

Knockout.js

BUILDING DYNAMIC
CLIENT-SIDE WEB APPLICATIONS



Jamie Munro

Knockout.js

Use Knockout.js to design and build dynamic client-side web applications that are extremely responsive and easy to maintain. This example-driven book shows you how to use this lightweight JavaScript framework and its Model-View-ViewModel (MVVM) pattern. You'll learn how to build your own data bindings, extend the framework with reusable functions, and work with a server to enhance your client-side application with persistence. In the final chapter, you'll build a shopping cart to see how everything fits together.

If you're a web developer with experience in JavaScript, HTML, and CSS, you're ready for Knockout.

- Learn how to create a ViewModel
- Bind HTML data and attributes, and CSS classes and styles
- Understand data binding in Knockout's context hierarchy
- Use properties that change dynamically through user interaction
- Work with forms by using several different bindings
- Bind multiple ViewModels on a single page
- Extend or attach custom functions to observables
- Perform server-side interactions with jQuery
- Map a JavaScript object or apply JSON data to a new object

Jamie Munro has been developing websites and web applications for over 15 years. For the past six years, he's been mentoring younger developers to enhance their web development skills. Jamie's website (<http://www.webistrate.com>) provides examples to help web developers expand their experience.

“This book is a great resource for any developer looking to make their web application user experience pop. It presents the features and benefits of Knockout.js with extensive examples and in-depth explanations. You also benefit from the author's recommendations and advice coming from lessons learned through his experience building web applications.”

—Steven Kennedy

Senior Software Developer at Fusebill

PROGRAMMING/JAVASCRIPT

US \$19.99

CAN \$20.99

ISBN: 978-1-491-91431-1

5 1 9 9 9

9



Twitter: @oreillymedia
facebook.com/oreilly

Knockout.js

Building Dynamic Client-Side Web Applications

Jamie Munro

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Knockout.js

by Jamie Munro

Copyright © 2015 Jamie Munro. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Meg Foley**Production Editor:** Kara Ebrahim**Copyeditor:** Gillian McGarvey**Proofreader:** Marta Justak**Indexer:** Judy McConville**Interior Designer:** David Futato**Cover Designer:** Ellie Volckhausen**Illustrator:** Rebecca Demarest

December 2014: First Edition

Revision History for the First Edition

2014-12-08: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491914311> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Knockout.js*, the cover image of a black tree kangaroo, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-91431-1

[LSI]

This book is dedicated to my family. I hope that you learn—like I have—that when you work hard and put your mind to something, you can accomplish anything!

Table of Contents

Preface.....	vii
1. Getting Started with KnockoutJS.....	1
Data Binding Syntax	1
Data Binding Example	2
What Is MVVM	3
Creating a ViewModel	3
Object-Oriented ViewModels	4
ViewModels with Parameters	5
2. The Ins and Outs of Data Binding.....	7
Binding HTML Data	7
Binding HTML Attributes, CSS Classes, and CSS Styles	8
Condition-Based Data Binding	9
3. Understanding the Data Binding Context.....	11
foreach Binding	12
foreach Callbacks (Events)	15
with Binding	17
4. Dynamically Changing Properties.....	21
Defining an Observable	21
pureComputed Observables	23
Showing and Hiding Elements	24
Adding and Removing Elements	26
When to Use Observables	28

5. Working with Forms.....	31
Form Data Bindings	31
Event Data Bindings	33
Listening for Changes	34
Unobtrusive Form Validation	36
6. Multiple ViewModels, Custom Data Bindings, and Templates.....	39
Binding Multiple ViewModels	39
Binding to a WYSIWYG Editor	40
Binding to a Knockout Template	43
7. Enhancing Observables.....	47
Extending Observables	47
Adding Custom Functions to Observables	49
Rate-Limiting Observables	51
8. Server-Side Interaction.....	55
Sending Data	55
Receiving Data	58
9. The Mapping Plugin.....	63
Map from an Object	63
Map from JSON (or a Server)	65
Observing and Ignoring Specific Properties	68
10. A Practical Example.....	71
Building a Shopping Cart	71
11. Next Steps.....	79
The Simplicity of Knockout	79
The User's Context	80
Page Reloads for Content Updates	80
Conclusion	81
Index.....	83

Preface

Dynamic Client-Side Applications

User Interfaces (UI) are becoming increasingly popular, much like responsive web design. Extremely popular websites like Gmail, Facebook, and Twitter have adopted extremely responsive client-side applications.

