInfo && plan,
         "The plan is ready to be set in motion!");               #B
}).catch(function(error){
  assert(false, "A problem in carrying out our plan!")
});
```

#A – The `Promise.all` method that takes an array of promises, an creates a new promise that succeeds if all promises succeed, and fails if even one of the promises fails
#B – The result is an array of succeed values, in the order of passed in promises

As Listing 7.13 shows, you don't have to care about the order in which different tasks are executed, whether some of them have finished, while others didn't. You simple state that you want to wait for a number of promises by using the `Promise.all` method. This method takes in a sequence 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 how much more elegant the code that processes multiple parallel asynchronous tasks with promises is.

The `Promise.all` method waits for all promises in a list, but there are certain cases in which we have a number of promises, but we only care that about the first one that succeeds. Meet the `Promise.race` method.

RACING PROMISES

Imagine that you have a group of ninjas at your disposal, and that you want to give the assignment to the first ninja that answers your call. In terms of promises, you would write something like this:

```
Promise.race([getJSON("data/Yoshi.json"),
              getJSON("data/Hatori.json"),
              getJSON("data/Hanzo.json")]).then(function(ninja)
{

  assert(ninja, ninja.name + " responded first");
}).catch(function(error){
  assert(false, "Failure!")
```

```
});
```

It's simple as that, no need for manually tracking everything. You just use the `Promise.race` method that takes an array of promises and returns a completely *new* promise that resolves or rejects as soon as one of the promises resolves or rejects.

So far you've seen how promises work, and how you can use them to greatly simplify dealing with a series of asynchronous steps, either in series, or in parallel. While the improvements, when compared to plain, old callbacks in terms of error handling and code elegance are great, promisified code is still not on the same level of elegance as synchronous code. In the next section, the two big concepts that we've introduced in this chapter: *generators* and *promises*, will come together to provide you with the non-blockingness of asynchronous code, and the simplicity of synchronous code.

7.3 Generators and Promises combined

Let's recap what we've covered so far.

- *Generators* are special functions that can generate sequences of values on a per-request basis. Once a value has been generated, the generator suspends itself, and when the next value is requested, the generator continues where it has left off. It is important to emphasize that this pausing is done without blocking the main thread.
- *Promises*, on the other hand, allow us to easily deal with results of asynchronous tasks.

In this section, we'll show you how to combine the ability of generators to pause and resume their execution with promises, in order to achieve more elegant asynchronous code.

First, we'll revisit the example from the beginning of the chapter, where we had to get the details of the highest rated mission done by the most popular ninja. All of these tasks are long-running and mutually dependent. If we would implement them in a synchronous fashion, we would get the following, straight-forward code:

```
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
}
```

While this code is great for its simplicity, the problem is that it blocks the UI. Ideally, we would like to change this code so that there is no blocking when a long-running task occurs. One way of doing this is by using generators.

Yielding from a generator function pauses the execution of the generator. In order to continue the execution of a generator from that pause position we have to call the `next` method on the generator's iterator. The idea is to combine generators and promises in the following way: we put the code that uses asynchronous tasks in a generator function, and we execute that generator function. When we reach a point in the generator execution that calls

the asynchronous task, we create a promise that represents the value of that asynchronous task. Since we have no idea when that promise 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 function, by calling the iterator's `next` method. We do this as many times as necessary. See the following listing, for how to generalize this idea to a practical example.

Listing 7.14 Combining generators and promises

```
async(function*()                                          #A
{
  try {
    var ninjas = yield getJSON("ninjas.json");  #B
    var missions = yield getJSON(ninjas[0].missionsUrl);
    var missionDescription = yield getJSON(missions[0].detailsUrl);  #B
    //Study the mission details
  }
  catch(e) {                                               #C
    //Oh no, we weren't able to get the mission details
  }                                                        #C
});

function async(generator) {                                #D
  var iterator = generator();                              #E

  function handle(iteratorResult) {                        #F
    if(iteratorResult.done) { return; }                    #G

    var iteratorValue = iteratorResult.value;

    if(iteratorValue instanceof Promise) {                 #H
      iteratorValue.then(function(res) {
          handle(iterator.next(res))
      }).catch(function(err){
        iterator.throw(err);
    });                                                    #H
    }

  try {                                                    #I
    handle(iterator.next());
  }
  catch (e) { iterator.throw(e); }                         #I
}
```

#A – The function making use of asynchronous results should be able to pause while waiting for results. Notice the function *. We are using generators!
#B – Yield on each asynchronous task
#C – We can still use all standard language constructs such as try-catch statements or loops
#D – Define a helper function that will control our generator
#E – Create an iterator through which we'll control the generator
#F – Define the function that will handle each value generated by the generator
#G – Stop when the generator has no more results
#H – 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.
#I – Start the generator execution

The `async` function takes a generator, calls it and creates an iterator that will be used to pause and 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 there's an error and the promise gets rejected, we simply throw that error to the generator, by using the iterator's `throw` method (told you it'll come in handy). We keep doing this, until the generator says it's done.

Now let's look at the generator function. On the first invocation of the iterator's `next` method, the generator executes up to the first `getJSON("ninjas.json")` call. This call creates a promise that will eventually contain the list of information about our `ninjas`. Since this value is fetched asynchronously, we have no idea about how much time it will take the browser to get it. But we know one thing, we don't want to block our application execution while we are waiting. Because of this, 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. Since 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 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 called. If the promise was resolved successfully, the success callback is called, which in turn causes the invocation 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 function, 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 which creates another promise which should eventually contain a list of missions done by the most-popular ninja. Again, since this is an asynchronous task, we have no idea how much it is going to take, so we again yield the execution, and repeat the whole process.

This cycle is repeated as long as there are asynchronous tasks in the generator.

Auf, this was a tad on the complex side, but we really like this example because it combines a lot of things that we'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 pause and resume execution.

- *Promises* – they help us deal with asynchronous code.
- *Callbacks* – we register success and failure callbacks on our promises, and even
- *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.

7.3.1 Combining Generators and promises leads to more elegant code

Now that you understand the whole process, let's take a minute and appreciate how much more elegant the code that implements our business logic is. Instead of something like:

```
getJSON("ninjas.json", function(err, ninjas)
{
  if(err) { console.log("Error fetching ninjas", err); return; }

  getJSON(ninjas[0].missionsUrl, function(err, missions)
  {
     if(err) { console.log("Error locating ninja missions", err); return; }
     console.log(misssions);
  })
});
```

With mixed control-flow, error handling, and not easy to figure out code, we've ended up with:

```
async(function*()
{
  try {
    var ninjas = yield getJSON("ninjas.json");
    var missions = yield getJSON(ninjas[0].missionsUrl);

    console.log(missions);
  }
  catch(e) {
    console.log("Error: ", e);
  }
});
```

The end result is code that 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, and from asynchronous code we get the non-blocking nature; the execution of our application is not blocked while waiting for long-running asynchronous tasks.

7.3.2 Looking onward to ES7 – async functions

Notice how we still had to write some boiler-plate code; we had to develop our little `async` function that takes care of handling promises and requesting values from the generator. While you can write this function only once, and the reuse it throughout your code, it would be even nicer if we wouldn'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 boiler-plate code for us. In ES7 we will be able to write something like this:

