Community Experience Distilled

JavaScript Promises Essentials

Build fully functional web applications using Promises, the new standard in JavaScript

Rami Sarieddine

[PACKT] open source*
community experience distilled
PUBLISHING

JavaScript Promises Essentials

Build fully functional web applications using promises, the new standard in JavaScript

Rami Sarieddine

[PACKT] PUBLISHING open source
community experience distilled

BIRMINGHAM - MUMBAI

JavaScript Promises Essentials

First published: September 2014

Production reference: 1190914

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-564-7

www.packtpub.com

Credits

Author
Rami Sarieddine

Reviewers
Chris Latko
Jan Oetjen
Rajeesh C V

Commissioning Editor
Edward Gordon

Acquisition Editor
Owen Roberts

Content Development Editor
Akshay Nair

Technical Editor
Edwin Moses

Copy Editors
Sarang Chari
Gladson Monteiro
Adithi Shetty

Project Coordinator
Swati Kumari

Proofreaders
Stephen Copestake
Paul Hindle

Indexers
Hemangini Bari
Mariammal Chettiyar
Monica Ajmera Mehta

Graphics
Abhinash Sahu

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Author

Rami Sarieddine is a passionate technology evangelist and a subject matter expert in web development, Windows apps, and Microsoft Azure. He is currently working as a Technical Evangelist with Microsoft. This is his second publication, following his first book *Developing Windows Store Apps with HTML5 and JavaScript*, *Packt Publishing* (`https://www.packtpub.com/application-development/developing-windows-store-apps-html5-and-javascript#`).

He has more than 8 years of experience in programming for the Web. He was awarded the Microsoft Most Valuable Professional status twice, in 2012 and 2013, for his contributions to the technical communities of ASP.NET/IIS and Visual C#. He has featured as a regular speaker at key technical events for Microsoft Lebanon, including Open Door and TechDays. He has delivered hundreds of training sessions on HTML5, JavaScript, Azure, and Visual Studio.

When not working, Rami enjoys running and spending time with his loved ones. And when on vacation, he enjoys traveling and visiting new places around the world.

He can be reached via his e-mail ID, `r.sarieddine@live.com`. You can follow his tweets as well; his Twitter handle is `@ramiesays`. You can also find his blog posts on `http://code4word.com`.

Acknowledgments

Apart from the hard work put into researching and writing, the realization of this book would not have been possible without my publisher, Packt Publishing, and the efforts of their Acquisition Editor and the team of reviewers who helped me complete the book. I would like to thank Owen Roberts (Acquisition Editor) and Akshay Nair (Content Development Editor) for their involvement and guidance.

Moreover, I wish to acknowledge Brian Cavalier, co-editor and contributor to the concepts and content of Promises/A+, who has given generous support and provided me with key information in this regard. I would also like to take this opportunity to thank my family, who always encourage me in such endeavors.

Above all, I want to express my sincerest appreciation and gratitude to my loved one, Elissar Mezher, for always motivating me and pushing me forward. Her relentless support and follow-up on my progress while writing this book were essential, in addition to bearing with me on those long days and nights at work.

Thank you.

About the Reviewers

Chris Latko graduated with honors from Indiana University with a double major in Business Economics and Japanese Language and Literature. In 1994, he developed his first website and has since acquired 20 years' experience in web development, a major part of which has been his participation in the development and deployment of over 200 websites that required his contribution in virtually all areas of the production process, including management, programming, design, and system administration. Other keynotes in his career include working for Fortune Global 500 companies (Disney and Sony) and several start-ups (PropertyMaps, WTIC, Brand Ichiba, and so on).

He is currently a Lead API Engineer at Boxfish, a company blazing new paths in capturing and understanding TV content and audiences.

I would like to thank my extremely supportive wife, Tomoko, and my wonderful daughter, Luna, for allowing me the late nights and technical conversations.

Jan Oetjen has been an all-round software developer from Germany since 1995. In the past, he has used languages such as Java, JavaScript, Ruby, Python, C, and C# on several frontends and backends. He currently works as a backend developer for Web Solutions.

Jan can be reached at `oetjenj@gmail.com`.

Rajeesh C V is a developer who has been writing code for a living, and fun, for more than a decade. He has hands-on experience in Microsoft technologies, such as C#, ASP.Net MVC, SQL Server, and XAML. A few years back, he fell in love with JavaScript, and thereafter, found time to learn more about it. Now, he is not afraid of JavaScript and recently built a large single-page application using Ember.js for a big corporation. In his free time, he writes for his blog at `http://rajeeshcv.com/`. He lives in Kerala, India, with his beautiful wife, Manjusha.

Currently, he is working in IdentityMine as a Technical Team Lead and is building applications for Windows 8, Windows Phone, and Xbox platforms.

You can follow him on Twitter; his handle is `@cvrajeesh`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

PACKTLIB™

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

JavaScript Promises Essentials is a practical guide to the new concept of promises. It presents a single resource replacing all the scattered information out there about the topic. It covers in detail the new standard that will enhance the way we do asynchronous programming in JavaScript. The book is a brief, yet concise, explanation of the Promises API and its features and how it can be used in JavaScript programming. It covers the essentials of JavaScript promises, touching on the details that matter the most in your new learning with some very useful tips on the different aspects of the topic.

Promises are for the most part a programming concept and provide a process that allows developers to arrange work to be executed on data and values that do not yet exist and allows them to deal with those values at an undetermined point in the future (asynchronously). It also presents an abstraction to handle the communication with asynchronous APIs. Currently, there are ways to achieve asynchronous calling in JavaScript through callback, timers, and events, but all come with caveats. Promises solve real development headaches and allow developers to handle JavaScript's asynchronous operations in a more native manner compared to the traditional approaches. Furthermore, promises present a straightforward correspondence between synchronous functionality and asynchronous functions, especially at the level of error handling. Several libraries have started using promises and now offer a robust implementation of promises. You can find promises on the Web in many libraries and also by interacting with Node.js and WinRT. Learning the details of promise implementations will help you avoid the many problems that arise in the world of asynchronous JavaScript and to construct better JavaScript APIs.

What this book covers

Chapter 1, JavaScript Promises – Why Should I Care?, presents an introduction to the world of asynchronous programming in JavaScript and the importance of promises in that world.

Chapter 2, *The Promise API and Its Compatibility*, takes you through more details of the Promises API. We will also learn about current browser support for the promises standard and have a look at the JavaScript libraries out there that implement promises and promise-like features.

Chapter 3, *Chaining of Promises*, shows you how promises allow easy chaining of asynchronous operations and what that entails. This chapter also covers how to queue asynchronous operations.

Chapter 4, *Error Handling*, covers exceptions and error handling in JavaScript. This chapter will also explain how promises make error handling easier and better.

Chapter 5, *Promises in WinJS*, explores all about the WinJS.Promise object and how it is used in Windows Apps development.

Chapter 6, *Putting It All Together – Promises in Action*, shows you the promises in action and how and where we can use promises in scenarios that put together everything we have learned so far.

What you need for this book

In order to implement what you will be learning in this book, you just need an HTML and JavaScript editor. You can choose from the following:

- Microsoft Visual Studio Express 2013 for Web: This provides a full-featured markup and code editors.
- WebMatrix: This is another option to run the sample code. It is a free, lightweight, cloud-connected web development tool that leverages the latest web standards and popular JavaScript libraries.
- jsFiddle: This is an online web editor that allows you to write HTML and JavaScript code and run it directly in the browser.

Who this book is for

This book is for all developers who are involved in JavaScript programming, be it in web development or with technologies, such as Node.js and WinRT, which make heavy use of asynchronous APIs. Moreover, it targets developers who are interested in learning about asynchronous programming in JavaScript and the new standard that will make that experience much better. In short, this book is for everyone who wants to learn about the new kid on the block that goes by the name of JavaScript Promise.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can achieve this event-based technique using the `object.addEventListener()` method."

A block of code is set as follows:

```
var testDiv = document.getElementById("testDiv");
testDiv.addEventListener("click", function(){
  // so the testDiv object has been clicked on, now we can do
    things
    alert("I'm here!");
});
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Name the app as you please, and click on **Ok**."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

JavaScript Promises – Why Should I Care?

There was never a time before when JavaScript had been this popular. What was once and maybe still is for some, the most misunderstood programing language, mainly because of its name, now ranks among the top popular programming languages available. Furthermore, nearly every personal computer out there might have at least one JavaScript interpreter in use or at least installed. The growing popularity of JavaScript is entirely due to its role as the scripting language of the Web. When it was first developed, JavaScript was designed to run in Netscape Navigator. Its success there led to it becoming the standard equipment in virtually all the web browsers, but JavaScript has grown and matured and is now exposed to a large portion of development not related to the Web. In this first chapter, we will be covering a brief introduction about the following:

- Asynchronous programming in JavaScript
- Issues that developers face with traditional approaches to handle asynchronous operations
- Introduction to JavaScript Promises
- Why we should care about promises when comparing it to the common way of doing things asynchronously

Asynchronous programming in JavaScript

When it comes to asynchronous programming in JavaScript, there are two things to talk about: the web and programming language. The web environment represented by browsers is different from the desktop environment and this reflects in the way we program and code for each of them. Browsers, contrary to the desktop environment, provide a single thread for everything that needs access to the user interface; in HTML terms, the DOM. This single threading model has a negative impact on the application code that might need to access and modify the UI elements, because it restricts the execution of that code to the same thread. Hence, we will have blocking functions and threads that basically block the UI until that thread has been executed. That is why, in web development, it is highly important to take advantage of any asynchronous capabilities the browser offers.

Let's review some history to get more context. Back in the day, websites comprised complete HTML pages in which every user action needed the entire web page to be loaded from the server. This caused a lot of problems for developers, especially when writing a server-side code that would affect the page. Furthermore, it resulted in an unpleasant end user experience. Responding to a user action or changes to an HTML form were carried on with an HTTP `POST` request to the same page that the form was on; this caused the server to refresh the same page using the information it had just received. This entire process and model were inefficient as it resulted in having the entire content of the page disappear, and then reappear, and sometimes the content would get lost along the way in a slow internet environment. The browser then reloaded a web page resending all of the content even though only some of the information had changed; this used excessive bandwidth and resulted in additional load on the server. Additionally, it reflected negatively on the user experience. Later, with much work and effort from different parties in the industry, asynchronous web technologies started emerging to help address this limitation. A famous player in this area is **Asynchronous JavaScript and XML** (**AJAX**), which is a group of technologies used on the client-side to create web applications that communicate in an asynchronous manner. The AJAX technology allowed web applications to send data to and retrieve data from a server in an asynchronous manner without interfering with the UI and behavior of the current page; basically, without the need to reload the whole page. The core API to achieve this was the `XMLHttpRequest` API.

While the web technologies and browsers advanced, JavaScript grew more prominent as a web scripting language allowing developers to access the DOM and dynamically display and interact with the content presented on the web page. However, JavaScript is also single-threaded by nature, which means that, at any given time, any two lines of script cannot run together; instead, JavaScript statements are executed line by line. Likewise, in browsers, JavaScript shares that single thread with a bunch of other workloads executed by the browser from painting and updating styles to handling user actions. One activity will delay the other.

When it first started, JavaScript was intended for short, quick-running pieces of code. Major application logic and calculations were done on the server side. Ever since loading content on the web changed from reloading the whole page to the client side, asynchronous loading developers started relying more heavily on JavaScript for web development. Now, we can find complete application logic being written with JavaScript, and so many libraries have flourished to help developers do so.

In web development, we have the following three main components:

- HTML and CSS
- The Document Object Model (DOM)
- JavaScript

And I will add a fourth component that plays a pivotal role in AJAX programming:

- The XMLHttpRequest API

Briefly, HTML and CSS are used for the presentation and layout of a web page. The DOM is used for the dynamic display and interaction with content. The XHR object sends HTTP/HTTPS requests to a web server and loads the server response data back into the script, mediating an asynchronous communication. Lastly, JavaScript allows developers to bring all these technologies together in order to create beautiful, responsive, and dynamic web applications.

In order to tackle the multithreading limitation, developers relied heavily on events and callbacks because that is the way browsers expose asynchronous programming to the application logic.

In event-based asynchronous APIs, an event handler is registered for a given object and the action is called when the event is fired. The browser will perform the action usually in a different thread, and triggers the event in the main thread when appropriate.

We can achieve this event-based technique using the `object.addEventListener()` method. This method will simply register a listener on the target object it is called on. The event target object may be an element in an HTML document, the document itself, a window, or any other object that supports events (such as XHR).

The following code shows what a simple event listener looks like using HTML and JavaScript.

The HTML part is as follows:

```
<div id='testDiv' style="width:100px; height:100px;
  background-color:red">/</div>
```

The JavaScript part is as follows:

```
var testDiv = document.getElementById("testDiv");
testDiv.addEventListener("click", function(){
   // so the testDiv object has been clicked on, now we can do things
    alert("I'm here!");
});
```

In the HTML part, we define a `div` element in the DOM with the `testDiv` ID. In the JavaScript part, we retrieve the `div` element in the first line of the code and assign it to a variable. Then, we add an event listener on that object and pass it to the `click` event followed by an anonymous function (a function without a name) as the listener function. This function will be invoked later in response to a click event on the element.

> If you add this JavaScript code before the HTML markup that includes the `div` element, it will raise an error. Since the element is not created yet when the code executes against it, the code will not find the target object to call `addEventListener` on.

As we can see in the previous code sample, the second parameter to the `addEventListener` method is a function in itself that contains some inline code. We are able to do so in JavaScript because functions are first-class objects. This function is a callback. Callback functions are *super* important and widely spread in JavaScript programming because they give us the ability to do things asynchronously.

Passing a callback function as a parameter to another function is only passing the function definition. Hence, the function is not executed immediately in the parameter; it is *called back* (hence the name) at some specified point inside the container function's body. This is very useful for scripts that take some time to complete actions such as making an AJAX request to the server or performing some IO activity without holding up the browser along the way.

> If you're new to JavaScript, seeing functions passed as parameters might be somewhat unfamiliar, but don't worry; it becomes easy when you think of them as objects.

Some browser APIs such as HTML5 Geolocation are called back based on design. I will use Geolocation's `getCurrentMethod` to use a callback function in an example. The code will look like the following:

```
navigator.geolocation.getCurrentPosition(function(position){
  alert('I am here, Latitude: ' + position.coords.latitude + ' ' +
                 '/ Longitude: ' + position.coords.longitude);
});
```

In the previous example, we simply called the `getCurrentPosition` method and pass it an anonymous function, which in turn invokes an alert method that will get called back with the result we requested. This allows the browser to execute this code synchronously or asynchronously; thus, the code will not be blocking the browser while the position is being retrieved.

In this case, we used a built-in browser API, but we can also make our applications asynchronous-ready by exposing their basic APIs in an asynchronous manner with callback functions at least the ones involved in an I/O operation or in heavy computing, which might take a great deal of time.

For example, in a callback scenario, the simplest code to retrieve some data would look like the following:

```
getMyData(function(myData){
   alert("Houston, we have : " + myData);
});
```

In the previous JavaScript code, we just defined a `getMyData` function that takes a callback function as a parameter, which in turn executes an alert that displays the data we are supposed to retrieve. This code actually obliges with the application UI code to be asynchronous-ready; thus, the UI will not be blocked while the code is retrieving the data.

Let's compare it to a non-callback scenario; the code will look like the following:

```
// WRONG: this will make the UI freeze when getting the data
var myData = getMyData();
alert("Houston, we have : " + myData);
```

In the previous example, the JavaScript code will run line by line, and the next line of code will run even though the first one is not finished. Such an API design will make the code UI-blocking because it will freeze the UI until the data is retrieved. Furthermore, if the the execution of the `getMyData()` function happens to take some time, for example, fetching data from the internet, the overall experience will not be pleasant to the user because the UI will have to wait for this function to finish executing.

Moreover, in the previous examples of callback functions, we passed an anonymous function as a parameter of the containing function. This is the most common pattern of using callback functions. Another way to use callback functions is to declare a named function and pass the name of that function as a parameter. In the following example, we will use a named function instead. We will create a generic function that takes a string parameter and displays it in an alert. We will call it `popup`. Then, we will create another function and call it `getContent`; this takes two parameters: a string object and a callback function. Lastly, we will call the `getContent` function, and pass it a string value in the first parameter and the callback function `popup` in the second. Run the script and the result will be an alert that contains the value in the first string parameter. The following is a code sample for this example:

```
//a generic function that displays an alert
    function popup(message) {
    alert(message);
    }
//A function that takes two parameters, the last
  one a callback function
    function getContent(content, callback) {
        callback(content); //call the callback function
    }
getContent("JavaScript is awesome!", popup);
```

As we can see in the previous example, we were able to pass a parameter to the callback function since, at the end of the day, it is just a normal function when it is executed. We can pass to the callback function any of the containing function's variables as parameters or even global variables from elsewhere in the code.

To summarize, JavaScript callback functions are powerful and have contributed greatly to the web development environment, thus allowing developers to have asynchronous JavaScript programming.

Why should I care about promises?

What do promises have to do with all of this? Well, let's start by defining promises.

> *A promise represents the eventual result of an asynchronous operation.*
>
> *-Promises/A+ specification,* `http://promisesaplus.com/`

So, a promise object represents a value that may not be available yet, but will be resolved at some point in the future.

Promises have states and at any point in time, can be in one of the following:

- **Pending**: The promise's value is not yet determined and its state may transition to either fulfilled or rejected.
- **Fulfilled**: The promise was fulfilled with success and now has a value that must not change. Additionally, it must not transition to any other state from the fulfilled state.
- **Rejected**: The promise is returned from a failed operation and must have a reason for failure. This reason must not change and the promise must not transition to any other state from this state.

A promise may only move from the pending state to the fulfilled state or from the pending state to the rejected state. However, once a promise is either fulfilled or rejected, it must not transition to any other state and its value cannot change because it is immutable.

> The immutable characteristic of promises is *super* important. It helps evade undesired side effects from listeners, which can cause unexpected changes in behavior, and in turn allows promises to be passed to other functions without affecting the caller function.

From an API perspective, a promise is defined as an object that has a function as the value for the property `then`. The promise object has a primary `then` method that returns a new promise object. Its syntax will look like the following:

```
then(onFulfilled, onRejected);
```

The following two arguments are basically callback functions that will be called for completion of a promise:

- `onFulfilled`: This argument is called when a promise is fulfilled
- `onRejected`: This argument is called when a promise has failed

Bear in mind that both the arguments are optional. Moreover, non-function values for the arguments will be ignored, so it might be a good practice to always check whether the arguments passed are functions before executing them.

> It is worth noting that, when you research promises, you might come across two definitions/specifications: one based on Promises/A+ and an older one based on Promises/A by CommonJS. While Promises/A+ is based on the concepts and then API presented in the CommonJS Promises/A proposal, A+ implementation differs from Promises/A in several ways, as we will see in *Chapter 2, The Promise API and Its Compatibility*.

The new promise returned by the `then` method is resolved when the given `onFulfilled` or `onRejected` callback is completed. The implementation reflects a very simple concept: when a promise is fulfilled, it has a value, and when it is rejected, it has a reason.

The following is a simple example of how to use a promise:

```
promise.then(function (value){
    var result = JSON.parse(data).value;
    }, function (reason) {
    alert(error.message);
});
```

The fact that the value returned from the callback handler is the fulfillment value for the returned promise allows promise operations to be chained together. Hence, we will have something like the following:

```
$.getJSON('example.json').then(JSON.parse).then(function(response) {
    alert("Hello There: ", response);
});
```

Well, you guessed it right! What the previous code sample does is chain the promise returned from the first `then()` call to the second `then()` call. Hence, the `getJSON` method will return a promise that contains the value of the JSON returned. Thus, we can call a `then` method on it, following which we will invoke another `then` call on the promise returned. This promise includes the value of `JSON.parse`. Eventually, we will take that value and display it in an alert.

Can't I just use a callback?

Callbacks are simple! We pass a function, it gets invoked at some point in the future, and we get to do things asynchronously. Additionally, callbacks are lightweight since we need to add extra libraries. Using functions as higher-order objects is already built into the JavaScript programming language; hence, we do not require additional code to use it.

However, asynchronous programming in JavaScript can quickly become complicated if not dealt with care, especially callbacks. Callback functions tend to become difficult to maintain and debug when nested within long lines of code. Additionally, the use of anonymous inline functions in a callback can make reading the call stack very tedious. Also, when it comes to debugging, exceptions that are thrown back from within a deeply nested set of callbacks might not propagate properly up to the function that initiated the call within the chain, which makes it difficult to determine exactly where the error is located. Moreover, it is hard to structure a code that is based around callbacks as they roll out a messy code like a snowball. We will end up having something like the following code sample but on a much larger scale:

```
function readJSON(filename, callback) {
    fs.readFile(filename, function (err, result) {
        if (err) return callback(err);
        try {
            result = JSON.parse(result, function (err, result) {
                fun.readAsync(result, function (err, result) {
                    alert("I'm inside this loop now");
                    });
                alert("I'm here now");
                });
            } catch (ex) {
        return callback(ex);
        }
    callback(null, result);
    });
}
```

> The sample code in the previous example is an excerpt of a deeply nested code that is sometimes referred to as the *pyramid of doom*. Such a code, when it grows, will make it a daunting task to read through, structure, maintain, and debug.

Promises, on the other hand, provide an abstraction to manage interactions with asynchronous APIs and present a more managed approach towards asynchronous programming in JavaScript when compared to the use of callbacks and event handlers. We can think of promises as more of a pattern for asynchronous programming.

Simply put, the promises pattern will allow the asynchronous programming to move from the continuation-passing style that is widespread to one where the functions we call return a value, called a promise, which will represent the eventual results of that particular operation.

It allows you to go from:

```
call1(function (value1) {
    call2(value1, function(value2) {
        call3(value2, function(value3) {
            call4(value3, function(value4) {
                // execute some code
            });
        });
    });
});
```

To:

```
Promise.asynCall(promisedStep1)
.then(promisedStep2)
.then(promisedStep3)
.then(promisedStep4)
.then(function (value4) {
    // execute some code
});
```

If we list the properties that make promises easier to work with, they will be as follows:

- It is easier to read as with the usage of cleaner method signatures
- It allows us to attach more than one callback to a single promise
- It allows for values and errors to be passed along and bubble up to the caller function
- It allows for chaining of promises

What we can observe is that promises bring functional composition to synchronous capabilities by returning values, and error bubbling by throwing exceptions to the asynchronous functions. These are capabilities that we take for granted in the synchronous world.

The following sample (dummy) code shows the difference between using callbacks to compose asynchronous functions communicating with each other and promises to do the same.

The following is an example with callbacks:

```
$("#testInpt").click(function () {
    firstCallBack(function (param) {
        getValues(param, function (result) {
            alert(result);
        });
    });
});
```

The following is a code example that converts the previous callback functions to promise-returning functions that can be chained to each other:

```
$("#testInpt").clickPromise()  // promise-returning function
.then(firstCallBack)
.then(getValues)
.then(alert);
```

As we have seen, the flat chains that promises provide allow us to have code that is easier to read and eventually easier to maintain when compared to the traditional callback approach.

Summary

Callbacks in JavaScript allow us to have a user interface that is more responsive by responding asynchronously to events (that is, user input), without blocking the rest of the application. Promises are a pattern that allows for a standardized approach in asynchronous programming, which enables developers to write asynchronous code that is more readable and maintainable.

In the next chapter, we will take a look at the browsers that support promises and their compatibility with jQuery. You will also learn about libraries that support promise-like functionalities.

2

The Promise API and Its Compatibility

Promises are fairly new to the JavaScript world, but workarounds have been around for some time now. As we have seen in the previous chapter, there are ways to address asynchronous programming in JavaScript, be it through events or callbacks. You also learned why promises differ from the traditional techniques for that purpose.

Next, we will go into more details of the Promise API. You will also learn about the current browser support for the promises standard and take a look at the JavaScript libraries out there that implement promises and promise-like features. In this chapter, we will cover the following topics:

- The Promise API and its details
- Browser compatibility
- Promise implementations
- Libraries with promise-like features

Getting to know the API

Throughout this book, we will be mostly addressing and using promises as defined in the specification of Promises/A+ (`http://promisesaplus.com/`). The Promises/A+ organization produced the Promises/A+ specification with the aim of expounding the initial Promises/A specification into one that is clearer and better tested. The following is a quote from their website:

> *Promises/A+ is based on the concepts and* `then` *API presented in the CommonJS Promises/A proposal.*
>
> *- http://promisesaplus.com/differences-from-promises-a*

These differences are seen at three levels: omissions, additions, and clarifications. At the level of omissions, Promises/A+ has removed the following features from the original one:

- **Progress handling**: This feature includes a callback function that handles the operation/promise while still in progress, as in not fulfilled nor rejected. It was removed because implementers have concluded that, in practice, these functionalities have proven to be underspecified and currently there is no total agreement on their behaviors within the promise implementer community.
- **Interactive promises**: This feature was an extended promise in the previous Promises/A proposal and it basically supported two additional functions for the promise methods; `get(propertyName)`, which requests the given property from the target of this promise, and `call(functionName, arg1, arg2, ...)`, which calls the given method/function in its parameter on the target of the promise. In the new A+ specification, this feature, along with the two functions, `call` and `get`, is considered out of scope when implementing a basic API required for interoperable promises.
- `promise!== resultPromise`: This feature was a requirement in the old proposal, which states that the result of a promise should not equal the promise, for example, `var resultPromise = promise.then(onFulfilled, onRejected)`. In fact, any implementation may allow for `resultPromise === promise`, provided that the implementation meets all the requirements.

At the level of additions, the Promises/A+ specification adds the following features and requirements to the existing Promises/A proposal:

- The behavior specifications in the scenario where `onFulfilled` or `onRejected` returns a thenable, including the details of the resolution procedure.
- The reason passed to the `onRejected` handler, which must be the exception that is thrown back in that case.

- Both handlers `onFulfilled` and `onRejected` that must be called asynchronously.
- Both handlers `onFulfilled` and `onRejected` that must be called functions.
- Implementations must abide by the exact ordering of calls to the handlers `onFulfilled` and `onRejected` in case of consequent calls to the `then` method on the same promise. In a more spoken language, this means that, if the `then` method is called more than once on the same promise as in `promise.then().then()`, all the `onFulfilled` handlers used in these `then` calls must execute in the order of the originating calls to `then`. Hence, the `onFulfilled` callback in the first `then` function will execute first, followed by the `onFulfilled` callback in the second `then`, and so on. The same thing applies to the execution of the `onRejected` callbacks in such a scenario. Was it very complex? Maybe the following example can explain it better:

  ```
  var p = [[promise]];
  p.then();
  p.then();
  ```

  The preceding code is not the same as the following line of code:

  ```
  promise.then().then();
  ```

  The difference is that `promise.then()` might return a different promise.

Lastly, at the level of clarifications, the Promises/A+ proposal applies different naming from Promises/A, because the authors of the new specifications wanted to reflect the vocabulary that has spread among promise implementations. These changes include the following:

- The promise states are referred to as pending, fulfilled, and rejected, replacing unfulfilled, fulfilled, and failed
- When a promise is fulfilled, the promise has a *value*; similarly, when a promise is rejected, it has a *reason*

The `then` method is the main player in the API. An object is not considered a promise if it does not have the `then` method specified to retrieve and access its current or eventual value or reason, as we saw in the previous chapter. This method takes two arguments that need to be a function, as the following example shows:

```
promise.then(onFulfilled, onRejected);
```

Let's dive deep into the details of `then` and the specs of its arguments, taking into consideration the previous code sample of a simple `then` method:

- Both the arguments, `onFulfilled` and `onRejected`, are optional.
- Both the arguments must be a function; otherwise, it must be ignored.

- Both the arguments must not be called more than once within the same `then` call.
- The `onFulfilled` argument must be called only after a promise is fulfilled, with the value of the promise as its first argument.
- The `onRejected` argument must be called after the promise is rejected, with the reason of promise rejection as its first argument.
- The `onFulfilled` and `onRejected` arguments must not be passed as a `this` value because, if we apply the strict mode to the JavaScript code, this will be treated as undefined inside the handlers; in the quirks mode, it will be treated as the global object in that JavaScript code.
- The `then` method can be called more than once on the same promise.
- When a promise is fulfilled, all the respective `onFulfilled` handlers must be executed in the same order as their originating calls to `then`. The same rule applies for the `onRejected` callbacks.
- The `then` method must return a promise as follows:

  ```
  promiseReturned = promise.then(onFulfilled, onRejected);
  ```

- If either `onFulfilled` or `onRejected` returns a value *x*, the promise resolution procedure must be called to resolve the value *x*, as the following code shows:

  ```
  promiseReturned = promise1.then(onFulfilled, onRejected);
  [[Resolve]](promiseReturned, x).
  ```

- If either the `onFulfilled` or `onRejected` handler throws an exception `e`, `promiseReturned` must be rejected with `e` as the reason of rejection or failure.
- If `onFulfilled` is not a function and the promise is fulfilled, `promiseReturned` must be fulfilled with the same value.
- If `onRejected` is not a function and `promise1` is rejected, `promiseReturned` must be rejected with the same reason.

The previous list was a detailed specification of the promise and `then` method as defined and specified in the Promises/A+ open standard. We talked about the promise resolution procedure in the previous list, but we don't know what it is yet. Well, the promise resolution procedure is basically an abstract operation that takes a promise and a value as arguments, and is indicated as follows:

```
[[Resolve]](promise, x)]
```

If *x* is a thenable, meaning that it is an object or a function that defines a `then` method, the `resolve` method will try to force a promise to assume the state of *x*, under the assumption that *x* behaves at least somewhat like a promise. Otherwise, it will fulfill the promise with the value *x*.

The technique that promise resolution procedure uses to handle thenables allows promise implementations to work reliably with one another, as long as that promise exposes a `then` method that is Promises/A+-compliant. Additionally, it also allows implementations to *integrate* non-standard implementations with reasonable `then` methods.

> *The Promise Resolution Procedure (PRP) is not a public API. It is intended to describe an important, yet abstract and internal/private procedure, where "procedure" here simply means "algorithm" rather than a concrete JavaScript function. A particular promise implementation may implement it however they feel is best.*
>
> *- Brian Cavalier, co-editor at Promises/A+ explains*

The promise resolution procedure allows us to have a correct implementation of `promise.resolve`. It is also necessary to guarantee a correct implementation of `then`. You might notice that there is no return value for the promise resolution procedure because it is an abstract procedure that may be implemented in any way the author of that particular promise implementation sees fit. Hence, the return value is left up to the implementer as long as it achieves the end goal, which is to put the promise into the same state as *x*. So, conceptually, it affects a state transition on the promise.

Although the implementation of the promise resolution procedure algorithm is left to the implementers, it has some rules of its own that we should abide by if we want to be compliant to the proposal when we need to run it. These rules are as follows:

1. If a promise and *x* refer to the same object, the promise should be rejected with a `TypeError` as the reason for the `onRejected` handler.
2. If *x* is a promise, we should take on its current state. This rule allows the use of implementation-specific behaviors to actually adopt the state of known-conformant promises. The following are the conditions:
    - If *x* is in the pending state, the promise must remain in the pending state until *x* is fulfilled or rejected
    - If/when *x* is fulfilled, the promise should be fulfilled with the same value that *x* has
    - If/when *x* is rejected, the promise should be rejected with the same reason that *x* was rejected with
3. If *x* is an object or a function, and not a promise, then the following is done:
    - When we want to call `then`, the method should be `x.then`. This is a defensive measure that is imperative to ensure consistency in the face of an `accessor` property. This has a value that could change whenever we retrieve it.

  - If retrieving the `x.then` property ends up with throwing an exception `e`, the promise should be rejected with `e` as the reason.
  - If `then` is a function, call it with *x* taking the value of `this`. The first argument should be `resolvePromise` and the second argument should be `rejectPromise`.
  - If `then` does not qualify as a function, directly fulfill the promise with *x*.

4. If *x* is neither an object nor a function, the promise should be fulfilled with *x*.

Let's have a look at the third rule. We saw that, if `then` is the function, the first argument should be `resolvePromise` and the second argument should be `rejectPromise` where the following rules apply:

1. If/when `resolvePromise` is called with a value *z*, the implementation must run `[[Resolve]](promise, z)`.
2. If/when `rejectPromise` is called with a reason *j*, the implementation must reject the promise with reason j.
3. If both the handlers `resolvePromise` and `rejectPromise` are called, or in the case of multiple calls to the same argument, the first call should take precedence, and any other subsequent calls are ignored.
4. If calling then results in throwing an exception e, we have two conditions:
   - If the `resolvePromise` or `rejectPromise` handlers have already been called, we should ignore `then`
   - If not, the implementation should reject the promise with `e` as the reason returned

The previous long list of rules acts as guidance for implementers. So, if you are implementing `then` in your Public API, these rules should apply to your algorithm to be complaint with the Promises/A+ standard specifications. I have asked Brian Cavalier about the need of PRP and he added the following:

> *One of the most important aspects of the PRP is that it has been carefully designed to allow different promise implementations to interoperate in a reliable way.*

Furthermore, the promise resolution procedure even allows correctness in the face of a non-compliant (and slightly dangerous) thenables. An example would be the use of a `resolve` function to convert jQuery's version of a promise, which doesn't comply with the A+ standard, to a really simple standard-conforming promise. The following code illustrates that implementation:

```
// an ajax call that returns jquery promise
var jQueryPromise = $.ajax('/sample.json');
//correct it and convert it to a standard conforming promise
var standardPromise = Promise.resolve(jQueryPromise);
```

At the end of the day, the core goal of Promises/A+ is to provide the minimum, most simple specification possible that will allow a reliable interoperation of different promise implementations, even in the face of hazards.

> Just to erase any confusion that might come out, the promise resolution procedure is not exactly the same as the `promise.resolve` method that some implementations provide in their public API.

In alignment with the core goal of the Promises/A+ standard, the Promises/A+ organization created a compliance test suite to test the compliance of a promise library or API implementation against the Promises/A+ specification. The compliance tests, which can be found at `https://github.com/promises-aplus/promises-tests`, check the correctness of the promise resolution procedure by testing `then`. These tests are also intended to provide more concrete guidance and evidence for whether the implementations meet the requirements and conform to standards.

Browser support and compatibility

JavaScript is tightly coupled with browsers and the same applies to promises because promises are not a standard in the previous version of ECMAScript and will be part of the new ECMAScript 6 release; they won't be supported across all the browsers. Moreover, promises can be implemented and we will witness several libraries offering promise-like features or exposing promise capabilities. In the remaining portion of this chapter, we will cover these two points that are essential when it comes to working with promises.

Checking the browser compatibility

As with any client-side technology, JavaScript has expressly been developed for use in a web browser in conjunction with HTML pages. It uses the browser to do the job, which is why it is a scripting language. Once the script is sent to the browser, it is then up to the latter to do something with it. There is a heavy dependency there; thus, browser compatibility is vital.

There are already implementations of promises in some browsers; at the time of writing this book, there is a small selection of browsers that support promises, as the following ECMAScript 6 compatibility table by Kangax shows:

Feature name	IE 10	IE 11	FF 24	FF 25	FF 27-28	FF 29	FF 30	FF 31	CH 30	CH 33	CH 34	CH 35	SF 6	SF 7	WK	OP 12	OP 15
Promise	No	No	No	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No	Yes	No	No

Source: http://kangax.github.io/compat-table/es6/#Promise

> **The abbreviations used in the compatibility table**
>
> IE stands for Internet Explorer, FF for Firefox, CH for Chrome, SF for Safari, WK for Webkit, and OP for Opera.

As we can see in the previous table, only the latest three versions of Firefox, as of version 29, and Chrome, as of 32, enable promises by default. Worry not, for there is a polyfill to add the promises functionalities to browsers that do not support it yet.

> A polyfill is a fairly new term coined by Remy Sharp and grew popular in the community of web developers. It represents a piece of code that delivers the technology and the behavior that we expect the browser to provide natively. We can think of it as a patch in terms of computing.

This polyfill that does the magic and provides us with the support for promises can be downloaded from this link: `https://www.promisejs.org/polyfills/promise-4.0.0.js`. It basically adds support for promises to browsers that don't yet implement it natively. It can also be used to provide support for promises in Node.js. The following code sample shows how to include it in our code files:

```
<script src="https://www.promisejs.org/polyfills/promise-4.0.0.js"></
script>
```

We are showing the ECMAScript 6 compatibility table because promises are part of the ECMAScript 6 specification, which provides promises as a first-class language feature, and the implementation is based on the Promises/A+ proposal.

Libraries with promise-like features

The notion of promises is not very new to the world of web development and JavaScript. Developers may have met or used promises in JavaScript in a non-standardized manner through libraries. These libraries are implementations of the promise concept; some of them are spec-adhering implementations and are starting to take on the promise pattern, while many are not. Moreover, some of these libraries do not conform to the Promises/A+ standard, which is a very important requirement when choosing what JavaScript libraries to use in our projects.

> Developers can test whether their libraries and APIs implementing promises are conforming to the Promises/A+ standard by using the Compliance Test Suite.

The following is a list of some libraries that are fully compliant with Promises/A+ specs, and that I can thus unreservedly recommend:

- **Q.js**: Developed by Kris Kowal and Domenic Denicola, it encompasses a full-featured promise library that includes adapters for Node.js and support for progress handlers. It can be downloaded from `https://github.com/kriskowal/q`.
- **RSVP.js**: Developed by Yehuda Katz, it features a very small and lightweight promise library. It can be downloaded from `https://github.com/tildeio/rsvp.js`.
- **when.js**: Developed by Brian Cavalier, it offers an intermediary library and includes functions to manage collections of eventual operations. It also features functions that expose the progress and cancellation handlers of a promise. It can be downloaded from `https://github.com/cujojs/when`.

In addition, we have `then` (`https://github.com/then`) that is a collection of libraries that are simple Promises/A+ implementations that meet the specification and extend it with some functionalities such as progress while a promise is fulfilled or rejected.

Also, the famous jQuery has an API they called Deferred— available at `http://api.jquery.com/jquery.deferred/`, which claims to be similar to a promise. jQuery's deferred didn't return a new promise from `then` as the specification necessitates until version 1.8; hence, developers relying on jQuery were not getting the full capability and power of the promises pattern. Furthermore, a lot of code written using this implementation doesn't piece perfectly with other promise implementations that did actually adhere to the specification. Deferreds are not Promise/A+-compliant, at least with the second part of the specification, which states that `then` doesn't return a new promise object when executing one of the handlers. Hence, we cannot have function composition and chaining of the `then` function and ultimately, error bubbling due to a broken chain, which are the two most important points in the specification. This makes jQuery different and somewhat less useful. Nevertheless, if we need to use the `promise` object exposed by jQuery or any other library that does not conform to the specification for that matter, we can use one of the libraries listed earlier to convert that non-conforming promise to a real promise that is complaint with the A+ proposal. For example, using Q, we can have the following code that converts a jQuery promise to a standard one:

```
var SpecPromise = Q.when($.get("http://example.com/json"));
```

Another example would be using the Promise polyfill library (`https://www.promisejs.org/polyfills/promise-4.0.0.js`) as the following code shows:

```
var specPromise = Promise.resolve($.ajax(
  ' http://example.com/json);););
```

Although these promise implementations follow a standardized behavior, their overall APIs differ.

Summary

As we have seen, the concept of promises is not very new and has been around in JavaScript with different implementations through libraries, be it standard-complaint or others. However, now, all these efforts have concluded in the Promises/A+ community specification that most libraries conform to. Thus, we now have native support for promises in JavaScript via a standard `Promise` class that is included in the next version of ECMAScript ECMAScript 6, allowing web platform APIs to return promises for their asynchronous operations. Also, we covered the promise API and the `then` method in depth and learned about the current browser compatibility for the new standard. Finally, we briefly went over some of the libraries that implement promises and are compliant with the Promises/A+ specification

In the next chapter, we will go over chaining of promises and how to achieve it using the `then` method to enable multiple asynchronous actions.

3
Chaining of Promises

One of the most important features of promises is the ability to chain and manage sequences of asynchronous operations. In the previous chapter, we learned the details of the Promise API and how it works; notably, we saw how the `then` method works. We also learned about the current browser compatibility for promises and the libraries that have implemented and extended JavaScript promises. In this chapter, we will cover the following topics:

- How chaining came to be in an asynchronous JavaScript
- Implementing chaining with promises
- Transforming from callback hell into well-organized promise chains

Chaining like never before

As we learned in the previous two chapters, promises tend to bring the prowess of synchronous programming to asynchronous functions. This ability of promises includes two key features of synchronous functions:

- A function that returns values
- A function that throws exceptions

The significance of these features is that they can be used to pass the value returned by one function directly into another—and not just once; this can be translated into the ability to chain these functions one after the other, whereby the binding association between the elements in this chain is the promise's return value by each operation. Now, what the second feature implies is very important as throwing exceptions allows us to primarily detect whether the process has failed; secondly, it allows us to capture those exceptions by any function that handles a catch in the chain and helps us to avoid losing it in the midst of these chained functions.

Now, how does this translate into an asynchronous world?

Well, to begin with, in an asynchronous world, one cannot simply return values because those values are not yet ready in time. Likewise, we cannot throw exceptions, basically because there is no one there to catch those raised exceptions. So developers, ever resourceful, have tried to solve this problem by reverting to nested callbacks. This allowed them to chain functions with return values, but at the cost of maintainability, readability, and of course, extra lines of code. When the code grows in lines and the nested callback grows in depth, the code becomes harder to maintain and debug when edits are needed or errors arise. Also, readability is negatively impacted with nested callbacks, and developers need to collapse and expand braces in order to follow with the code to tell where the callback function begins and ends.

Moreover, catching errors in these nested callbacks was very strenuous and required developers to pass the errors up the chain of callbacks manually. This ordeal in asynchronous programming is in famous and termed *callback hell*; it usually ends up in a code that looks like the following dummy code:

```
function shout(shoutTxt, callbackFunct) {
    alert(shoutTxt);
    callbackFunct("b");
}

shout('First Shout!', function (a) {
    if (a == "a"){
        alert("hey, there is an error!");
    }
    else {
        shout('Shout Again!', function (a) {
            shout('Third shout!', function (a) {
                a = "c";
                if (a == "c") {
                    shout('I am inside the third shout!',
                      function (a) {
                        alert("hey, I can " + a.toString());
                    });
                } else {
                    shout('I am still inside the third shout!',
                      function (a) {
                        alert("Alright I am tired");
                    });
                }
            });
        });
    }
});
```

In the previous example, you will notice the widespread presence of `function` and `});` in what looks like a pyramid of code, bearing in mind that we didn't even include error-handling code. The previous example depicts, on a small scale, what *callback hell* looks like. We can also observe how nested callbacks—which are quite flourishing in JavaScript programing—can grow uncontrollably into an intertwined and hard–to-maintain code. So imagine how the code will look in a more complex scenario.

Nevertheless, there are remedies that developers can implement to have nested callbacks that are more readable and maintainable. These remedies include the use of named functions instead of anonymous functions in the callback parameters. Another solution would be to break down the code into smaller chunks by placing the code that does a specific task into a separate module and then plug that module into the application code somewhere else. However, these remedies are more of a workaround and not a standard practice; also, the workarounds are still not enough to address the concept of chaining asynchronous operations entirely.

Promises, on the other hand, deliver the functional composition that we have in synchronous programming in more of an *out-of-the-box* way when compared with asynchronous programming in JavaScript.

Why so? Because the specification states that a promise is required to provide a `then` method. Not just that; the specification also necessitates that the `then` function, or any other function that has a compliant implementation, should return a promise. The returned promise contains either a value, if fulfilled, or an exception, if rejected. Hence, `then` can combine with another `then` function using the returned promise to compose a chain whereby the result of the first operation will be passed on to the next one and so on. Also, this chain can be cut at any point in time by a rejection, which can be handled by any operation in the chain that declares an exception-handling code; in other words, the error will bubble up automatically through that chain.

> Some promise enthusiasts regard this chaining of promises as being the best part about the new standard.

In JavaScript programming, chaining is very important when we have a scenario where multiple asynchronous operations need to be executed. These scenarios include the case where the work of one operation depends on the outcome of the previous operation. Moreover, we might have the case where the first operation needs to process some code before it can return a result and pass it to the next operation. Bear in mind that all of this should take place without blocking other threads, especially the UI thread. Hence, we need a straightforward, standard mechanism to chain these asynchronous operations, and this is exactly what promises provide.

When it comes to chaining promises, the chain can go as deep as we want to as `then` will always return a promise. One thing to watch out for, though, if we are making a call such as `promise.then(onFullfilled)`, is that the `onFulfilled` function can only be called after the promise has run its course, with the promise's value as its first argument. Therefore, if we return a simple value inside the first `then` and chain it to another one, the next `then` will be called with that simple value returned by the previous `then`. If we were to return a promise from the first `then`, the following `then` will have to wait for the returned promise and will only be called or executed when that promise has been fulfilled or completed.

Let's see this in action. The following is a very basic sample code that demonstrates a chained promise:

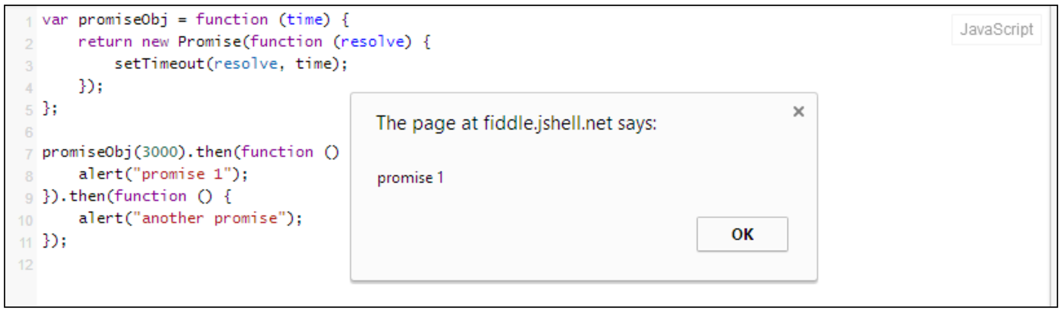
```
var promiseObj = function (time) {
    return new Promise(function (resolve) {
        setTimeout(resolve, time);
    });
};

promiseObj(3000).then(function () {
    alert("promise 1");
}).then(function () {
    alert("another promise");
});
```

The script is very straightforward, and you can write it in any development environment or even in an online code editor such as JSFiddle.net. First, we create a promise by defining a `promiseObj` object. This object is a function that takes one argument at a time and returns a new promise.

> Remember that not all browsers support promises as of now, as we learned in *Chapter 2*, *The Promise API and Its Compatibility*. To do this, you will need to run or test the code on jsFiddle in a compatible browser. Revert to this chapter to check for compatible browsers.

We construct the promise using `new Promise`. The constructor takes an anonymous function that will execute the work. This function is passed with one `resolve` argument that will fulfill the promise. Inside this constructor, we call the `resolve` argument to execute a `setTimeout` function that takes another `time` argument besides the function that will be executed after a given time. Hence, `setTimeout` will resolve the promise.

The second part of the code is where the chaining takes place. We first call the `promiseObj` we just created; since it will return a promise, we can call `then` on it. By definition, `promiseObj` takes the `time` argument, in milliseconds, to be passed on to the `setTimeout` function. Here, we passed `3000` (3 seconds) and inside it, we simply called an `alert()` function that will pop up on the screen as the following screenshot shows:



Now, since `then` returns a promise, we can chain another `then` call to it; this will be executed after the promise has been resolved and in turn will execute an `alert()` function. The previous example, though very basic, illustrates how we can easily chain asynchronous operations with promises.

Let's try converting the example we saw earlier from nested callbacks into a chain of promises. For illustration purposes, I will add an HTML element, `div`, to populate it with the content as the promise propagates.

The HTML part is as follows:

```
<div id="log"></div>
```

The JavaScript part is as follows:

```
var log = document.getElementById('log');
var shout = new Promise(function (resolve) {
    log.insertAdjacentHTML('beforeend', '(
    <small>Promise started </small>)<br/>');
    window.setTimeout(
    function () {
        resolve('First Shout!'); // fulfill the promise !
    }, 2000);
});


shout.then(function (val) {
```

```
        log.insertAdjacentHTML(
          'beforeend', val + '  (<small>Promise
          fulfilled</small>)<br/>');
        var newVal = 'Shout Again!';
        return newVal;
    }).then(function (val) {
        log.insertAdjacentHTML('beforeend', val + ' (
          <small>Promise fulfilled</small>)<br/>');
        var newVal2 = "Third shout, you're out!";
        return newVal2;}).then(function (val) {
        log.insertAdjacentHTML('beforeend', val + '
          (<small>Promise fulfilled</small>)<br/>');
        return val;
    });
```

In HTML, we only have a `div` element that is empty and has the ID log. In JavaScript, we first declare a variable called `log` to hold the `div` element. Then, we construct a new promise and assign it to a variable called `shout`. Inside that promise object, we add text to emphasize that we just started the promise. What we are promising here is the `shoutText` string after waiting for 2 seconds (2000 ms). Again, we used the `window.setTimeout` function to simulate an asynchronous operation that needs some time to finish. It will fulfill the promise by resolving it after a given time.

Next, we call `shout` with the `then` method inside which we have defined what it will do when the promise is fulfilled. In the first `then` method, we simply pass the `val` parameter, which contains the value of `shoutText`, to the `log.insertAdjacentHTML` function. This will display the value alongside the content of the `div` element that contains the text `Promise fulfilled` in a small font. Next, we define a new variable `newVal`, assign the text `Shout Again!` to it, and return it. Moving forward, the second `then` also displays the value returned from the previous promise call. We also define a new variable, assign it a text value, and return it. The last `then` call just adds the value of `val`, which by now is equal to `newVal2`, and adds it to the content of the `div` element. Note that `val` holds the content of the value returned by the promise from one operation to the next in the chain.

> This example can also be tested on JSFiddle.net.

Chaining in sequence

Not only can we chain asynchronous operations with promises, but we can also chain them in such a way that they run these operations in a sequence. As we learned earlier in this chapter, if a `then` operation returns a value, the subsequent `then` is called with that value unless the first `then` returns a promise; if this happens, the subsequent `then` will wait on the promise that is returned and will only be called when that promise gets fulfilled or rejected. This rule allows us to queue these asynchronous operations in such a way that each operation will wait for the previous one to finish and thus run in sequence. Let us look at an example that better explains it. In this example, we have a function called `getData` that takes a JSON file and retrieves data from that JSON file. The first JSON file has categories, and for each category, we need to get the items under each category in a sequence. Let's use the following code to do this:

```
getData(jsonCategoryUrl).then(function(data) {
//get the items per category 1
getItemsPerCategory(data.categories[0]).then(function(items)
{
        //items are retrieved here
});
  return getData (data.categories[0]); //return category 1
}).then(function(category1) {
   alert("We now have category 1", category1);
//return category 2
return getData (data.categories[1]);
}).then(function(category2) {
alert("We now have category 2", category2);
//return category 3
return getData (data.categories[2]);
});
```

The previous code sample makes an asynchronous call to the `jsonCategoryUrl` function, which will return a set of categories; after this, we request the items in the first of those categories by passing the `data.categories[0]` parameter and then passing on the first category to the next `then` call. In the second link of these chained promises, we retrieve the second category `data.categories[1]` and pass it to the last `then` call, which in turn retrieves the third category, `data.categories[2]`. This example shows us how we can queue asynchronous operations in chained promises if we need to have a chain in which one link depends or needs to wait on the result of the preceding promise.

This functionality really makes promises stand out from regular callback patterns. We can optimize the previous code by making a shortcut method to retrieve the categories, as shown in the following code:

```
//declare categorypromise var
var catPromise;
function getCategory(i) {
//if catPromise have no value get Data else just populate
  it from value of catPromise.

  catPromise = catPromise || getData(jsonCategoryUrl);
  return catPromise.then(function(category) {
      //get the items under that category
      return getData(category.Items[i]);
  })
}
getCategory(0).then(function(items) {
alert(items);
    return getCategory(1);
}).then(function(items) {
   alert(items);
});
```

In the previous code sample, we first declare a variable called `catPromise` to hold the category of the promise. Next, we declare a function called `getCategory(i)` that takes the values of `i` as a parameter; inside this function, and we set `catPromise` to the JSON data retrieved by the `getData(jsonCategoryUrl)` function; however, using the `||` (or) operator, we can first check whether the `catPromise` object has a value so that we don't fetch the category JSON file again, but only once. When we call `getCategory` with the value `0`, it will retrieve the first category; after this, it will return the next category with `getCategory(1)` and pass it to the last `then` call. In this way, we won't download the category JSON file until `getCategory` is called; however often we call the `getCategory` function again, though, we will not need to redownload the category JSON file; rather, we will reuse it since it will be called again in the sequence of operations. As the `getCategory` function returns another promise object, it allows you to have promise-pipelining, where we have the result from the first operation getting passed to the subsequent one. Also, the important feature that this sample shows is that, if the function provided to `then` returns a new promise, the promise returned by `then` will not be fulfilled until the promise returned by that function is fulfilled, thereby queuing the asynchronous operation in that chain of promises.

In addition, the previous sample clearly shows how promises address the traditional callback model and the pyramid code that it generates.

Summary

Promises represent a great way to address the intricacies of asynchronous operations. Promises provide a great mechanism for easy chaining of asynchronous operations in JavaScript. They allow you to manage the sequences of these operations in a better way than the callback mode does.

In the next chapter, we will learn about handling errors in promises, see how exceptions are managed with promises, and go over some examples on how to handle errors that arise during an asynchronous operation in promises.

4
Error Handling

As in any programming language, errors and exceptions are bound to rise; to ensure a smooth running code and easier debugging, we will need to throw and catch these exceptions. Handling errors with asynchronous JavaScript programming can be tedious. Promises, however, offer us a great mechanism to handle errors, which we will explore in this chapter. Throughout the previous chapter, we learned about the chaining of asynchronous operations. We also saw how we can transform from callback hell to the more readable and maintainable promise chains. In this chapter, we will cover the following topics:

- Exceptions and error handling in promise
- How to handle errors with promises using `then` and `catch` methods

Exceptions and promises

There is no standard or agreed-on mechanism to handle exceptions in asynchronous JavaScript programming, mainly due to the fact that these exceptions happen in the future and there is no way to tell if a rejected promise will eventually be handled. Moreover, in an asynchronous world, we can't just simply throw exceptions, because there is no one there to catch these errors when they are not ready yet. Hence, workarounds were created to address this issue. The common technique of catching errors and exceptions involved passing these exceptions manually up the chain of nested callbacks. Promises, on the other hand, provide us with error handling and bubbling out of the box. They do so by stating that your functions should return a promise that is rejected with a reason if it fails.

We learned in *Chapter 1, JavaScript Promises – Why Should I Care?* that a promise can exist in three different states: pending, fulfilled, and rejected.
The requirements for a rejected state are as follows:

- The promise must not change to any other state (pending or fulfilled)
- The promise must have a reason for being rejected, and that reason must not change within that promise

These two requirements of the rejected state allow error handling, and more importantly error composition, whereby the reason for the rejection of that promise will automatically bubble up that chain of promises using the `then` method. Promises allow errors to propagate up the chain of code similar to the synchronous exceptions. Moreover, it provides a cleaner style to handle errors in asynchrony.

Typically, in asynchronous programming using the Callback approach, we need to wrap the code block that we think is unsafe in a `try` catch block. This is shown in the following code sample:

```
try {
    return JSON.parse("json"); //this will cause an error
} catch (error) {
    alert("I have an error with the following details:
      \n" + error);
}
```

The previous code sample shows a block of script that intends to alert an error. In that block of code, we wrapped `return JSON.parse("json");` in a `try`...`catch` block and intentionally caused an error by passing it an invalid JSON parameter. The JavaScript function `JSON.parse()` is used to convert JSON text into a JavaScript object. In our example, it will try to parse the text `json` and will throw an error. We will catch that exception and display an alert with the details of that error.

If we run this script in an HTML page or online JavaScript Editor, the result will be an alert box with the following message:

I have an error with the following details:

SyntaxError: Unexpected token j

We can browse the code via this public jsFiddle URL at `http://jsfiddle.net/RamiSarieddine/mj6hs0xu/`

Promises, as we have seen so far, are either fulfilled or rejected if an error occurs in the promise. When a promise is rejected, it is similar to throwing an exception in synchronous code. A standard promise with a `then` function takes the two parameters `onFulfilled` and `onRejected` as the following code shows:

```
promise.then(onFulfilled, onRejected)
```

The `onRejected` parameter is a function that will act as an error handler, and it will be called when the promise fails. When an error or exception happens in a promise, it means that the promise was rejected, and the error raised will be provided to the `onRejected` error handler. There are a couple of considerations when we call `onRejected`, which can be summarized in the following list, assuming we have a simple `promise.then(onFulfilled, onRejected)`:

- `onRejected` must be called only after the promise had been rejected, with the rejection reason as its first argument
- `onRejected` must not be called more than once

The second consideration is very straightforward. The `onRejected` function does not get called multiple times on the same promise. The first consideration asserts that `onRejected` will not be called if a promise was rejected.

Nevertheless, rejections do happen implicitly as well as in cases where an error is thrown in the constructor callback of that promise. The following code sample illustrates this:

```
var promiseTest = new Promise(function (resolve) {
    // JSON.parse will throw an error because of invalid JSON
    // so this indirectly rejects
    resolve(JSON.parse("json"));
});

promiseTest.then(function (data) {
    alert("It worked!" + data);
}, function (error) { //error handler
    alert(" I have failed you: " + error);
});
```

In the previous code, we define a new promise called `promiseTest` and call `then` upon that promise. All that this promise does in its constructor callback is resolve `JSON.parse()`, to which we intentionally passed an invalid argument to cause an error. Now, this will throw an error in the constructor, and it will indirectly cause a rejection when we call the promise with a `then` method. If we only had an `onFullfilled` handler, we wouldn't have caught the error. The exception will be raised as an argument in the rejection with its value `error`. We provided an `error` handler in the arguments of `promiseTest.then()`; hence, we can catch and handle the error.

You can test this sample code via this public Fiddle at `http://jsfiddle.net/RamiSarieddine/x2Latjg6/`.

> As errors do automatically bubble up and become rejections, it becomes quite handy to process all the promise-related jobs inside the promise constructor callback; if any error arises there, it will be caught when the promise is called.

Handling errors with promises

As we have seen, promises offer a richer error handling mechanism in asynchronous programming. Although the Promises/A+ spec tackles only one method, that is `.then(onFulfilled, onRejected)`, and does not provide any other, the specifications of `.then()` lay the foundation for promise interoperability and, hence, extend the promise features, including error handling.

We might come across several implementations for error handling within JavaScript libraries that are compatible with Promises/A+. Some of these extensions include the `catch()` method, which is implemented on top of the elementary `then()` function. Anyone can author a `catch()` method and include it in their scripts by extending the promise object as per the following code:

```
Promise.prototype.catch = function(onRejected) {
    return this.then(null, onRejected);
};
```

In the previous code sample, we defined a method named `catch` that extends the current `this.then` method and returns a rejected promise by executing the `onRejected` handler only and neglecting the `onFulfilled` handler argument for `then`. In use, the `catch()` method will look like the following:

```
var promiseTest = new Promise(function (resolve) {
    resolve(JSON.parse("json"));
});

promiseTest.then(function (data) {
    alert("It worked: " + data)
}).catch(function(error) {
  alert("I have Failed you! " + error);
});
```

The `catch()` function allowed us to replace the error handler with a more readable function that provides a cleaner approach to handle the errors.

We can conclude from the previous code samples that there is nothing distinctive about catch, simply its sugarcoating `then(null, function)` function. Moreover, as Brian Cavalier, one of the authors of Promise/A+ specifications, puts it: `catch()` is simply a restricted subset of `then()`. But does it make the code in general and error handling specifically more readable? The ECMAScript 6.0 incorporates `catch()` as a requirement in the promise specification, and as I stated earlier, most of the popular implementations nowadays include it.

However, there is a gotcha between `then()` and `catch()` in implementation as `then()` tends to be somehow misleading at times. To see it in an example so as to better understand it, take the following two sample lines of code:

```
promise.then(handler1, handler2);

promise.then(handler1).catch(handler2);
```

These two lines of code include `promise`, `then`, and `catch` methods with two handlers: `handler1` and `handler2`. These two calls are not equivalent—the first line will not call `handler2` if an error occurs in `handler1`. That is because, if the promise is fulfilled, `handler1` will be invoked, and if the promise is rejected, `handler2` will be invoked. But if `handler1` throws `error`, `handler2` will not be called in return.

Meanwhile, in the second line, `handler2` will be invoked if either promise is rejected or `handler1` throws an exception. Since `catch()` is merely a sugarcoat for `then(null, handler)`, the second line is identical to the following, which can make this conundrum clearer:

```
promise.then(handler1).then(null, handler2);
```

The reason behind this nonequivalence in the previous two lines of code is the way `then()` operates. The `then(handler1, handler2)` methods register two parallel handlers for a promise, whereby either `handler1` or `handler2` will be called but never both. On the other hand, with `then(handler1).catch(handler2)`, both handlers/functions will be invoked if `handler1` rejects, because they represent two separate steps in the promise chain. Promise rejections will move forward to the succeeding `then` method with a rejection callback only when we have `catch` as an equivalent of `then`.

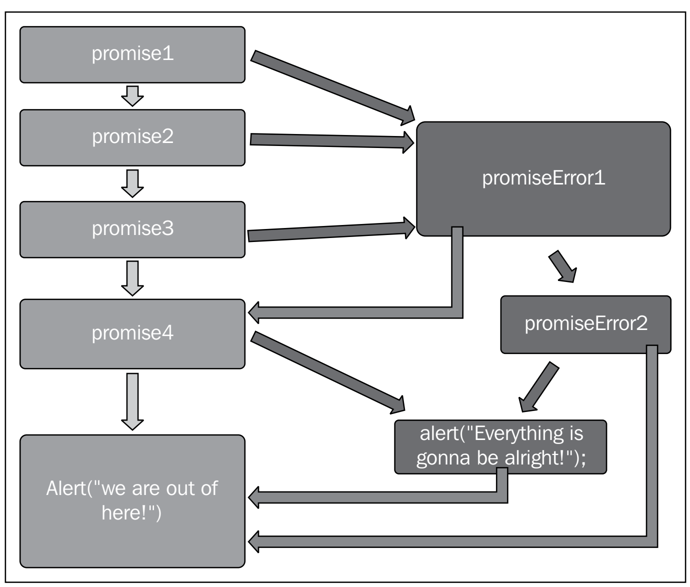
Although this does not seem very intuitive at first glance, it is very important to provide an easier reasoning about asynchronous programming and it makes rejecting a promise quite similar to throwing an exception in synchronous programming. In the synchronous world, exceptions do not allow for executing both the code that immediately follows a `throw` block and the code inside the closest `catch` block, whereby the errors that happen within a `try` block moves directly to the `catch` block.

> The `Catch()` function is preferable for application developers as we have learned, for better readability and its intuitive error handling flow, while `promise.then(handler1, handler2)` is mostly used internally when implementing a promise.

The error handling mechanism allows us to write functions that do things in a safe manner. Let us take the following chain of promises that includes error handling with `catch()` and see how it translates in a flowchart:

```
promise1.then(function () {
    return promise2();
}).then(function () {
    return promise3();
}).catch (function (error) {
    return promiseError1();
}).then(function () {
    return promise4();
}, function (error) {
    return promiseError2();
}).catch (function (error) {
    alert("Everything is gonna be alright!");
}).then(function () {
    alert("We are out of here!");
});
```

The corresponding flowchart for the preceding chain of promises and errors will look like the following diagram:



The boxes in green will be the ones that fulfill; the ones that are colored red represent the error handlers if the promise is rejected. We can follow the flow of the process with the lines to know which is fulfilled and which is rejected and compare it to the previous code sample to get a better visual idea of how the errors will propagate in that promise chain.

Summary

JavaScript's promises provide a standardized approach to error handling with the foundation of its implementation existing in the specifications of the `then` method that can be extended to breed methods such as `catch`, which allows a more readable and intuitive error handling code. The `then` function is provided with two powerful parameters: `onFulfilled` and `onRejected`. These function parameters allow us to handle the values returned from a promise operation that has been fulfilled and the errors returned when a promise is rejected. In the next chapter, we will cover the WinJS library; we will learn about the promises object in that library and how we can use it in Windows development.

5
Promises in WinJS

Promises have various implementations from a variety of frameworks, all of which share a common base; this is the concept of promises. Practically all of the promise libraries out there deliver a common feature in different forms to make asynchronous programming using JavaScript easier and better. WinJS, the Windows library for JavaScript, is one of the libraries that have their own implementation of promises, which we will explore throughout this chapter. In the previous chapter, we learned about handling exceptions that arise during promise operations. We also saw how JavaScript promises are equipped with a powerful error handling mechanism. Moreover, we learned how to handle errors with `then` and `catch` methods. In this chapter, we will cover the following topics:

- An introduction to the WinJS namespace
- The promise object of WinJS in full detail
- A basic example of using WinJS.Promise in Windows app development

Introducing WinJS

WinJS, which stands for the Windows library for JavaScript, is a JavaScript library developed by Microsoft, which was recently made open source. This library has been designed with the purpose of allowing developers to build Windows Store apps for Windows 8 (with HTML5 and JavaScript) that are first-class and native-quality experience, such as Skype and Music apps. It prevailed as the second option to programming the native apps with XAML and C#, VB.Net, or C++. This alternative allowed web developers to leverage their knowledge and skills to build store apps. WinJS library is more of a comprehensive toolkit. Not only does it provide a rich namespace, but it also includes the following features:

- Access to the device hardware via the Windows Runtime (WinRT)

- Delivers polished UI controls, such as ListView, FlipView, and Semantic Zoom alongside page controls
- Provides a solid infrastructure, such as promises and data-binding

Moreover, WinJS can be used in standalone solutions alongside other libraries and frameworks.

WinJS has evolved much since its initiation. What started as a platform specific to Windows Store apps now supports web browsers and other devices in an attempt to become cross-platform. That attempt is crystalizing with the latest version of WinJS 2.1, which supports Windows Phone 8.1, and now WinJS is being used in the Xbox One apps. Moreover, it is now prepared to cover websites and web apps on other non-Microsoft browsers and devices.

> The open source WinJS is now hosted on GitHub via `https://github.com/winjs/winjs/`, where community members can have a look at the library and contribute to its source code.

All WinJS library functions are defined under a namespace called WinJS. The WinJS namespace provides for special functionalities in the Windows Library for JavaScript that includes the promise object and `xhr` function. It includes three types of member objects: properties, and functions

Objects include the following two members:

- `ErrorFromName`: This is simply an error object.
- A promise object: This is our talk of the town in this chapter. Similar to the promise object, we have been discussing throughout this book, it basically provides a technique to assign work to be executed against a value that does not yet exist. It presents an abstract mechanism for handling interactions with APIs that are exposed asynchronously.

Properties include the following:

- `validation`: This property contains a setter to display the outcomes of a validation process

Functions include the following three members:

- `log`: This function logs output and writes it to the JavaScript console within Visual Studio. This function can be extended with a custom implementation or use `WinJS.Utilities.startLog` to log on to the JavaScript console.

- `strictProcessing`: This function is no longer needed, as strict processing is by default always turned on. With the function no longer needed, it has been declared deprecated.
- `xhr`: This function simply wraps the call to `XMLHttpRequest` in a promise.

This sums up the WinJS namespace from a high-level view; the code for WinJS is found in the `base.js` file.

Explaining the WinJS.Promise object

This object is one of the most important aspects of the WinJS library, and instances of promise are involved with anything we do with asynchronous APIs. Let us dive into the details of this object. In terms of anatomy, the promise object includes the following three types of members.

Constructors

At the level of constructors in WinJS, a class is created using the `WinJS.Class.define` function. In this first parameter is a function that acts as the constructor. Now, in the case of the `Promise` class, it is derived from a base class called `PromiseStateMachine` using the `WinJS.Class.derive` function, whose second parameter is the constructor function. In both cases, constructor functions can be named anything; alternatively, they can be anonymous as well. The description of this `WinJS.Promise` constructor, however, is the same as the object description itself. The `WinJS.Promise` constructor takes two function parameters: `init` and `onCancel`.

When declaring a new promise object, we need two parameters: `init` and `onCancel`. These two parameters are both functions. The syntax will look like the following:

```
var promiseObj = new WinJS.Promise(init, onCancel);
```

The `init` parameter is optional. The `init` function is called during the initialization or construction of the promise object, which comprises the actual implementation of the work that promise, in this case `promiseObj`, will represent. This implementation can be either asynchronous or synchronous, depending on the scope and nature of the work needed.

> One important thing to note here is that the code written within the `init` function does not render it by default to be asynchronous. In order to ensure the code runs asynchronously, we must use asynchronous APIs, such as the Windows Runtime asynchronous APIs, `setTimeout`, `setImmediate`, and `requestAnimationFrame`.

The `init` function used in this parameter takes the following three arguments:

- `completeDispatch`: This parameter will be invoked when the operation within `init` has been completed, thus passing the result of that operation The `init` code should invoke this when the operation is complete, passing the result of the operation as an argument.
- `errorDispatch`: This parameter will be invoked when an error occurs in that operation and, hence, the promise acquires an error state. Since it is an error, the argument to `errorDispatch` should be an instance of `WinJS.Promise.ErrorFromName`.
- `progressDispatch`: This parameter will be invoked, but periodically, while the operation is being processed. The argument to this function will contain an intermediate result. This parameter is used if the operation within the promise needs to support progress.

The `onCancel` parameter is the second parameter to the Promise constructor. This function can be used by the consumer of the promise to cancel any of its uncompleted work. However, promises are not obliged to provide or support cancellation in WinJS.

Events

Next on the list of promise object member types, we have `Events`. Currently, the Promise object has a single event called `onerror`. As the name shows, this event happens when an error occurs during the processing of a promise. Furthermore, this `onerror` event is fired whenever a runtime error is raised in any promise regardless of whether this event is handled somewhere else or not. The error handler can aid in debugging, where it can be used to set breakpoints and provide error logging. However, it will only provide insight and details on the code or input that caused the error at the end of the day. This `onerror` event delivers a general error-handling mechanism. In code, adding a generic error handler will look like the following:

```
WinJS.Promise.onerror = errorHandler;

function errorHandler(event) {
     // get generic error handling info
     var exc = event.detail.exception;
     var promiseErrored = event.detail.promise;
}
```

The first line of the code sample is simply attaching the `errorHandler` function to the `onerror` event of the promise object. Next, we define the `errorHandler` function, which takes an argument `event`; what the function does is simply retrieve information from the event in this example, such as the `exception` and the `promise`. Then, we assign the values to variables. The parameter `event` is an event handler argument of the type `CustomEvent`; usually it's an object that contains information about the event.

Methods

The last member type of the promise object is `Methods`, and currently `WinJS.Promise` has the following six methods:

- `addEventListener`: This method simply attaches and adds an event listener to the promise. It takes three parameters: `eventType`, which is the string type name of the event; `listener`, which is a function to be invoked when the event is triggered; and `capture` which is a Boolean value to enable or disable capture. This method has no return value and its basic syntax looks like the following:

  ```
  promise.addEventListener(eventType, listener, capture);
  ```

- `removeEventListener`: This method takes out an event listener from the control. In syntax, it is similar to the `addEventListener` method and looks like the following line of code:

  ```
  promise.removeEventListener(eventType, listener, capture);
  ```

- `Cancel`: This method tries to cancel the promise. In cases where a promise supports cancellation and hasn't been fulfilled yet, this method will cause the promise to enter the error state, with the value Error("Canceled"). It has no parameters and no return values. Its basic syntax is like the following:

  ```
  promise.cancel();
  ```

- `dispatchEvent`: This method simply dispatches and raises an event with the specified type and properties. It takes two parameters and returns a Boolean value depending on whether `preventDefault` was called on the event or not. The parameters of this method are of a string value type, containing the name of the event and `eventDetails`, an object which includes the set of additional properties to be attached to the event object. The basic syntax of this method looks like the following:

  ```
  promise.dispatchEvent(type, eventDetails);
  ```

- `Then`: This is the most important method of the promise object. This takes three parameters of the type function, which allows us to specify the work to be done on the completion of the promise: the promise value has been fulfilled; the error handling that will be performed when the promise raises an error, and it has failed to fulfill a value; and lastly the handling of the work progress within the promise along the way. The return value of `then` is a promise that contains the result of executing the `onComplete` function in its value. In its basic form, the `then` method will have the following syntax:

  ```
  promise.then(onComplete, onError, onProgress);
  ```

  The three arguments of the `then` method are of the type function. These are as follows:

  - `onComplete`: This handler will be called when the promise is completed successfully and fulfilled with a value. The value will be passed as a
    single argument. The value returned from `onComplete` becomes the fulfilled value of the promise returned by `then`. In the case of an error or exception during the execution of this function, the promise returned by `then` will enter the error state.
  - `onError`: This handler will be called when the promise breaks and it is fulfilled with an error; the value returned from `onError` will become the value of the promise returned by the `then` method. Instead of passing a value as in the `onComplete` function, here the error will be passed as an argument.
  - `onProgress`: This handler is used if we need to report the progress of a promises operation. It has a single argument, which is the progress data. Note that promises are not obliged to support progress in WinJS.

- The `Done` method, like `Then`, also allows us to specify what needs to be executed when a promise has been fulfilled, the error handling when a promise fails, and the reporting of progress along the way. In addition, this function will throw any error that would have been returned from `then` as a value for the promise in the error state. Unlike `then` which returns a promise, `Done` does not return a value. The basic syntax of this method looks like the following line of code:

  ```
  promise.done(onComplete, onError, onProgress);
  ```

As we can see in the previous code syntax, `promise.done` is similar to `promise.then` in terms of parameters as it has the function parameters: `onComplete`, `onError`, and `onProgress` that practically behave and do the same thing as their counterparts in the `Then` method.

There are some differences between `then` and `done`; the most obvious one is the return value. As stated earlier, the `then` method returns the promise while `done` has no return value, which has a direct effect on the chaining of WinJS promises. The following list summarizes those differences:

- **In chaining**: `Then` allows for chaining multiple `then` functions, because it returns a promise. While with `done` we cannot chain more than one `done` method because it does not return a value; more specifically, it returns `undefined`. Hence, `done` must be the final call. For example, we can have `.then().then().then().then()`, and so on, while with `done` it ends at `.then().then().done()`.
- **In error handling**: If there was no error handler provided to `done` and an error occurs (in other words, an unhandled exception), an exception will be thrown to the event loop allowing us to catch it in the `window.onerror` event, but not within a `try`/`catch` block. Thus, the `done` function guarantees to the caller method to throw any error that is not handled inside the method. While, with `then`, those unhandled exceptions that arise are silently caught and traversed as part of the promise state, `then` does not throw an exception but instead returns a promise that is in the error state.

Knowing the difference between these two methods is critical to using them. Nevertheless, for both methods, it is recommended to adopt flat promise chains in favor of nested ones, as the formatting of promise chains makes them easier to read and easier to handle errors. For example, the following sample code is preferable:

```
asyncFunct()
    .then(function () { return asyncFunct1(); })
    .then(function () { return asyncFunct2(); })
    .done(function () { theEnd(); });
```

And the following is labeled as one of the *don'ts*:

```
//not very neat!
asyncFunct().then(function () {
    asyncFunct1().then(function () {
            asyncFunct2().done(function () { theEnd(); });
    })
});
```

> We chain Windows Runtime (WinRT) methods that return promises, which is the same as chaining WinJS promises.

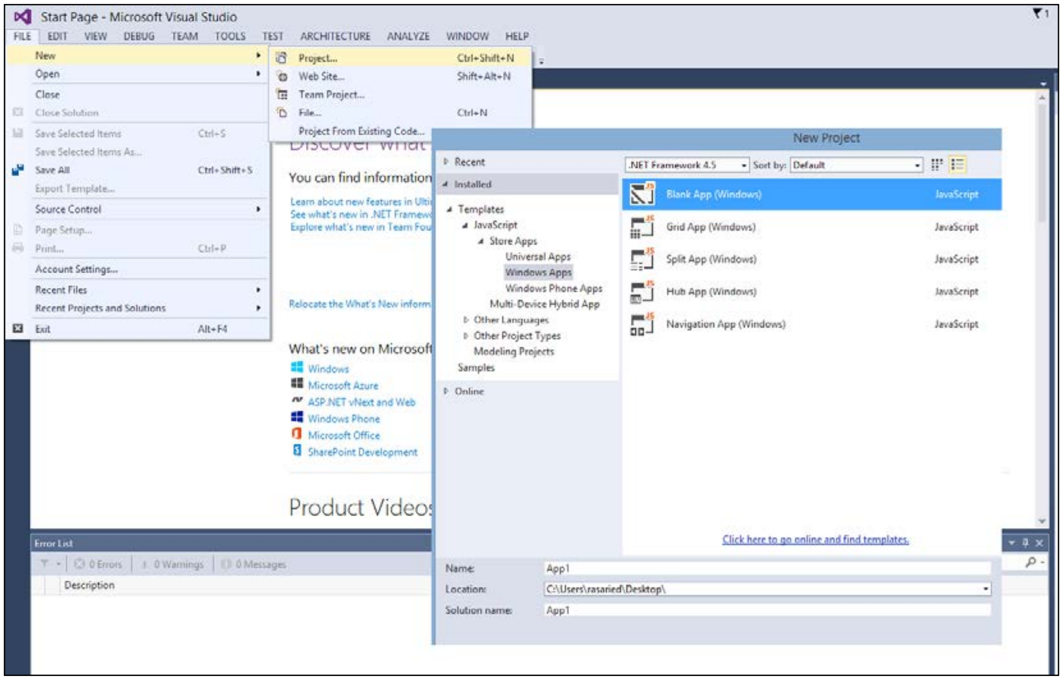
Bear in mind that the promises in WinJS are compliant to the promises defined in the CommonJS Promises/A proposal, and till the time of this writing, WinJS promises have not been tested against the new Promises/A+ specifications. This has no effect on Windows app development, as the apps are running in a store. In the browser, the main difference that can arise between WinJS promises and A+-complaint promises is that WinJS promises do not guarantee that the callback function of the promise will be asynchronous. For example, if we call `promise.then(a)` with `a` being the callback function, we won't be certain whether `a` will be called in an asynchronous or synchronous manner. While in Promises/A+ specifications, the callback function `a` will always be called asynchronously. It is a must. The authors of the specifications explain that not ensuring an asynchronous callback will make promises much harder to reason about and debug. Nevertheless, as I mentioned earlier in the chapter that WinJS itself is now open source and hosted on GitHub, the community members, and anyone interested for that matter, can download WinJS, build, and test it against the Promises/A+ Compliance Test Suite.

Up next, let us have a look at how to use these WinJS promises in Windows apps development.

Using WinJS promises

We leverage promises on the Web to make the UI more responsive and avoid blocking the UI thread by executing the work asynchronously. Likewise, we use WinJS promises to handle the work asynchronously and thus keep the UI thread of the Windows app available to respond to user input. We also allow the app layout and static items to load properly and promptly while fetching what needs to be fetched from servers and databases asynchronously in the background. For that purpose, the asynchronous APIs in WinJS and Windows Runtime are exposed in JavaScript as promises.

Let us have a look at a basic example of promises. In order to follow with and replicate the following example, we will need Visual Studio (the Express Version would do). We need to start by creating a basic Windows app of the type JavaScript. In order to do so, from the Visual Studio top menu, we need to go to **File | New | Project**, which will pop up a small window containing the project type. There, we need to go to **JavaScript | Store Apps | Windows Apps**, which will list for us the different JavaScript Windows App templates available. For this example, we can select **Blank App**, which is a project for a single-page Windows app that has no predefined controls or layout. Name the app as you please, and click on **Ok**. The following screenshot illustrates the steps taken:



Now, we have a blank Windows app that we can add some code to. For that, we need to navigate to the `default.html` page and modify it. Open that page, and insert an `input` element and a `div` element to display some result in the `body` element, as per the following syntax:

```
<body>
    <p>Content goes here</p>
    <br/>
    <div>
        <input id="urlInput" />
    </div>
 <br/><br/>
    <div id="resultDiv">The result will show here</div>
</body>
```

Next, we need to attach some code to the change handler for the `input` element so that we can do some work whenever the value of the `input` element changes. We can achieve this with the `addEventListener` method and request this as part of the `WinJS.Utilities.ready` function. Adding the event listener inside this function will allow the change handler that we are attaching to be called directly after the DOM has been loaded via the `DOMContentLoaded` event, which will in turn, fire after the page code has been parsed and before all the resources have been loaded.

Navigate to the `default.js` file, located inside the `js` folder. There, we need to add the following code to the end of the `app.onactivated` event handler:

```
WinJS.Utilities.ready(function () {
    Var inpt = document.getElementById("urlInput");
    inpt.addEventListener("change", onChangeHandler);
  }, false);
```

In the previous code, we are adding an anonymous function code to `WinJS.Utilities.ready`. In that anonymous function, we first get that `input` element from the DOM, assign it to a variable named `inpt`, and then call the method `addEventListener` on that `inpt` variable, which adds the function named `onChangeHandler` to the change event.

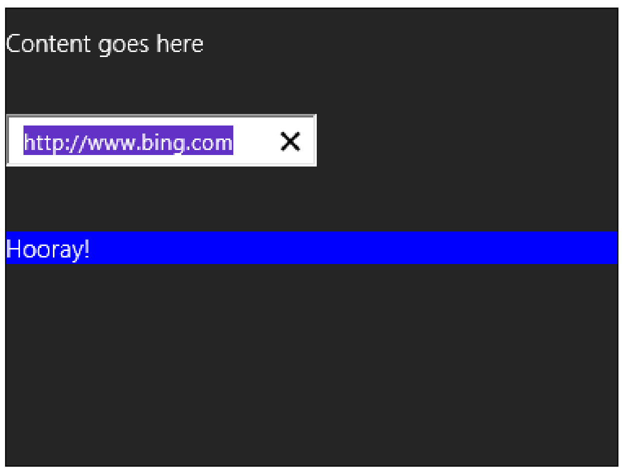
The last step would be to write the code for the `onChangeHandler` function. In that function, we will call the `WinJS.xhr` method, which basically wraps calls to `XMLHttpRequest` and exposes it as a promise. We can use this method for cross-domain requests and intranet requests. We will pass the URL that the user enters in the `input` element to the parameter of `xhr` and accordingly update the `resultDiv` element with the result. `Xhr` is an asynchronous function that returns a promise; hence, we can call the `then` or `done` methods of the promise object on this function to update the UI. For this example, we will call the `then` method that is invoked as soon as the `xhr` function has either successfully completed the `XmlHttpRequest` or has raised an error. `Then` can take up to three parameters for success, error, and progress, as we have seen in the definition earlier. However, for this basic example, we will just see how to add the `onCompleted` function. This success handler will apply some changes to the `resultDiv` element by setting its background color to blue and the inner text to `Hooray!`.

The syntax for the `onChangeHandler` function will look like the following code:

```
function onChangeHandler(e) {
            var input = e.target;
            var resDiv = document.getElementById("resultDiv");

            WinJS.xhr({ url: e.target.value }).then(function
              onCompleted(result) {
                if (result.status === 200) {
                    resDiv.style.backgroundColor = "blue";
                    resDiv.innerText = "Hooray!";
                }
            });
        }}
```

Let's analyze the previous code sample. We first retrieve the `input` element from the `e` argument, we get the `resultDiv` element to a variable `resDiv`, and then we call `WinJS.xhr` and pass it the value of the `input` element we got from the target. This value holds the URL that we enter in the textbox. Next, we call `then` on the `xhr` function, and pass to `then` the success handler `onCompleted` that contains the result as an argument. The result here represents the HTTP request. If the request holds the status 200, which is the status of success in HTTP requests, we will apply the changes on the `resultDiv`.

If we run the app now, we will have the following outcome after we enter a URL into the textbox:



How about reporting the progress while the result is retrieved? In order to do so, we will need to write the handler for progress in the `then` call on the `xhr` function. We will change the background color to green until the request is completed and the `onCompleted` handler is invoked, which will change the background color to blue. We will modify the code to include the progress handler as per the following code:

```
function onChangeHandler(e) {
    var input = e.target;
    var resDiv = document.getElementById("resultDiv");
    WinJS.xhr({ url: e.target.value }).then(
      function onCompleted(result) {
        if (result.status === 200) {
            resDiv.style.backgroundColor = "blue";
            resDiv.innerText = "Hooray!";
        }
    }, function myfunction() {
```

```
    //no error handling here; just passing an empty parameter
        }, function progress(result) { //handle progress here
            if (result.status != 200) {
                resDiv.style.backgroundColor = "green";
            }
        });
    }
```

In the previous code sample, we added an empty error handler and a progress handler as an anonymous function `function progress(result)`, which will check whether the request status is different from 200, which means it is not a success yet and sets the background color to green. If we run the app now and enter a URL into the textbox, we will notice that the background color of the `div` element is green for a second or so, and then it changes to blue and the text is updated to Hooray!.

WinJS promises can now also be used in the browser as the product team has enabled some WinJS features to run on the Web. You can see WinJS in live action using the new online editor via `http://try.buildwinjs.com/`. In any browser, we can view and edit the code, play around with WinJS, and check the results live.

Summary

WinJS provides a strong implementation of promises that we can use to wrap any kind of operation and leverage asynchronous programming for Windows apps using JavaScript in an efficient and effective manner.

In the next and last chapter, we will summarize what we have learned throughout the previous chapters on JavaScript promises and put into action more mature code samples than the ones we have seen so far.

6
Putting It All Together – Promises in Action

In *Chapter 5*, *Promises in WinJS*, we were introduced to the WinJS library and learned about the WinJS promise object in detail. We also had a quick glance at how to use WinJS promises in windows app development in a basic example. Finally, we are here in the last chapter, in which we will put into action the learning we have gathered throughout this book about promises. We will try to get a deeper understanding of how promises work by creating a simple implementation. After we create the implementation library, we will use it in a basic example that leverages that library for an asynchronous operation. In this chapter, we will cover the following topics:

- Summary of what we have covered and learned so far
- Creating a promise implementation in a simple JavaScript library

Implementing a promise library

Promises have grown to be very popular as expressed by the numerous standalone implementations of them. Moreover, Promises/A+ have more than 35 compliant implementations so far as we approach the launch of ECMAScript 6. One thing to note is that the growing adoption of Promise/A+ in JavaScript is reflected in other languages, with a number of implementations in ActionScript, Python, and Objective C. Although, in terms of semantics, these implementations might not necessarily match the ones we have in the JavaScript specifications due to different language capabilities, it ultimately cannot be verified for compliance by directly testing them against the JavaScript test suite of Promise/A+. Nevertheless, it's worth mentioning the implementations and showcasing the efforts taken.

Let us go through a code example for a basic implementation of promises; this will give us a better understanding of how promises work. A deep understanding of how things work improves our ability to take advantage of code and debug it more easily and quickly when it goes wrong. We will create a minimal JavaScript library that implements promises, and we shall start coding that library with the states of the promises. We learned, when exploring the Promise API in *Chapter 2*, *The Promise API and Its Compatibility*, that promises have three different states: pending, fulfilled, and rejected.

The specification of promises does not specify a value for these states, so let us declare them and assign the values to an enumerator, as the following code shows:

```
var promState = {

 pending: 1,

 fulfilled: 2,

 rejected: 3

};
```

This enumeration will allow us to call the state by name, for example, `promState.fulfilled`. Next, we will create an object that holds all the promise logic from transition between states to the `then` method and resolves the promise. Let us call this object `PromiseMe`.

First, we will need to define the change in the promise's states and its transition from one state to the other. The specification dictates some rules and considerations for the transitions between the states, which we have dived into detail in *Chapter 2*, *The Promise API and Its Compatibility*. These rules can be summarized as follows:

- A promise can only be in one state at a certain point of time
- When a promise transitions from pending to any other state, either fulfilled or rejected, it cannot go back
- When a promise fulfills, it must have a value (it can even be undefined) and, when it fails, it must have a reason (any value that specifies why the promise was rejected)

Inside the `PromiseMe` object, we will first define a function titled `changeMyState` that handles and manages the transition between states for this promise governed by the preceding rules, as the following code shows:

```
var PromiseMe = {
    //set default state
 myState: promState.pending,
```

```
 changeMyState: function(newState, newValue) {

  // check if we are changing to same state and report it
  if (this.myState == newState) {
   throw new Error("Sorry, But you can't do this to me! You are
transitioning to same state: " + newState);
  }

  // trying to get out of the fulfilled or rejected states
  if ( this.myState == promState.fulfilled ||
    this.myState == promState.rejected ) {
   throw new Error("You can't leave this state now: " +
     this.myState);
  }
  // if promise is rejected with a null reason
  if ( newState == promState.rejected &&
    newValue === null ) {
   throw new Error("If you get rejected there must be a reason. It
     can't be null!");
  }

  // if there was no value passed with fulfilled
  if (newState == promState.fulfilled &&
    arguments.length < 2 ) {
   throw new Error("I am sorry but you must have a non-null value to
proceed to fulfilled!");
  }

  //we passed all the conditions, we can now change the state
  this.myState = newState;
  this.value = newValue;
  return this.myState;
 }
};
```

The code inside the object will first set a property named `myState` to the pending value of the enumeration `promState` with `myState: promState.pending`. Afterwards, we set a property named `changeMyState` to an anonymous function that takes two arguments: `newState` and `value`. Within that function, we handle the transitions of states and check if it adheres to the rules. We have four checkpoints before we can proceed with the code:

1. First, we check whether we are transitioning to the same state and throw an error.
2. In the second check, we make sure that the promise is not trying to transition from rejected or fulfilled and throws an error accordingly.

3. The third check is for the value passed with rejected. If it is null, an error will be thrown and this makes sure that the promise gets rejected with a value other than one with null. We are writing this checkpoint because the promise, as per the specification, only accepts non-null values.
4. The final check would be for the fulfilled state and its value; we check with `arguments.length < 2` to determine if there was a value passed in the second argument; if not, we throw an error.

> I gave the error messages a meaningful wording to better understand what we are checking for in these conditions. After we pass all the condition statements, we close the `changeMyState` method by setting the `myState` property of the promise object to `newState`, passed in with the arguments. We also assign the value to the `newValue` argument, and we finish by returning `this.myState`, which, in turn, returns the state of the promise.

Implementing the then method

Up next in our implementation is the `then` method. This is the main pillar of promises, and it is what actually makes the promise useful. This method allows for and brings about the chaining of promises and handling errors. We will implement a basic `then` method that will first check the rules for the validity of the promise.

Let us define the `then` method as the following:

```
then: function (onFulfilled, onRejected) {
        // define an array named handlers
        this.handlers = this.handlers || [];
        // create a promise object to return
        var returnedPromise = Object.create(PromiseMe);

        this.handlers.push({
            fulfillPromise: onFulfilled,
            rejectPromise: onRejected,
            promise: returnedPromise
        });
        return returnedPromise;
    }
```

What the previous code does basically is define a `then` method for this promise. `Then` is defined as an anonymous function that takes two arguments: `onFulfilled` and `onRejected`. We define an array for this promise and initialize it to either the current array `this.handlers`, if it exists, or a new array, if it doesn't. We instantiate a new promise and store it in the `returnedPromise` variable. We store `onFulfilled`, `onRejected`, and `returnedPromise` in the array so that we can invoke these handlers later after we return the promise. This function closes with the returning of the promise.

> The rules of the `then` method, as per the Promise/A+ specification, state that the function arguments: `onFulfilled` and `onRejected`, must be called only after the promise is fulfilled or rejected accordingly. That is why, in the implementation, we stored those two functions in an array so that we can call them later.

You might have noticed that the handlers array contains two properties: `fulfillPromise` and `rejectPromise`. These are two functions that are set to the handlers that are passed in the arguments of the `then` method. Let us define these two functions so that we can use them later in the `resolve` method. These functions are helper methods and simply allow us to manually change the state of a promise. Moreover, these functions will call the `changeMyState` method to change the state of the promise which in turn returns a state.