Rarely do you see a full-page refresh; instead, the content is loaded inline. Most importantly, your *context* is maintained and you are not forced to jump around between multiple pages.

A perfect example of this is Gmail. Ironically, it came from writing this book. I was reviewing an email from my editor and technical reviewer. I had the email opened and wanted to respond to it. Because Gmail placed my reply inline, I could easily review the email and write my response at the same time—no need to jump around or have two windows open.

KnockoutJS makes implementing web applications like this quick, efficient, and most importantly, easy to maintain!

Installing KnockoutJS

KnockoutJS made its debut back in July of 2010. It was released as an open source project by author Steve Sanderson. It is now being maintained by the open source community. Version 3.0 was released in late 2013.

It continues to evolve daily with new features, enhancements, and bug fixes. All of the examples in this book use version 3.2 (the stable release).

Setting up KnockoutJS is very simple. Begin by visiting [the KnockoutJS downloads page](#).

You can download the minified version of Knockout by right-clicking (Ctrl+Click for Mac) and selecting the “Save link as...” option.

I suggest creating a new folder for all of the code examples in this book. Inside that folder, make a subfolder called *js* and place the *knockout-3.2.0.js* file inside.



Has the KnockoutJS Version Changed?

It is quite normal for popular open source frameworks to evolve quite quickly. If the current version of Knockout is not 3.2 when you download it, the examples in this book should continue to work. However, you will need to update the following:

```
<script type='text/javascript'
src='js/knockout-3.2.0.js'></script>
```

to:

```
<script type='text/javascript'
src='js/knockout-<current-version>.js'></script>
```

You will need to replace `<current-version>` with the correct value.

If you want to use version 3.2 like the examples in this book, you can visit the [KnockoutJS project on Github](#) and grab your version of choice.

Once the Knockout framework is downloaded, create an HTML page that loads the framework, like in [Example P-1](#).

Example P-1. Installing KnockoutJS

```
<!DOCTYPE html>
<html>
<head>
  <title>Installing KnockoutJS</title>
</head>
<body>
  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
</body>
</html>
```

In the example above, I included KnockoutJS just before the end body tag. This is done in order to allow the web browser to render the HTML without waiting for the JavaScript file to be downloaded and executed.

Some developers prefer to place the JavaScript inclusion inside the head tag. This is completely valid, especially in scenarios where you do not want the user to interact with your website prior to all JavaScript files being loaded and processed by the browser.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/oreillymedia/knockout_js.


This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples

from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Knockout.js* by Jamie Munro (O'Reilly). Copyright 2015 Jamie Munro, 978-1-491-91431-1.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

 **Safari**® *Safari Books Online* is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of **plans and pricing** for **enterprise, government, education**, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds **more**. For more information about Safari Books Online, please visit us **online**.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://bit.ly/knockoutjs-book>.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

Where do I begin? I thought that writing my fourth book would be a lot easier than it was. My last book came together so smoothly, and I thought for sure this would be the same. It definitely was not! At every stage of this book, I was forced to stop and think “Why am I doing this a particular way?” There are so many things I do with Knockout and JavaScript a certain way without thinking, that I was forced to think about the reasons I do it this way. It was such a huge personal growth experience and I feel like a much better developer and hopefully these lessons will be passed on to each and every one of you.

I must first thank my wife, Shannon. She helped immensely by being such a good mom to our three young children—Lily, Owen, and Kayla—while I was writing this book. Without her love and support, I would get no sleep at all!

Secondly, to Steve Kennedy (you may see his name pop up every now and then in examples). We spent many a lunch hour going over minute details of almost every aspect of the book. He made sure that every example was technically accurate and ensured the “why” was thoroughly covered. I hope the “why” things are done will be invaluable throughout this book.

Third, to Mike Wilson (best to not try the email address in Example 9-4!). Mike really helped shape the early chapters to ensure we got right to the examples before presenting the theory. As a reader of many books, this always makes me feel better to see the results of the “Hello World” example immediately.

I can’t forget the team at O’Reilly. Thanks to Meg, Kara, and Gillian for making this a wonderful experience.

And finally, to you, the reader: I hope you enjoy reading this book and learn as much as I did from writing it!

CHAPTER 1

Getting Started with KnockoutJS

KnockoutJS is an open source JavaScript library. It was built to allow you to create dynamic and rich web applications. It is built with the **Model-View-ViewModel (MVVM) pattern**. Knockout makes it really simple to implement a complex user interface that responds to user interactions.