```
(async function ()
{
  try {
    var ninjas = await getJSON("ninjas.json");
    var missions = await getJSON(missions[0].missionsUrl);

    console.log(missions);
  }
  catch(e) {
    console.log("Error: ", e);
  }
})()
```

You use the `async` keyword in front of the function keyword to specify that this function relies on asynchronous values, and at every place where you call an asynchronous task, you 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 you don't need the worry about it yourself.

7.4 Summary

In this chapter, you learned:

- Generator functions are functions that generate sequences of values; not all at once, but on a per-request basis.
- Unlike normal functions, generator functions can pause and resume their execution. Once a generator has generated a value it suspends its execution, without blocking the main thread, and patiently waits for the next request.
- A generator function is declared by putting a star '*' after the function keyword. Within the body of the generator function, you can use the new `yield` keyword that yields a value and suspends the execution of the generator.
- Calling a generator function creates an iterator object through which we control the execution of the generator.
- Promises are placeholders for the results of asynchronous computations, it is a guarantee that eventually, we'll know the result of an asynchronous computation. A promise can either succeed or fail.
- Promises significantly simplify our dealings with asynchronous tasks. We can easily deal with sequences of interdependent asynchronous steps by chaining promises by taking advantage of the `then` method. Parallel handling of multiple asynchronous steps is also greatly simplified, just use the `Promise.all` method.
- Finally, we can combine generators and promises to deal with asynchronous tasks with the simplicity of synchronous code.

8

Object-orientation with prototypes

This chapter covers

- Exploring prototypes
- Using functions as constructors
- Extending objects with prototypes
- Avoiding common gotchas
- Building classes with inheritance

Now that we've learned how functions are first-class objects in JavaScript, how closures make them incredibly versatile and useful, and how we can combine generator functions with promises to tackle the problem of asynchronous code, we're ready to tackle another important aspect of JavaScript: *object prototypes.*

A prototype is simply an object to which the search for a particular property can be delegated to. They are a convenient means of defining properties and functionality that will be automatically accessible to other objects.

In other words, they serve a similar purpose to that of *classes* in classical object-oriented languages. Indeed, the main use of prototypes in JavaScript is in producing JavaScript code written in an object-oriented way, similar to, but not exactly like, code in a more conventional, class-based languages such as Java or C#.

In this chapter, we'll go deep into how prototypes work, we'll study their connection to constructor functions, and we'll see how to mimic some of the object-oriented features that are often used in other, more conventional, object-oriented languages. We'll also explore a new addition to JavaScript, the `class` keyword that doesn't exactly bring full featured classes to JavaScript, but does enable us to easily mimic classes and inheritance.

Let's start exploring.

8.1 What are prototypes?

In JavaScript, objects are simply collections of named properties with values. For example, we can easily create new objects with the so called object literal notation:

```
var obj = {
  prop1: 1,                      //#A
       prop2: function(){},      //#B
  prop3: {}                      //#C
      }
```

#A Assign a simple value
#B Assign a function
#C Assign 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;  //#A
obj.prop1 = [];          //#B
delete obj.prop2;   //#C
```

#A `prop1` stores a simple number
#B Assign a value of a completely different type, here an array
#C remove the property from the object

and even by adding completely new properties:

```
obj.prop4 = "Hello"; //#A
```

#A add 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 all the time, so we try to reuse as much code as possible. One form of code reuse that also helps us conceptually deal with our programs in *inheritance*, extending the features of one object into another. In JavaScript, inheritance is implemented with a simple mechanism called *prototyping*.

The idea of prototyping is very 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 8.1 With prototypes objects can access properties of other objects

```
var yoshi = { skulk: true };     //#A
var hatori = { sneak: true };    //#A
var kuma = { creep: true };      //#A

assert("skulk" in yoshi, "Yoshi can skulk");            //#B
assert(!("sneak" in yoshi), "Yoshi cannot sneak");      //#B
assert(!("creep" in yoshi)), "Yoshi cannot creep");     //#B

Object.setPrototypeOf(yoshi, hatori);                   //#C

assert("sneak" in yoshi, "Yoshi can now sneak");        //#D
assert(!("creep" in hatori)), "Hatori cannot creep");   //#E

Object.setPrototypeOf(hatori, kuma);                    //#F
assert("creep" in hatori, "Hatori can now creep");      //#G
assert("creep" in yoshi, "Yoshi can also creep");       //#H
```

#A Creates three objects, each with its own property
#B Yoshi has access only to its own, `skulk` property
#C Use the `Object.setPrototypeOf` method to set one object as the prototype of another object
#D By setting `hatori` as `yoshi`'s prototype, `yoshi` now has access to `hatori`'s properties
#E Currently, `hatori` cannot `creep`
#F Set `kuma` as a prototype of `hatori`
#G Now `hatori` has accees to `creep`
#H `yoshi` also has access to `creep`, through `hatori`.

Listing 8.1 starts with by creating three objects: `yoshi`, `hatori`, and `kuma`, each having one specific property accessible only to that object: only `yoshi` can `skulk`, only `hatori` can `sneak`, and only `kuma` can `creep`. See the following figure:

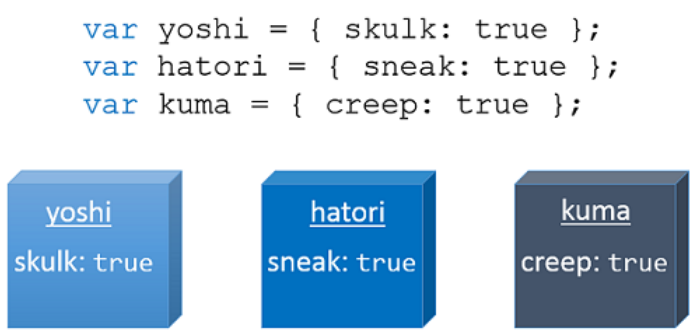


Figure 8.1 Initially, each object has access only to its own properties

In order 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; while executing `"sneak" in yoshi` will return `false`.

In JavaScript, the object's prototype property is an internal property that is not directly accessible (so we'll mark it with `[[prototype]]`). Instead, there exists a built-in method `Object.setPrototypeOf` that takes in two object arguments and sets the second object as the prototype of the first object. For example, calling: `Object.setPrototypeOf(yoshi, hatori);` will set up `hatori` as a prototype of `yoshi`.

This essentially means that whenever we ask `yoshi` for a property that it doesn't have, `yoshi` delegates that search to `hatori`. In essence, `yoshi` now has access `hatori`'s `sneak` property. Check the following figure:

```
var yoshi = { skulk: true };
var hatori = { sneak: true };
var kuma = { creep: true };
Object.setPrototypeOf(yoshi, hatori);
```

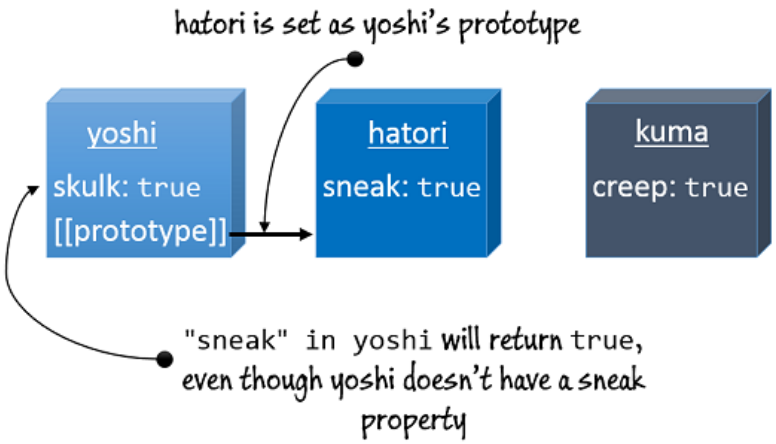


Figure 8.2 When we access a property that the object doesn't have, the object's prototype is searched for that property. Here, we can access `hatori`'s sneak property through `yoshi`, because `yoshi` is `hatori`'s prototype.