```
fulfillPromise: function (value) {
//change state to fulfilled and return a promise with a value
        this.changeMyState(promState.fulfilled, value);
    },
rejectPromise: function (reason) {
//change state to rejected and return a promise rejected with a reason
        this.changeMyState(promState.rejected, reason);
    }
```

Defining a resolve method

Moving forward, we need to address resolving that promise. We need to define a resolve method that will deal with the promise and will either fulfill it or reject it, depending on the state of the promise. You can think of the `resolve` method as an internal method that the promise calls and is aimed at executing the `then` calls only when the promise is fulfilled; in literary terms, it resolves a fulfilled promise. Actually, the function that you will need to call in order to fulfill a promise or reject it would be `changeMyState` in our case here. Let's start by creating a basic logic for the `resolve` method as per the following code:

```
    resolve: function () {
        // check for pending and exist
```

```
        if (this.myState == promState.pending) {
            return false;
        }
```

The previous code assigns the property `resolve` to a function. Inside that function, we first check for the state of this promise. If it is pending, we return `false`. Next in code, we will loop through the array that contains the handlers we defined in the `then` method:

```
// loop through each then as long as handlers array contains items
while(this.handlers && this.handlers.length) {

//return and remove the first item in array
var handler = this.handlers.shift();
```

Inside that loop, we apply the `shift()` function on the array. The `shift()` function allows us retrieve the first item from that array and remove it directly after. Thus, the `handler` variable will contain the first item in the `handlers` array, and in return, the `handlers` array will contain all the items minus that first one which is now stored in the `var` handler.

Next in the `resolve` function, we will define a variable named `doResolve`, which is set to the value of either the `fulfillPromise` function or the `rejectPromise` handler depending on the state as per the following code:

```
//set the function depending on the current state
var doResolve = (this.myState == promState.fulfilled ?
  handler.fulfillPromise : handler.rejectPromise);
```

The preceding syntax uses the ternary operator. It is called the ternary operator, because unlike all other operators that take two values, this operator actually requires a third value to be placed in the middle of the operator. It is like a single-statement shorthand alternative for an `if` statement, where both the `if` and `else` clauses will assign different values to the same variable as per the following example:

```
if (condition == true)
  result = "pick me";
else
  result = "No! pick me instead";
```

The ternary operator will transform the `if` statement to the following single-line condition statement:

```
result = (condition == true) ? "pick me" : "No!
pick me instead";
```

We need to add some sanity check for the `doResolve` function. If it is not of the type function or the function doesn't exist, then we invoke the `changeMyState` method on the promise so that we pass along the state and the value:

```
//if doResolve is not a function
if (typeof doResolve != 'function') {
handler.promise.changeMyState(this.myState, this.value);

}
```

Implementing the doResolve function

The other route for this code would be that the `doResolve` function exists, and we need to return the promise with a value or reject it with an error. So, we follow up the `if` condition with an `else` statement to implement this case as per the following code:

```
else {
//fulfill the promise with value or reject with error
try {
```

As per the code logic so far, we would now have `doResolve` containing the `handler.fulfillPromise` or `handler.rejectPromise` functions. These two functions can manually change the state of the promise and take one argument, which is the current value or current reason. Both values are contained in the `this.value` variable. Hence, we will pass the current value to `doResolve` and assign the result to a variable named `promiseValue` as per the following line of code:

```
var promiseValue = doResolve(this.value);
```

Next, we need to manage the promise returned with `promiseValue`. First, we check if the promise exists and has a valid `then` function as per the following code:

```
// deal with promise returned
        if (promiseValue && typeof promiseValue.then ==
          'function') {
```

Assuming that we pass this condition, inside it we can call the `then` method on `promiseValue` since it now contains a promise that is the result of the `doResolve` function. We will pass two arguments to its `then` method: a function parameter for `onFullfilled`, and another one for `onRejected`, as the following code shows:

```
//invoke then on the promise
promiseValue.then(function (val) {
    handler.promise.changeMyState(
      promState.fulfilled, val);
}, function (error) {
```

```
        handler.promise.changeMyState(
          promState.rejected, error);
    });
    }
```

On the other hand, if the value returned by `promiseValue` was not a promise, we will not need to invoke the `then` method. Instead, we simply change the state to fulfilled and pass it the value. We will deal with this in an `else` statement as the following code shows:

```
// if the value returned is not a promise
else {
handler.promise.changeMyState(promState.fulfilled, promiseValue);
}
```

Finally, since we are in a `try` statement, we will need to provide a `catch` statement accordingly, where we will deal with any error that is thrown if the operation fails. In that `catch` statement, we will change the state of the promise to rejected and pass it the error that arises. We will also close all the trailing braces:

```
// deal with error thrown
} catch (error) {
handler.promise.changeMyState(promState.rejected, error);
   }
}
}
}
```

Resolving the promise includes some tedious checks, but these are necessary to guarantee the fidelity of the promise implementation against the specification. As you have seen, we added the logic as we proceeded, which starts off with a simple check to see if we are running the `onFulfilled` or `onRejected` functions based on the promise state. Following this, we change the state of their corresponding promise based on the return values.

> Bear in mind that the implementation needs to adhere to the considerations and rules that exist in the specification. At any point of time, you can double-check the code with the details of the Promise API explained in *Chapter 2*, *The Promise API and Its Compatibility*, of this book.

We are almost finished, and what remain are two scenarios we haven't addressed yet. The first scenario is that the `onFulfilled` and `onRejected` handlers must not be called in the same round of the event loop `while` (`this.handlers && this.handlers.length`). We do this check because `while` is looping through each `then` call. In a `then` call, the promise is either fulfilled or rejected. Hence, in our case here, we have the `onFulfilled` and `onRejected` handlers. To fix this issue, we will need to add the `then` methods to the array only after the event loop. We can achieve this with the use of the `setTimeout` function, which, in turn, ensures we are always running asynchronously. Let us add the `setTimeout` function in the `then` method and wrap the function that stores the promise handlers in the array, as the following code shows:

```
var that = this;
setTimeout(function () {
    that.handlers.push({
         fulfillPromise: onFulfilled,
         rejectPromise: onRejected,
         promise: returnedPromise
      });
    that.resolve();
 }, 2);
```

Wrapping the code

The final step in this implementation would be to indicate where and when we actually resolve the promise. There are two conditions that we need to check. The first one is when we are adding the `then` method, because the state of the promise might already be set there. And the second case is when the state of the promise changes in the `changeMyState` function. Hence, we will need to add a `this.resolve()` call to the end of the `changeMyState` function. All we need to do now before we finalize the implementation is wrap all the code in an anonymous function named `PromiseMe`. It will use `Object.create` to give us a promise. And with that, the final code of this promise implementation will look as follows:

```
var PromiseMe = function () {
    var promState = {
        pending: 1,
        fulfilled: 2,
        rejected: 3
    };
    //check the enumeration of promise states

    var PromiseMe = {
        //set default state
```

```
        myState: promState.pending,
        changeMyState: function (newState, newValue) {

            // check 1: if we are changing to same state and
              report it
            if (this.myState == newState) {
                throw new Error("Sorry, But you can't do this to
                  me! You are transitioning to same state: " +
                  newState);
            }

            // check2: trying to get out of the fulfilled or
              rejected states
            if (this.myState == promState.fulfilled ||
              this.myState == promState.rejected) {
                throw new Error("You can't leave this state now: "
                  + this.myState);
            }
            // check 3: if promise is rejected with a null reason
            if (newState == promState.rejected && newValue ===
              null) {
                throw new Error("If you get rejected there must be
                  a reason. It can't be null!");
            }
            //check: 4 if there was no value passed with fulfilled
            if (newState == promState.fulfilled &&
              arguments.length < 2) {
                throw new Error("I am sorry but you must have a
                  non-null value to proceed to fulfilled!");
            }

            // we passed all the conditions, we can now change the
              state
            this.myState = newState;
            this.value = newValue;
            this.resolve();
            return this.myState;
        },
        fulfillPromise: function (value) {
            this.changeMyState(promState.fulfilled, value);
        },
        rejectPromise: function (reason) {
            this.changeMyState(promState.rejected, reason);
        },
        then: function (onFulfilled, onRejected) {
```

```
        // define an array named handlers
        this.handlers = this.handlers || [];
        // create a promise object
        var returnedPromise = Object.create(PromiseMe);
        var that = this;
        setTimeout(function () {
            that.handlers.push({
                fulfillPromise: onFulfilled,
                rejectPromise: onRejected,
                promise: returnedPromise
            });
            that.resolve();
        }, 2);

        return returnedPromise;
    },
    resolve: function () {
        // check for pending and exist
        if (this.myState == promState.pending) {
            return false;
        }
        // loop through each then as long as handlers array
          contains items
        while (this.handlers && this.handlers.length) {
            //return and remove the first item in array
            var handler = this.handlers.shift();

            //set the function depending on the current state
            var doResolve = (this.myState ==
              promState.fulfilled ? handler.fulfillPromise :
              handler.rejectPromise);
            //if doResolve is not a function
            if (typeof doResolve != 'function') {
                handler.promise.changeMyState(this.myState,
                  this.value);

            } else {
                // fulfill the promise with value or reject
                  with error
                try {
                    var promiseValue = doResolve(this.value);

                    // deal with promise returned
                    if (promiseValue && typeof
                      promiseValue.then == 'function') {
```

```
                            promiseValue.then(function (val) {
                                handler.promise.changeMyState(
                                  promState.fulfilled, val);
                            }, function (error) {
                                handler.promise.changeMyState(
                                  promState.rejected, error);
                            });
                            //if the value returned is not a
                              promise
                        } else {
                            handler.promise.changeMyState(
                              promState.fulfilled, promiseValue);
                        }
                        // deal with error thrown
                    } catch (error) {
                        handler.promise.changeMyState(
                          promState.rejected, error);
                    }
                }
            }
        }
    };
    return Object.create(PromiseMe);
};
```

The previous code represents a basic promises implementation in a small JavaScript library. It implements a promise object with its `t0068en` method, taking into consideration the specification requirements of how to fulfill and reject a promise while addressing the necessary checks to avoid anomalies in the implementation. We can take this library and start calling its `PromiseMe` object and its corresponding functions, `then`, `fulfillPromise`, and `rejectPromise`, to achieve some asynchronous operations.

> This implementation is a basic one; we can extend it to include many features and helper methods that can be built on top of the Promises API. Furthermore, we can build this implementation and test it against the Promises/A+ Compliance Test Suite, which can be found via this link: `https://github.com/promises-aplus/promises-tests`.

On the link provided in the preceding information box, we can find the steps needed to complete the tests, which will need to run in a Node.js environment, and we need to make sure that Node.js is installed already.

Putting the promise into action

We can take this basic implementation of promise that we just authored and use it in our code to handle our asynchronous operations. Let us have a look at an example of how to use this `PromiseMe` library. The following code can be added after this code for the `PromiseMe` object:

```
var multiplyMeAsync = function (val) {
    var promise = new PromiseMe();
    promise.fulfillPromise(val * 2);

    return promise;
};

multiplyMeAsync(2)
    .then(function (value) {
    alert(value);
});
```

What we are doing in the previous code is simply creating a function named `multiplyMeAsync`, which in turn instantiates `PromiseMe` to a variable named `promise` and then calls the `fulfillPromise` method on the `promise` variable we created, which is an instance of the `PromiseMe` object. What the `fulfillPromise` method does is simply multiply the `val` argument by the number 2. Following that, we call `multiplyAsync` and pass it the number 2 as a value for its parameter; since it returns a promise, we can call the `then` method on it. The `then` method has a single handler, which handles the success and simply pops up an alert with the value that should now be 4.

Run the script in an HTML page, and we should have an alert displaying the number 4.

> You can find the complete code and test it out at jsFiddle via `http://jsfiddle.net/RamiSarieddine/g8oj4guo/`. Make sure the browser supports promises.

Let us try to add some error handling to this code. First, for the sake of simplicity and readability, I will create a function named `alertResult` to replace `alert(value);`.

Hence, we will have a function as the following:

```
var alertResult = function (value) {
    alert(value);
};
```

We will add another function called `onError`, which basically displays an alert with the error message passed to it. The function will have the following syntax:

```
var onError = function(errorMsg) {
 alert(errorMsg);
};
```

Now, let us add a function that will include error handling by detecting an anomaly and rejecting the promise. The following code shows this:

```
var divideAsync = function (val) {
    var promise2 = new PromiseMe();
    if (val == 0) {
        promise2.rejectPromise("cannot divide by zero");
    }
    else{
        promise2.fulfillPromise(1 / val);
    }
    return promise2;
};
```

What the previous function does is simply check for the value; if it is zero, the function rejects the promise; otherwise, it fulfills the promise by dividing the number 1 by `val`. To test this, we will pass the value 0 to `multiplyAsync` , invoke `divideAsync` in its `then` call, and finish by calling an error function in the `then` method of `divideAsync`. The code will look as follows:

```
multiplyMeAsync(0)
    .then(divideAsync)
    .then(undefined, onError);
```

The end result will be an error message that displays the text **cannot divide by zero**. That is because zero got passed to `divideAsync`, which in turn rejected the promise, and an error message was passed to the `onError` handler.

> You can find the updated code with this error handler scenario on the following jsFiddle URL:
>
> `http://jsfiddle.net/RamiSarieddine/g8oj4guo/15/`

To wrap it up, promises do offer a very good solution to address the complexities of asynchronous operations. The abstraction that promises provide allows us to do several things more easily, especially common asynchronous patterns using callbacks, with the support of the following major properties:

- A single promise can be attached to and cater to more than one callback
- Values and errors get passed along in the promise

Summary

Throughout the previous five chapters of this book, we have learned a great deal about promises. We started with asynchronous programming in JavaScript and took stock of where promises stand amidst that world, during which we discussed in detail why you should care about promises as of now. Next, we dived deep into the Promises API and a thorough description of its `then` method. Following this, we learned about the browsers that currently support promises and the libraries that implement promise-like features. Next, we covered the chaining of promises and gave a detailed explanation of how we can achieve this as well as queue the asynchronous operations using promise chains. Our third stop was error handling, one of the most important aspects of the concept of promises. We stepped back and took a look at exceptions in JavaScript and how they are addressed in promises. We also learned about the `catch` method as part of handling errors with promises.

Promises now have native support in JavaScript, and this is a focal time in the world of web and client-side development to start leveraging this technology. As we move forward, more browsers will start adopting promises as a standard and have it become native in the browser.

If people around you have been making noise about JavaScript promises, well now you know why. This book represents a single point of reference—a comprehensive one—just when you want to learn about promises and saving you from all these bits and pieces scattered out there on the Web. You can take this learning and implement it right away. You can always come back to this book for reference and details on the API. From here, you can start implementing your own promise libraries and make use of other libraries available as well as delve into other implementations, such as Node.js. You can also start using promises for the asynchronous requests to databases, the Web, or file servers.

I hope you enjoyed reading this book and that it left you equipped with the right knowledge, tools, and tips to put that learning into practice and make some killer applications that leverage the power of JavaScript promises.

Index

L

M

O

P

Q

R

S

T

V

W

X

Thank you for buying

JavaScript Promises Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.