I like Knockout because it is one of the most lightweight JavaScript libraries available today. It also doesn't try to be an all-in-one framework. It serves a single purpose: data binding your ViewModel to your user interface.

Implementing Knockout involves three distinct things: a view that contains HTML and CSS elements that get data-bound to it, a ViewModel that contains the data to bind to the view, and telling Knockout to perform the data binding to the view with the ViewModel.

The examples in this chapter demonstrate how to create an HTML page with a basic ViewModel. After we review the basic data binding syntax, we will explore the various types of ViewModels that work with the view.

Data Binding Syntax

Data binding is accomplished by adding an HTML attribute called `data-bind` to any HTML element that you want Knockout to replace with information from your ViewModel.

Sometimes an HTML tag does not work, so Knockout also allows you to specify data bindings with HTML comments, as shown in **Example 1-1**.

Example 1-1. Knockout bindings using HTML comments

```
<!-- ko -->
<!-- /ko -->
```

Binding through HTML comments is extremely convenient when you want to wrap your binding around a block of HTML code, or you do not want to create an extra HTML object for the sole purpose of housing a data binding.

Data Binding Example

One of the most common things that Knockout is used for is to dynamically display text or HTML based on data contained within a ViewModel.

Example 1-2 shows creating a header that will display the `name` property within the ViewModel.

Example 1-2. Data binding text

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>
  <h1>Hello <span data-bind="text: name"></span></h1>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    var viewModel = function() {
      this.name = 'Steve Kennedy';
    };

    ko.applyBindings(viewModel);
  </script>
</body>
</html>
```

When this example is executed in a browser, it outputs `Hello Steve Kennedy` inside an `h1` tag.

A `span` tag is data-bound to the `name` property of the ViewModel. This is done by placing `text: name` inside the `data-bind` HTML attribute.

As mentioned in the introduction to this chapter, implementing Knockout requires three distinct things. The first is the view, which in this example is the HTML that contains the `h1` and `span` tags that identify the data binding.

The second is the `ViewModel`, which in this example is the JavaScript variable/function called `viewModel` that contains a single variable name.

The third is telling Knockout to perform the data binding of the view and the `ViewModel`. This is accomplished by calling the `ko.applyBindings` function with a `ViewModel`.

When this function is executed, Knockout processes both the view and the `ViewModel`. All data bindings in the view are executed and dynamically replaced with the data from the `ViewModel`.

Knockout does not limit you to a single `ViewModel` per view. Large projects or classes that are well designed for reusability are common reasons to bind multiple `ViewModels` to a single view. This is discussed in more detail in [Chapter 6](#).

What Is MVVM

The *Model-View-ViewModel* (MVVM) design pattern is based largely on the *Model-View-Controller* (MVC) pattern. In fact, the MV is shared between them. It's the `ViewModel` that really separates the two.

MVVM was designed to implement data binding between your `ViewModel` and your View. This is what KnockoutJS does for us and does very well. It is accomplished using some simple-to-implement HTML attributes and a JavaScript `ViewModel` as shown in [Example 1-2](#).

The most important thing to remember when you are building `ViewModels` is that they should be organized to make it easy to represent how your View uses the data. We explore several common scenarios in the upcoming examples.

Creating a ViewModel

A `ViewModel` can be any type of JavaScript variable. In [Example 1-3](#), let's start with a simple JavaScript structure that contains a single property called `name`.

Example 1-3. Basic ViewModel

```
var myFirstViewModel = {
  name: 'Steve Kennedy'
};
```

The previous example is a perfect scenario of a `ViewModel` that might not correlate 100% to a data model. Frequently, a data model separates the first and last name into two separate fields to make editing the data easier. However, in a view, it makes more sense to concatenate them into a single field for display.

The previous ViewModel is quite basic. ViewModels are not limited to such a simple structure, as the next examples demonstrate.

Object-Oriented ViewModels

Typically, when I create ViewModels, I create simple or complex JavaScript classes that allow me to leverage an object-oriented style of programming (functions, properties, abstraction, etc.).



Object-Oriented Programming with JavaScript

JavaScript is a fully object-oriented programming (OOP) language based on prototyping. It doesn't contain class statements like C++, C#, or PHP; however, JavaScript functions can be used to simulate the same behavior.

It also offers full support of OOP language features such as namespaces, objects, properties, inheritance, abstraction, etc.

If you are new to JavaScript or object-oriented programming, the Mozilla Developer Network (MDN) offers a [great introductory article](#).

In [Example 1-4](#), I create a ViewModel that is built with some basic object-orientation that will provide more functionality for our view.

Example 1-4. Object-Oriented ViewModel