We can do a similar thing with `hatori` and `kuma`. By using the `Object.setPrototoypeOf` method, we can set `kuma` as the prototype of `hatori`. Similarly, this means that if we ask `hatori` for a property that he doesn't have, that search will be delegated to `kuma`. In this case, `hatori` has access to `kuma`'s `creep` property. See the following figure.

```
var yoshi = { skulk: true };
var hatori = { sneak: true };
var kuma = { creep: true };
Object.setPrototypeOf(yoshi, hatori);
Object.setPrototypeOf(hatori, kuma);
```

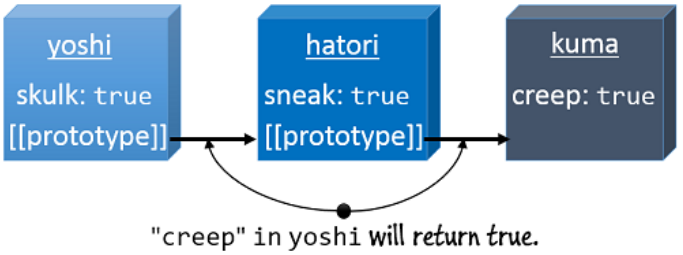


Figure 8.3 The search for a particular property stops when there are no more prototypes to explore. Accessing `yoshi.creep` will will trigger the search first in `yoshi`, then in `hatori`, and finally in `kuma`.

PROTOTYPE CHAIN

It is important to emphasize that every object can have a prototype; an object's prototype can also have a prototype, and so on, forming what is called a *prototype chain*. The search delegation for a particular property is done up the whole chain, and it stops only when there are no more prototypes to explore. For example, if we go back to figure 8.3, asking `yoshi` for the value of the `creep` property, triggers the search for the property first in `yoshi`. Since the property is not found, `yoshi`'s prototype, `hatori` is searched. Again, `hatori` doesn't have a property named `creep`, so `hatori`'s prototype, `kuma` is searched, and the property is finally found.

Now that we have a basic idea on how the search for a particular property is done through the prototype chain. Let's see how the prototypes are used when constructing new objects with constructor functions.

8.2 Object construction and prototypes

The simplest way to create a new object is with a statement like this:

```
var warrior = {};
```

This creates a new and empty object, which we can then populate with properties via assignment statements:

```
var warrior = {};
warrior.name = 'Saito';
warrior.occupation = 'marksman';
```

But those coming from an object-oriented background might miss the encapsulation and structuring that comes with the concept of a class constructor: a function that serves to initialize the 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 is not only tedious but also highly error-prone. We'd like to have a means to consolidate the set of properties and methods for a class of objects in one place.

JavaScript provides such a mechanism, though in a very 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. Rather, the `new` operator, applied to a constructor function (as we observed in chapter 4), 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 is automatically set as the prototype of the objects created with that function. We can modify the function's prototype object at will. Let's see how that works in the following listing:

Listing 8.2 Creating a new instance with a prototyped method

```
function Ninja(){}                                          //#A
Ninja.prototype.swingSword = function(){                    //#B
  return true;                                              //#B
};                                                          //#B

var ninja1 = Ninja();                                       //#C
assert(ninja1 === undefined,                                //#C
       "No instance of Ninja created.");                    //#C

var ninja2 = new Ninja();                                   //#D
assert(ninja2 &&                                            //#D
       ninja2.swingSword &&                                 //#D
       ninja2.swingSword(),                                 //#D
       "Instance exists and method is callable." );         //#D
```

#A Defines a function that does nothing and returns nothing.
#B Every function has a built-in prototype object. We add a method to the prototype of the function.
#C Calls the function as a function. Testing confirms that nothing at all seems to happen.
#D Calls the function as a constructor. Testing confirms that not only is new object instance created, 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: `var ninja1 = Ninja()`, and as a constructor: `var ninja2 = new Ninja()`. After the function is created, we add a `swingSword` method to its `prototype`:

```
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 the following figure for a glimpse of the current application state.

```
function Ninja(){}
Ninja.prototype.swingSword = function(){
  return true;
};
…
var ninja2 = new Ninja();
```

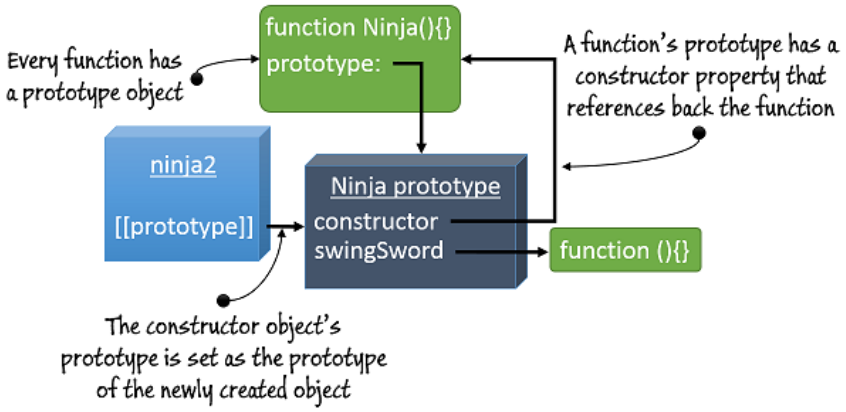


Figure 8.4 Every function, when created, gets a new prototype object. When we use a function as a constructor, the constructed object's prototype is set to the function's prototype.

As can be seen in figure 8.4, a function, when created, gets a new object that is 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 our example, we have extended the `Ninja.prototype` with the `swingSword` method, and when the `ninja2` object is created its `prototype` property is set to `Ninja`'s prototype. This means that when we try to access the `swingSword` property on `ninja2`, the search for that property is delegated to the `Ninja` prototype object. Notice how a*ll* objects created with the `Ninja` constructor will have access to the `swingSword` method. Now that's code reuse!

So 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 a bit.

8.2.1 Instance properties

When the function is called as a constructor via the `new` operator, its context is defined as the new object instance. This means that 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 8.3 Observing the precedence of initialization activities

```
function Ninja(){
 this.swung = false;                                          //#A
 this.swingSword = function(){                                //#B
   return !this.swung;                                        //#B
 };
}
Ninja.prototype.swingSword = function(){                      //#C
  return this.swung;                                          //#C
};                                                            //#C

var ninja = new Ninja();                                      //#D
assert(ninja.swingSword(),                                    //#D
      "Called the instance method, not the prototype method."); //#D
```

#A Creates an instance variable that holds a boolean value initialized to `false`.
#B Creates an instance method that returns the inverse of the swung instance variable value.
#C Defines a prototype method with the same name as the instance method. Which will take precedence?
#D Constructs a `Ninja` instance and asserts that the instance method will override prototype method of the same name.

Listing 8.3 is very 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 actually advise doing in real-world code; quite the opposite. We're doing it here just to demonstrate the precedence of properties.

When we run the test by loading the page into the browser, we see that the test passes! This shows that instance members will hide properties of the same name defined in the prototype, see the following figure:

```
function Ninja(){
  this.swung = false;
  this.swingSword = function(){
    return !this.swung;
  };
 }
 Ninja.prototype.swingSword = function(){
   return this.swung;
 };
 var ninja = new Ninja();
```

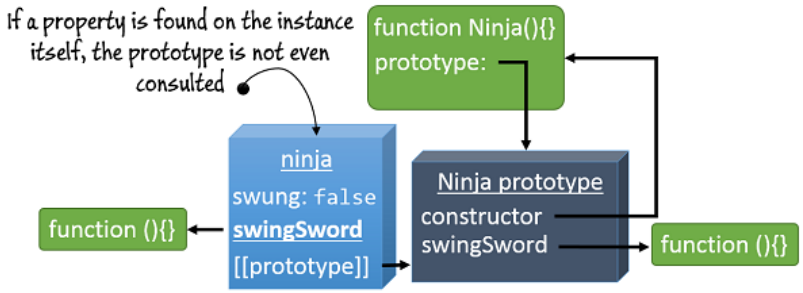


Figure 8.5 If a property can be found on the instance itself, the prototype is not even consulted!

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 is no need for traversing the prototype chain (as we did in figure 8.4), the property created within the constructor is immediately found and returned (see figure 8.5).

There's an interesting side-effect of this. Take a look at the following figure that shows the state of the application if we create a few `ninja` instances.

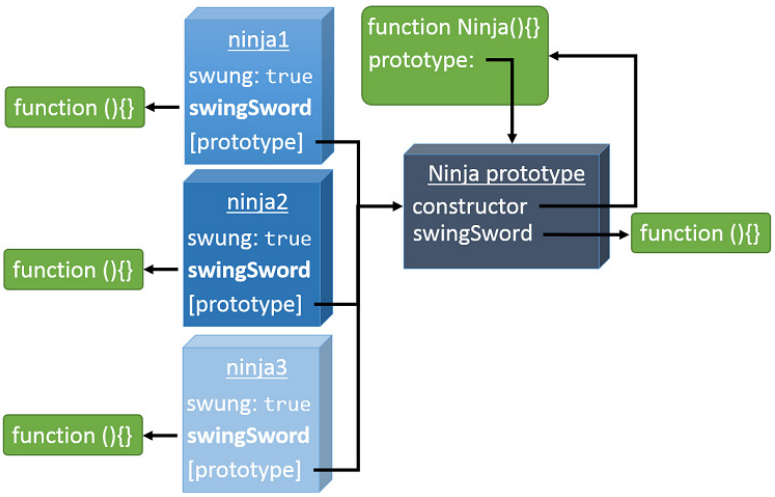


Figure 8.6 Every instance gets its own version of the properties created within the constructor, but they all have access to the same prototype's properties.

As can be seen in figure 8.6, 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 property. This is ok for value properties (for example `swung`) that are specific to each object instance. However, in certain cases it might be a bit problematic for methods.

In this example, we would have three different 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. Since each method copy behaves the same, often it doesn't make any sense to create multiple copies because this will only consume more memory. Sure, in general, the JavaScript engine might do 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 6 on closures: methods defined within constructor functions allow us to mimic private object variables. If this is something we need, then specifying methods within constructors is the only way to go.

8.2.2 Dynamicity caveats

We've already seen how JavaScript is a dynamic language, where properties can be easily added, removed, and modified at will. The same thing holds for prototypes, both the function prototypes and the object prototypes. See the following listing:

Listing 8.4 With prototypes, everything can be changed at runtime

```
function Ninja(){                                          //#A
  this.swung = true;                                       //#A
}                                                          //#A

var ninja1 = new Ninja();                                  //#B

Ninja.prototype.swingSword = function(){                   //#C
  return this.swung;                                       //#C
};                                                         //#C
assert(ninja1.swingSword(),                                //#D
       "Method exists, even out of order.");               //#D

Ninja.prototype = {                                        //#E
 pierce: function() {                                      //#E
   return true;                                            //#E
 }                                                         //#E
}                                                          //#E

assert(ninja1.swingSword(),                                //#F
       "Our ninja can still swing!");                      //#F