```
var mySecondViewModel = function() {
    var self = this;

    self.name = 'Steve Kennedy';

    self.getName = function() {
        return self.name;
    };
};
```

In this example, my ViewModel is a JavaScript function that I've made to act like a class in OOP. It contains the same `name` variable that I created in the previous example, but this time it is encapsulated in my class and acts like a property of it.

I also added a new function called `getName` that allows me to access my `name` property without calling my class property directly from elsewhere in my code.



Self = This?

You may be wondering why the first line of my class is `var self = this;`. By creating a variable called `self` and assigning it the variable `this`, it provides me a property in my class that I can use inside methods of my class and easily reference other methods or properties in my class.

Even though the resulting code between [Example 1-4](#) and [Example 1-3](#) look quite different, in fact, they are quite similar. The `name` property can be accessed in the exact same way from either example, as shown in [Example 1-5](#).

Example 1-5. Accessing the name property

```
alert(myFirstViewModel.name);

alert(mySecondViewModel.name); // access the property name
alert(mySecondViewModel.getName()); // access the function getName
```

All three of the previous statements will alert `name`.

ViewModels with Parameters

In [Examples 1-3](#) and [1-4](#), the `name` property was hardcoded to a value. In many scenarios, data like this would be populated from a different source: a database, the results of an AJAX call, etc.

In [Example 1-6](#), the `name` will be populated via an input to the `ViewModel`.

Example 1-6. A ViewModel with a parameter

```
function ViewModel(name) {
    var self = this;

    self.name = name;

    self.getName = function() {
        return self.name;
    };
};

var myThirdViewModel = new ViewModel('Steve Kennedy');
```

In [Examples 1-3](#) and [1-4](#), the `ViewModel` class was assigned directly to the variable. This example is slightly different in that a function (acting like a class) called `ViewModel` is created that accepts the `name` property. This class is then instantiated by passing the `name` into the constructor.

In all future examples, I will continue to use JavaScript classes as they provide a lot more flexibility. However, if a simple JavaScript variable or structure will suffice for your view, do not feel the need to wrap your ViewModel in a class.

CHAPTER 2

The Ins and Outs of Data Binding

Binding HTML Data

In **Example 1-1**, text was data-bound to a `span` tag. Like text, it's quite common to data-bind content that contains HTML data.

In **Example 2-1**, the `getName` function is expanded to wrap the output of the name data binding with some HTML code.



HTML Versus Text Binding

HTML bindings work similarly to text with the exception that they render any HTML tags provided. If you were to use the text binding with HTML content, the HTML would be escaped, and the content would be displayed as text.

In **Example 2-1**, I update two things. First, the `data-bind` attribute is moved from the `span` tag to the `h1` tag. Second, it is bound to the `getName` function, which has been updated to contain some HTML code.

Example 2-1. `h1` tag with HTML binding

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <h1 data-bind="html: getName()"></h1>
```

```

<script type='text/javascript' src='js/knockout-3.2.0.js'></script>
<script>
    function ViewModel() {

        var self = this;

        self.name = 'Steve Kennedy';

        self.getName = function() {
            return 'Hello <em>' + self.name + '</em>!';
        };

        var viewModel = new ViewModel();
        ko.applyBindings(viewModel);
    }
</script>
</body>
</html>

```

Running [Example 2-1](#) would output a similar message to [Example 1-1](#), with the exception that the name would be italicized because of the HTML tag around the output of the name property.

Binding HTML Attributes, CSS Classes, and CSS Styles

Knockout can bind almost any HTML attribute, CSS class, or CSS style. These are done quite similarly to the text and html bindings with the exception that you can specify more than one property at a time. This is accomplished by wrapping the properties in curly brackets like in [Examples 2-2](#) and [2-3](#).

Example 2-2. Example CSS and Style bindings

```

<p data-bind="
    style: { marginBottom: 0, paddingBottom: '1em' },
    css: 'myClass'">
    This text has custom styles and a CSS class.</p>

```

The following example adds two custom styles to the p tag. Notice how the style properties `margin-bottom` and `padding-bottom` are CamelCase instead of hyphenated.

By placing a comma after the closing curly brace, I added a second data binding to the p tag. This time I set a CSS class of `myClass`.

Adding HTML attributes is done in a similar fashion. In [Example 2-3](#), an `id` attribute is added.