var ninja2 = new Ninja();                                  //#G
assert(ninja2.pierce(),"Newly created ninjas can pierce");     //#G
assert(!ninja2.swingSword, "But they cannot swing!");          //#G
```

#A Defines a constructor that creates a `Ninja` with a single boolean property
#B Creates an instance of `Ninja` by calling the constructor function via the new operator
#C Adds a method to the prototype after the object has been created
#D Show that the method exists in the object
#E Completely override the `Ninja`'s prototype with a new object with the `pierce` method
#F Even though we've completely replaced the `Ninja` constructor's prototype, our `Ninja` can still swing a sword, because it keeps a reference to the old `Ninja` prototype.
#G Newly created ninjas reference the new prototype, so they can `pierce` but they cannot `swing`!

In listing 8.4, we again define a `Ninja` constructor and proceed to use it to create an object instance. The state of the application at that moment is shown in figure 8.7.

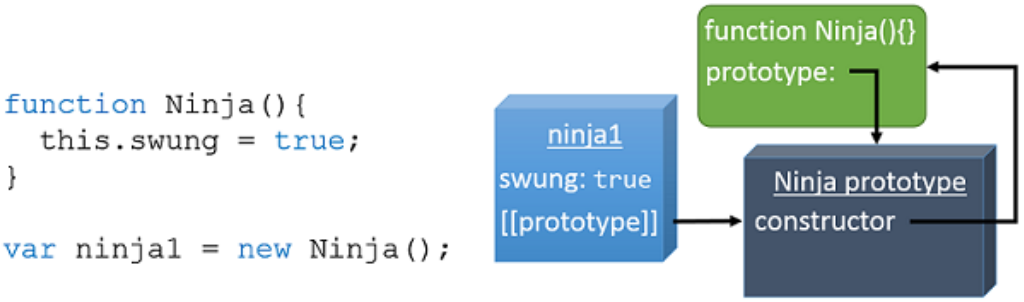


Figure 8.7 After construction, `ninja1` has the property `swung`, and its prototype is the `Ninja` prototype which has only a `constructor` property.

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 the figure 8.7.

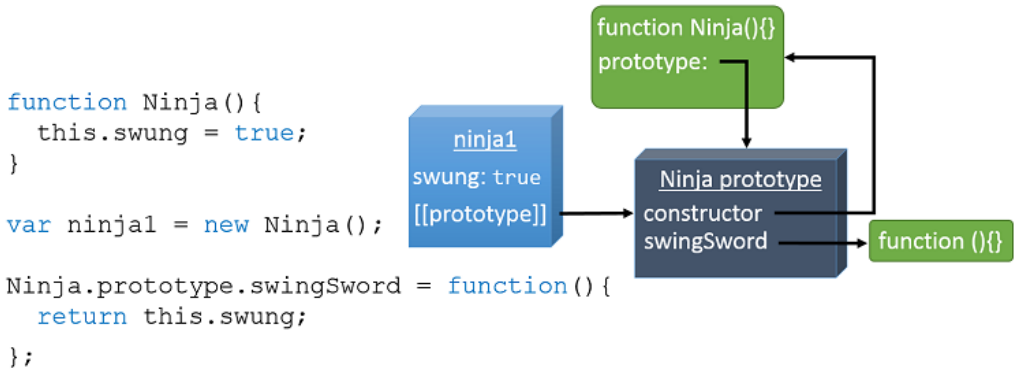


Figure 8.8 Since the `ninja1` instance references the `Ninja` prototype, even changes made after the instance was constructed are accessible.

Later, we completely override the `Ninja` function's prototype by assigning it to a completely new object with the `pierce` method. This results with the application state shown in the following figure.

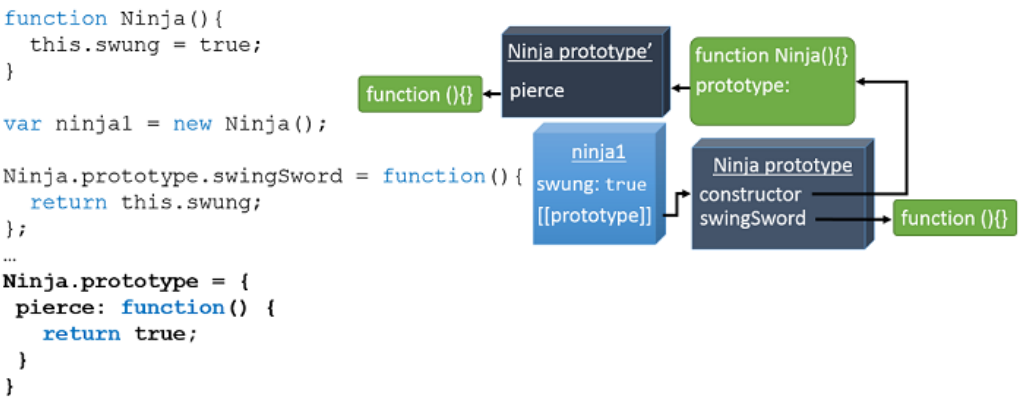


Figure 8.9 Function's prototype can be replaced at will. The already constructed instances reference the old prototype!

As can be seen in figure 8.9, 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.

However, if after this prototype switcheroo, we create new objects, the state of the application will be as shown in the next figure:

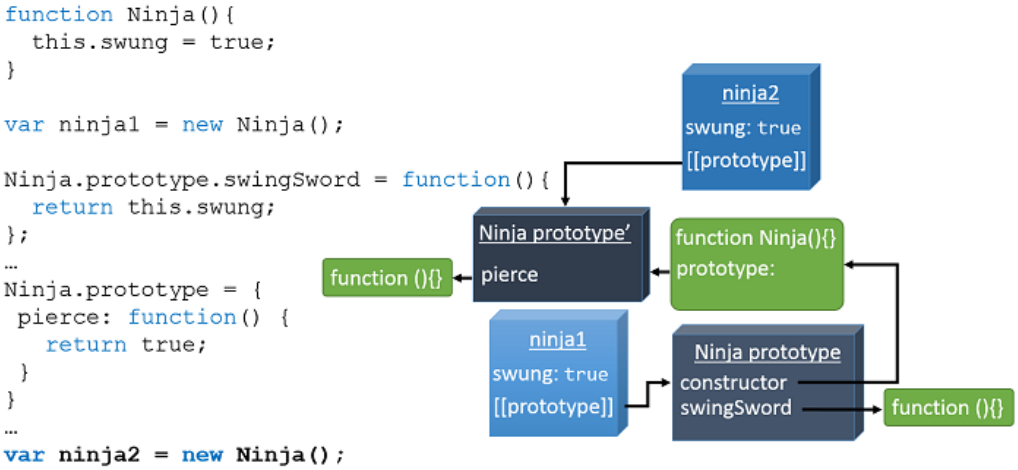


Figure 8.10 All newly created instances, will reference the new prototype.

The reference between an object and the function's prototype is established at the time of object instantiation. This means that newly created objects will have a reference to the new prototype, and will have access to the `pierce` method, while the old, pre-prototype-change objects keep their original prototype, happily swinging their swords away (see figure 8.10).

We went pretty deep into how prototypes work and how they are 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.

8.2.3 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 for us to know which function constructed the object instance.

As we've seen earlier, the constructor of an object is available via the `constructor` property of the constructor function prototype. For example, the following figure shows the state of the application when we instantiate an object with the `Ninja` constructor.

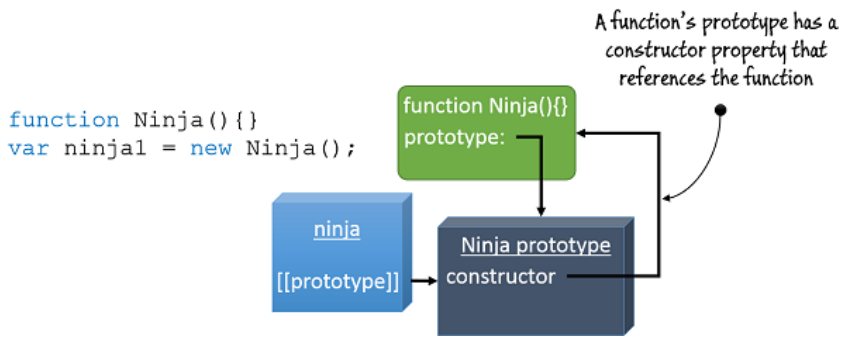


Figure 8.11 The prototype object of each function has a constructor property that references back to the function.

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 8.5 Examining the type of an instance and its constructor

```
function Ninja(){}
var ninja = new Ninja();

assert(typeof ninja == "object",                                    #A
      "The type of the instance is object.");
assert(ninja instanceof Ninja,                                      #B
         "instanceof identifies the constructor." );
assert(ninja.constructor == Ninja,                                 #C
         "The ninja object was created by the Ninja function.");
```

#A Tests the type of ninja using typeof. That tells us it's an object, but not much else.
#B Tests the type of ninja using instanceof. This gives us more information—that it was constructed from Ninja.
#C Tests the type of ninja using the constructor reference. This gives us an actual reference to the constructor function.

In listing 8.5 we define a constructor and create an object instance using it. Then we examine the type of the instance using the `typeof` operator. This isn't very revealing, as all instances will be objects, thus always returning `"object"` as the result. Much more interesting is the `instanceof` operator, which is really helpful in that it gives us a way to determine whether an instance was created by a particular function constructor. We'll go a bit deeper into how the `instanceof` operator works later in the chapter.

On top of this, we can also make use of the `constructor` property that we now know is accessible to all instances, as a reference back to the original function that created it. We can use this to verify the origin of the instance (much like how we can with the `instanceof` operator).

Additionally, because this is just a reference back to the original constructor, we can instantiate a new `Ninja` object using it, as shown in the next listing.

Listing 8.6 Instantiating a new object using a reference to a constructor

```
function Ninja(){}

var ninja = new Ninja();
var ninja2 = new ninja.constructor();                                    #A

assert(ninja2 instanceof Ninja, "It's a Ninja!");                         #B
assert(ninja !== ninja2, "But not the same Ninja!");                      #C
```

#A Constructs a second Ninja from the first
#B Proves the new object's Ninja-ness
#C They aren't the same object but two distinct instances

Listing 8.6 defines a constructor and creates 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 one might think of some malicious ones), as its reason for being is to inform us from where the object was constructed. If the `constructor` property is overwritten, the original value is simply lost.

That's all very useful, but we've just scratched the surface of the superpowers that prototypes confer on us. Now things get really interesting.

8.3 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 different than in other popular object-oriented languages. Let's consider the example in the following listing, in which we'll attempt to achieve inheritance.

Listing 8.7 Trying to achieve inheritance with prototypes

```
function Person(){}                                          #A
Person.prototype.dance = function(){};                       #A

function Ninja(){}                                           #B
Ninja.prototype = { dance: Person.prototype.dance };         #C
```

```
var 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" );
```

#A Defines a dancing Person via a constructor and its prototype
#B Defines a Ninja
#C Attempts to make Ninja a dancing Person by copying the dance method from the Person prototype

As the prototype of a function is just 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 while we may have taught the ninja to dance, we failed to make the `Ninja` a `Person`, as shown in figure 8.12. Although we've taught the `Ninja` to mimic the dance of a person, it hasn't *made* the `Ninja` a `Person`. That's not inheritance—it's just copying.

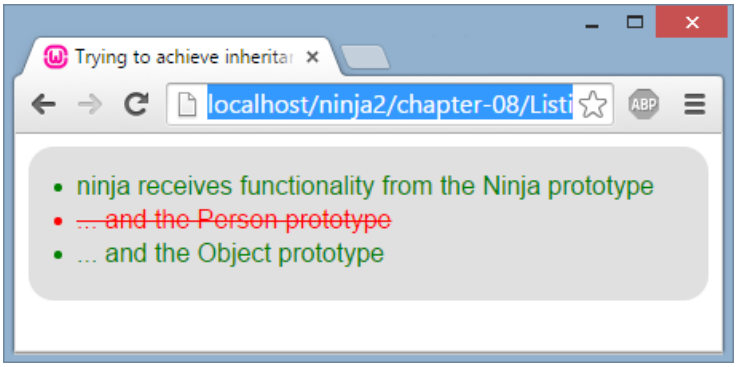


Figure 8.12 Our `Ninja` isn't really a `Person`. No happy dance!

Apart from the fact that this approach is not 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 will preserve 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 `SuperClass`, and which will in turn have a prototype pointing to an instance of *its* superclass, and on and on.

Let's change the code of listing 8.7 slightly to use this technique in the next listing.

Listing 8.8 Achieving inheritance with prototypes

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();                          #A

var 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!")
```

#A Makes a Ninja a Person by making the Ninja prototype an instance of Person.

The only change we made to the code was to use an instance of `Person` as the prototype for `Ninja`. Running the tests shows that we've succeeded, as shown in figure 8.13.

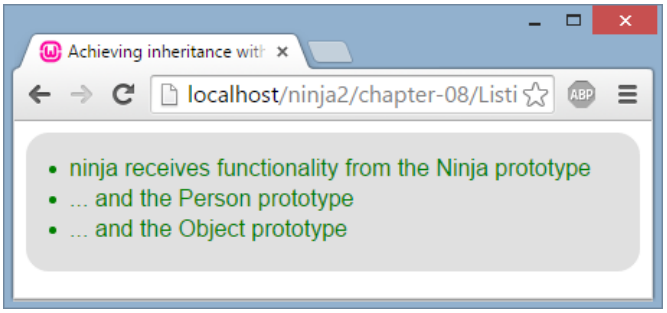


Figure 8.13 Our `Ninja` is a `Person`! Let the victory dance begin.

Now we'll take a closer look at how this exactly works by looking at the state of the application after we've created the new `ninja` object, as shown in the following figure:

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();
var ninja = new Ninja();
```

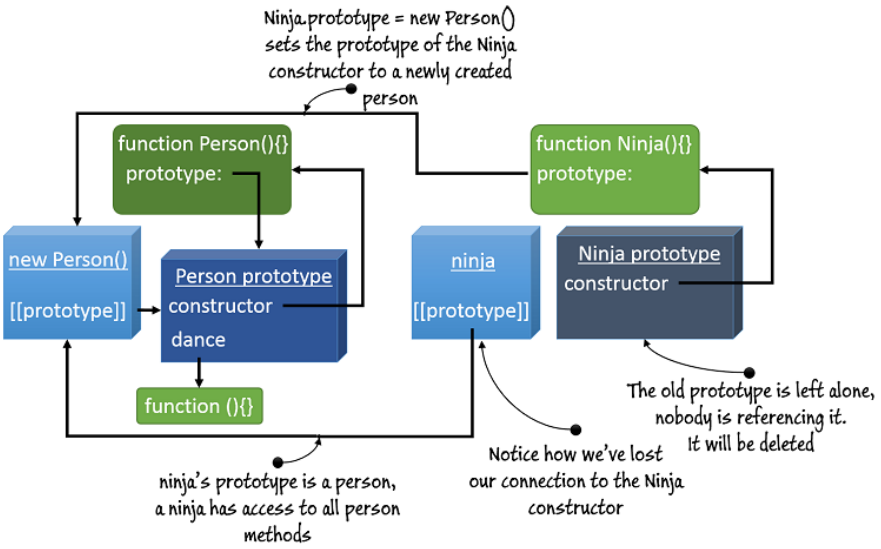


Figure 8.14 We've achieved inheritance by setting the prototype of the Ninja constructor to a new instance of a Person object.

Figure 8.14 shows that, when we define a `Person` function, a Person 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:

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 `canDance` method through the `ninja` object, the JavaScript runtime will first check the `ninja` object itself. Since it does not have the `canDance` property, its prototype is searched, the person object. The person object also doesn't have the `canDance` property, so its prototype is searched, and the property is finally found. This is how we achieve inheritance in JavaScript! (see figure 8.13)

The very important implications of this are that 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`. By doing this, any changes to the `Ninja` prototype will 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, object that inherit from the prototype always have access to the current prototype properties.

8.3.1 The problem of overriding the constructor property

If we take a closer look at figure 8.13, we'll see that by setting the new `Person` object as a prototype of the `Ninja` constructor, we've effectively lost our connection to the `Ninja` constructor that was previously kept by the original `Ninja` prototype. This is a problem, since 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");
```

However, in our current state of the application this test fails. As figure 8.13 shows, if we search the `ninja` object for the property `constructor`, we won't find it, so we go over to its prototype, which also does not have one, again, we follow the prototype and end up in the prototype object of the `Person`, which has a `constructor` property referencing the `Person` function. So in effect, we get the wrong answer: if we ask the `ninja` object which function has constructed it, we will get `Person` as the answer. This can be the source of some serious bugs.

It is 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 descriptor of the property can be changed and

the property can be deleted, and 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. Cannot 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 cannot be defined in conjunction with *value* and *writable*.