Example 2-3. Data-binding HTML attributes

```
<p data-bind="attr: { id: 'myCustomId' }">
  This p tag has an id data bound to it.</p>
```

Just like in [Example 2-2](#), if I wanted to bind additional HTML attributes, I could place a comma before the closing curly brace and add additional properties.

Condition-Based Data Binding

When you are defining a data bind, you might want to specify a condition that when your data contains a specific value, it outputs a different class name. This can be accomplished by placing conditional statements inside your data binding.

In [Example 2-4](#), a button is created. The text data binding contains a conditional statement that when the `id` is 0, the button text will say “Create.” However, when the `id` is greater than 0, it will say “Update,” allowing the use of a single button tag that dynamically changes.

Example 2-4. Conditional data binding

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <button type="submit" data-bind="text: (id == 0) ? 'Create' : 'Update'"></button>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    var viewModel = function() {

      var self = this;

      self.id = 0;

    };

    ko.applyBindings(viewModel);
  </script>
</body>
</html>
```

For this example to work, the ViewModel has been updated to include a new property called `id`. By setting it to 0, when this example is displayed in a web browser, the button text will say “Create.” By updating the `id` to be any other positive number, the text will change to say “Update.”



Condition Statements

When you are defining a conditional statement in your data binding, brackets are not required; however, I find it to be much more readable. It helps identify where the conditional statement is.

Just like when using data bindings, conditional bindings can be accomplished by using a function instead of placing the condition inside the data binding, as shown in [Example 2-5](#).

Example 2-5. Condition data binding with a function

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>
  <button type="submit" data-bind="text: (isNew()) ? 'Create' : 'Update'">
  </button>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    var viewModel = function() {

      var self = this;

      self.id = 1;

      self.isNew = function() {
        return id == 0;
      };

      ko.applyBindings(viewModel);
    }
  </script>
</body>
</html>
```

The ViewModel was updated in the example to add a new function called `isNew`. This contains the same conditional statement that was used in [Example 2-4](#). The data binding was also updated to use the function inside of the condition statement.

The `id` was updated from 0 to 1, so running this example outputs “Update” for the button text.

CHAPTER 3

Understanding the Data Binding Context

Knockout maintains a parent/child hierarchy of contexts. When you are accessing properties to data bind, everything is relative to the context you are in. The root context is the ViewModel that was supplied to the `ko.applyBindings` function.

Everything is relative to the current context, which is why the text bindings used the `name` property without the prefix of the ViewModel.

In the following examples, some of the new data bindings create a child context under the root context.

Knockout offers several useful variables that allow you to navigate between the context you are in to a parent or even the root context:

`$root`

This accesses the root context (the ViewModel bound to Knockout) in any child context. This is handy when you are unsure of how many parent/child contexts are above the one you are currently in.

`$parent`

When you are in a child context, this will access the current context's direct parent.

`$parents`

This is similar to the `$parent` variable except that it contains an array of parent contexts to the context you are currently in. `$parents[0]` is the same as `$parent`. Similarly, using `$parents[$parents.length - 1]` is the same as using `$root`.

`$data`

This provides access to the current object your context is in. This is quite useful when you are in a context that is a variable and contains no properties.

\$index

This is only available within the `foreach` binding and contains an integer that represents the current position of the loop starting at 0 and going to length -1.

foreach Binding

The `foreach` binding is used to repeat a section of HTML on an array of objects. In [Example 3-1](#), a list of books that I've written is displayed in an unordered list.

Example 3-1. A list of books using the `foreach` binding

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>
  <ul>
    <!-- ko foreach: books -->
      <li data-bind="text: $data"></li>
    <!-- /ko -->
  </ul>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    var viewModel = function() {
      var self = this;

      self.books = [
        'Rapid Application Development With CakePHP',
        '20 Recipes for Programming MVC 3: Faster, Smarter Web Development',
        '20 Recipes for Programming PhoneGap:
        Cross-Platform Mobile Development for Android and iPhone'
      ];