If we create a property through a simple assignment, for example:

```
ninja.name = "Yoshi";
```

The property will be configurable, enumerable, and writable, its value will be set to "`Yoshi`", and functions `get` and `set` would be `undefined`.

In cases where we want to fine-tune our property configuration, we can use the built-in `Object.defineProperty` method that takes in an object on which the property will be defined, the name of the property, and a property descriptor object. For example, by using the following code:

Listing 8.9 Configuring properties

```
var ninja = {};                  //#A
ninja.name = "Yoshi";            //#A
ninja.weapon = "kusarigama";     //#A

Object.defineProperty(ninja, "sneaky", {  //#B
  configurable: false,                    //#B
  enumerable: false,                      //#B
  value: true,                            //#B
  writable: true                          //#B
});                                       //#B

assert(ninja.sneaky, "We can access the new property");

for(var prop in ninja){                                 //#C
  assert(true, "An enumerated property: " + prop);      //#C
}                                                       //#C
```

#A Create an empty object, use assignments to add two properties
#B The built-in `Object.defineProperty` method is used to fine-tune the property configuration details
#C Use the for…in loop to iterate over ninja's enumerable properties

Listing 8.9 starts 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`, that is not `configurable`, not `enumerable`, it has its `value` set to `true`, and its `value` can be changed because it is `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. The following figure shows the result.

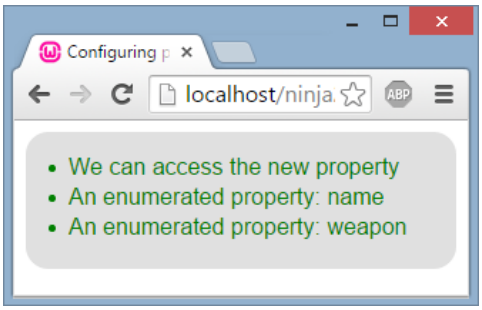


Figure 8.15 Properties `name` and `weapon` will be visited in the for…in loop, while our specially added `sneaky` property, will not (even though we can access it normally).

By setting `enumerable` to `false`, we can be sure that the property will not appear when using the `for…in` loop. For why we would like to do something like this, we'll go back to the original problem that we were trying to solve.

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 problem that 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 that was 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 simply define a new `constructor` property on the new `Ninja.prototye` by using the `Object.defineProperty` method, see the following listing.

Listing 8.10. Fixing the constructor property problem

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();

Object.defineProperty(Ninja.prototype, "constructor", { //#A
  enumerable: false,                                    //#A
  value: Ninja,                                         //#A
  writable: true                                        //#A
});                                                     //#A

var ninja = new Ninja();
```

```
assert(ninja.constructor === Ninja,                          //#B
      "Connection from ninja instances to Ninja constructor     reestablished!");
                                                             //#B
for(var prop in Ninja.prototype){                                    //#C
  assert(prop == "dance", "The only enumerable property is dance!"); //#C
}                                                                    //#C
```

#A We define a new non-enumerable `constructor` property pointing back to `Ninja`
#B We've reestablished the connection
#C 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 `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`, will not be visited. Now that's the mark of a true ninja, we went deep in, did our job, and got out, without anybody noticing anything from the outside!

8.3.2 The instanceof operator

In most programming languages, the straight-forward approach for checking whether an object is a part of some class hierarchy is to simply use the `instanceof` operator. For example, in Java, the `instanceof` operator works by checking whether the object on the left hand side is either the same class, or a subclass of the class type on the right hand side.

While certain parallels could be made with how the `instanceof` operator works in JavaScript, as a lot of things, the `instanceof` operator comes with a little twist.

In JavaScript, the `instanceof` operator works on the prototype chain of the object. For example if we have the following expression:

```
ninja instanceof Ninja
```

the `instanceof` operator works by checking is the *current* prototype of the `Ninja` function in the prototype chain of the `ninja` instance. Let's go back to our persons and ninjas, for a more concrete example:

Listing 8.11 Studying the instanceof operator

```
function Person(){}
function Ninja(){}

Ninja.prototype = new Person();

var ninja = new Ninja();

assert(ninja instanceof Ninja, "Our ninja is a Ninja!");        //#A
assert(ninja instanceof Person, "A ninja is also a Person. ");  //#A
```

#A 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, just to nail the point down, let's study the following figure for how the whole thing works behind the scenes.

```
function Person(){}
function Ninja(){}

Ninja.prototype = new Person();
var ninja = new Ninja();
```

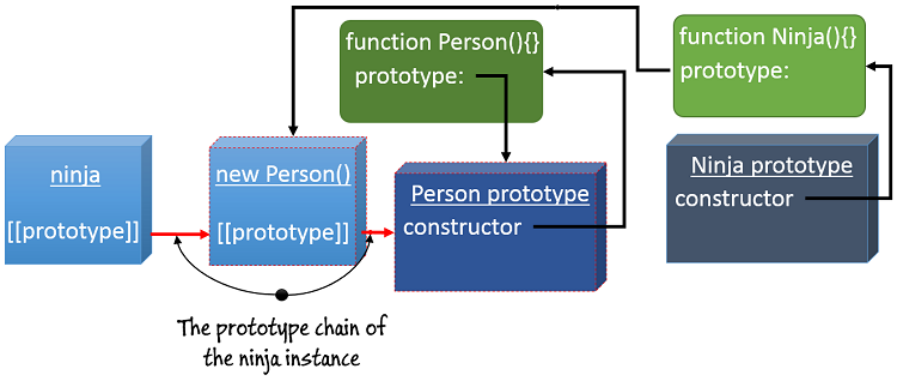


Figure 8.16 The prototype chain of a ninja instance is composed out of a new Person() object and the Person prototype.

As figure 8.16 shows, the prototype chain of a ninja instance is composed out 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 is in the prototype chain of the `ninja` instance. Since 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, since it is 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's to know about the `instanceof` operator. While it's most common usage is that it should give us a clear way to determine whether an instance was created by a particular function constructor, it doesn't exactly work like that. Instead, it simply checks whether the prototype of the right hand side function is in the prototype chain of the object on the left hand side. Because of this there are some caveats that you should be careful about.

THE INSTANCEOF CAVEAT

As we'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, see the following example.

Listing 8.12 Watch out for changes to constructor prototypes

```
function Ninja(){}

var ninja = new Ninja();

assert(ninja instanceof Ninja, "Our ninja is a Ninja!");

Ninja.prototype = {}; //#A

assert(!(ninja instanceof Ninja), "The ninja is now not a Ninja!?"); //#B
```

#A We change the prototype of the Ninja constructor function
#B Even though our ninja instance was created by the Ninja constructor, the `instanceof` operator now says that ninja is not an instance of Ninja any more!

In listing 8.12, we again repeat all the basic steps of making a `ninja` instance, and our first test goes fine. However, if we change the prototype of the `Ninja` constructor function *after* the creation of the `ninja` instance, and again test whether `ninja` is an `instanceof Ninja`, we will see that the situation has changed. This will only surprise us if we cling on to the inaccurate assumption that the `instanceof` operator tells us whether an instance was created by a particular function constructor. If we, on the other hand, take the real semantics of the `instanceof` operator: that it simply checks whether the prototype of the function on the right hand side is in the prototype chain of the object of the left hand side, there will be no surprises.

Now that we have a deep understanding of how prototypes work in JavaScript, and how we can 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.

8.4 JavaScript "classes" in ES6

While it's great that JavaScript lets us use a form of inheritance via prototypes, a common desire for many developers, especially those from a classical object-oriented background, is a simplification or abstraction of JavaScript's inheritance system into one that they're more familiar with.

This inevitably leads us toward the realm of classes; that is, what a typical object-oriented developer would expect, even though JavaScript doesn't support classical inheritance natively. As a response to this need, a number of JavaScript libraries, that simulate classical inheritance have popped out. Since each of these libraries implements classes in its own, slightly specific way, the ECMAScript committee has decided to standardize 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 is added to the new ES6 version of JavaScript, not all browsers currently implement it (see http://kangax.github.io/compat-table/es6/ for the current support).

Let's start by studying the new syntax.

8.4.1 Using the class keyword

The new ECMAScript 6 introduces a new `class` keyword that enables a much more elegant way of creating objects and implementing inheritance, than what we had to do if manually implementing it ourselves with prototypes. Using the `class` keyword is really easy, just take a look at the following code example:

Listing 8.13 Creating a class in ES6

```
class Ninja{                        //#A
  constructor(name){                //#B
    this.name = name;               //#B
  }                                 //#B

  swingSword(){                     //#C
    return true;                    //#C
  }                                 //#C
}

var ninja = new Ninja("Yoshi");                                    //#D

assert(ninja instanceof Ninja, "Our ninja is a Ninja");            //#E
assert(ninja.name === "Yoshi", "named Yoshi");                     //#E
assert(ninja.swingSword(), "and he can swing a sword");            //#E
```

#A Use the `class` keyword to start specifying an ES6 class
#B Define a constructor function that will be called when we call the class with the keyword `new`
#C Define an additional method accessible to all `Ninja` instances
#D Instantiate a new `ninja` object with the keyword `new`
#E Test for the expected behavior

Listing 8.13 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 simply 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 like we would if `Ninja` was a simple constructor function (like earlier in the chapter):

```
var ninja = new Ninja("Yoshi");
```

Finally, we can tests that the `ninja` instance behaves as expected, that it is an `instanceof Ninja`, that it 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 we mentioned earlier, even though ES6 has introduced the `class` keyword, under the hood we are still dealing with good old prototypes; classes are simply syntactic sugar designed to make our lives a bit easier when mimicking classes in JavaScript.

Our class code from Listing 8.13 can be translated to functionally identical ES5 code:

```
function Ninja(name) {
  this.name = name;
}
Ninja.prototype.swingSword = function() {
  return true;
}
```

As we can see there's nothing especially new with ES6 classes, the code is simply more elegant, but the same concepts are still applied.

STATIC METHODS

In the previous examples, we've seen how to define object methods, actually prototype methods, accessible to all object instances. In addition to such methods, in more classical object-oriented languages, such as Java, there's also the notion of static methods, methods defined on class level. Check out the following example.

Listing 8.14 Static methods in ES6

```
class Ninja{
  constructor(name, level){
    this.name = name;
    this.level = level;
  }

  swingSword() {
    return true;
  }

  static compare(ninja1, ninja2){       //#A
    return ninja1.level - ninja2.level; //#A
  }                                     //#A
```

```
}

var ninja1 = new Ninja("Yoshi", 4);
var ninja2 = new Ninja("Hatori", 3);

assert(!("compare" in ninja1) && !("compare" in ninja2), //#B
       "A ninja instance doesn't know how to compare");  //#B

assert(Ninja.compare(ninja1, ninja2) > 0,                //#C
      "The Ninja class can do the comparison!");         //#C

assert(!("swingSword" in Ninja),
      "The Ninja class can not swing a sword");
```

#A Use the `static` keyword to make a static method
#B Ninja instances don't have access to `compare`
#C The class `Ninja` has access to the `compare` method

In listing 8.14, we again create a `Ninja` class that has a `swingSword` method which is accessible from all ninja instances. We also define a static method `compare`, by simply prefixing the method name with the keyword `static`.

```
static compare(ninja1, ninja2){
    return ninja1.level - ninja2.level;
}
```

This means that the `compare` method, which compares the skill levels of two ninjas, is defined on class level, and not instance level! Later we test that this effectively means that the `compare` method is not accessible from ninja instances, but it is 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!");
```

Also, the `swingSword` method can only be accessed through `Ninja` instances, and not through the `Ninja` class.

Now let's move onto inheritance.

8.4.2 Implementing inheritance

To be really honest, performing inheritance in pre-ES6 code is a bit of 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
});
```

As we can see, 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, like we did with the `dance` method and the `Person` constructor. If we want to implement inheritance, then we have to set the prototype of the derived "class" to the instance of the base "class", in our 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 as inheritance. Luckily, with ES6 all of this is significantly simplified.

Let's see how it's done in the following listing.

Listing 8.15 Inheritance in ES6

```
class Person {
  constructor(name){
    this.name = name;
  }

  dance(){
    return true;
  }
}

class Ninja extends Person {    #A
  construct(name, weapon){
    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");
```

#A use the `extends` keyword to inherit from another class

Listing 8.15 shows how to achieve inheritance in ES6: we simply 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 {
  construct(name, weapon){
    this.weapon = weapon;
  }

  wieldWeapon(){
    return true;
  }
}
```

We continue by creating a `person` instance and checking that it is an `instanceof` the `Person` class that has a `name` and can `dance`. Just to be sure, we also check that a person is *not* a Ninja that can 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 we check that it is an `instanceof Ninja` and that it can wield a weapon. Since every `ninja` is also a `Person`, we check that a ninja is an `instanceof 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 was, there was no need for thinking about prototypes or the side effects of certain overridden properties, we simply 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.

There's one more thing, that won't be surprising to anyone with a more conventional object-oriented background, we would like to turn your attention to. Notice how we've created our `ninja` with the expression: `new Ninja()`, and that our `ninja` has a `name` property even though we don't mention the `name` property anywhere within the body of the `Ninja` class. The thing is, since the `Ninja` class `extends` the `Person` class, whenever we instantiate a new `Ninja` object, before calling the `Ninja` constructor, the `Person` constructor is called with the arguments that we've passed onto the `Ninja` constructor. So the `name` property of our `ninja` object is set in the `Person`'s constructor.

And that's really it, with ES6 we build class hierarchies almost as easily as in any other, more conventional object-oriented language. To be honest, there are still some things missing, but we're sure that the JavaScript will continue catching on.

8.5 Summary

In this chapter, we learned:

- 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 the search for a particular property will be delegated to, 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 is 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.
- Be careful, in JavaScript, almost everything can be changed at runtime, object's prototypes, function's prototypes!
- If we want that the instances created by a `Ninja` constructor function "inherit" (more accurately, have access to) properties accessible to instances created by the `Person` constructor function, simply set the prototype of the `Ninja` constructor to a new instance of the `Person` class.
- In JavaScript, properties have attributes (configurable, enumerable, writable). This properties can be defined by using the `Object.defineProperty` built in method.
- JavaScript, ES6 adds support for a `class` keyword that enables us to more easily mimic class in JavaScript. Behind the scenes, prototypes are still in play!
  - The `extends` keyword enables elegant inheritance.