      ko.applyBindings(viewModel);
    };
  </script>
</body>
</html>
```

[Example 3-1](#) takes several early things that were mentioned and shows them in action.

The `foreach` binding was not bound to an HTML tag; instead, it is placed inside HTML comments. I find this quite convenient when using a `foreach` binding because it avoids an unnecessary element to wrap the HTML that I want repeated for each element in the array.

In the ViewModel, an array called books was created. This contains the name of each book. Because there are no child properties to access inside the books array, the `li` tag is using the `$data` variable that is provided to my child context.

Another common use for the `foreach` binding is to display a list in an HTML table.

Example 3-2 extends the previous books array to contain additional information and display it in a table.

Example 3-2. A table of books using the `foreach` binding

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <table>
    <thead>
      <tr>
        <th>Title</th>
        <th>ISBN</th>
        <th>Published</th>
      </tr>
    </thead>
    <tbody data-bind="foreach: books">
      <tr>
        <td data-bind="text: title"></td>
        <td data-bind="text: isbn"></td>
        <td data-bind="text: $parent.formatDate(publishedDate)"></td>
      </tr>
    </tbody>
  </table>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel() {
      var self = this;

      self.books = [
        {
          title: 'Rapid Application Development With CakePHP',
          isbn: '1460954394',
          publishedDate: '2011-02-17'
        }, {
          title: '20 Recipes for Programming MVC 3:
          Faster, Smarter Web Development',
          isbn: '1449309860',
          publishedDate: '2011-10-14'
        }, {
```

```

        title: '20 Recipes for Programming PhoneGap:
Cross-Platform Mobile Development for Android and iPhone',
        isbn: '1449319548',
        publishedDate: '2012-04-06'
    }
];

self.formatDate = function(dateToFormat) {
    var months = new Array("January", "February", "March", "April",
        "May", "June", "July", "August", "September", "October",
        "November", "December");

    var d = new Date(dateToFormat);
    return months[d.getMonth()] + ' ' + d.getDate() + ', ' +
        d.getFullYear();
};

var viewModel = new ViewModel();
ko.applyBindings(viewModel);
</script>
</body>
</html>

```

Example 3-2 introduces several new things as well. First, the ViewModel is no longer being assigned directly to a variable. It is now being instantiated as a class, and then passes the resulting variable to the `ko.applyBindings` function.

The books array has been extended to contain multiple properties: `title`, `isbn`, and `publishedDate`. These properties are then bound in the `td` tag within the table. Notice how the `foreach` binding has changed the context to be relative to the books array and the properties are being accessed by name without referencing the books array.

The “date published” `td` tag is slightly different than the previous two. It is being data-bound to a function that accepts a date as a parameter. Because the function is not contained within the books array, it needs to be accessed using the `$parent` variable because it is outside the current context.



\$parent Versus \$root

In the previous example, `$parent` is interchangeable between `$root` and `$parents[0]`. I have experienced that even though `$root` provides quick access to the topmost parent, it is better to use the more relative-based `$parent` or `$parents` arrays.

As code is refactored in the future, you might find that your existing HTML is reused in a different context where `$root` may not be the same as `$parent`, and this gives you less chance of accessing the wrong context.

foreach Callbacks (Events)

The `foreach` binding contains several callback methods that can be executed by Knockout after certain events happen:

`afterRender`

This is called when the `foreach` first finishes initializing and every time an element is added to the array.

`afterAdd`

This is called every time an element is added to the array. Unlike `afterRender`, this is not called when the array is first initialized.

`beforeRemove`

This is called when an item is removed from the array. This is often used to animate a removal of an item.

`beforeMove`

This is called when an item is moved within the array. It's another great opportunity to begin an animation or add an effect to the element being interacted with.

`afterMove`

This is called after the item is moved within the array. Much like the `beforeMove`, this would be the opportunity to finish any effects on the moved element.

To demonstrate how one of these callbacks is used, in [Example 3-3](#), I will enhance [Example 3-2](#) to include an image. The `afterRender` will fetch the image from the server to help it prevent blocking the page being rendered while an image is loaded.

Example 3-3. Load image with the `afterRender` callback

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
```

```

<body>

  <table>
    <thead>
      <tr>
        <th>Thumbnail</th>
        <th>Title</th>
        <th>ISBN</th>
        <th>Published</th>
      </tr>
    </thead>
    <tbody data-bind="foreach: { data: books, afterRender: loadImage }">
      <tr>
        <td></td>
        <td data-bind="text: title"></td>
        <td data-bind="text: isbn"></td>
        <td data-bind="text: $parent.formatDate(publishedDate)"></td>
      </tr>
    </tbody>
  </table>

  <script type='text/javascript' src='js/jquery.js'></script>
  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel() {
      var self = this;

      self.books = [
        {
          title: 'Rapid Application Development With CakePHP',
          isbn: '1460954394',
          publishedDate: '2011-02-17',
          image:
            'http://ecx.images-amazon.com/images/I/41JC54HEroL._AA160_.jpg'
        }, {
          title: '20 Recipes for Programming MVC 3:
          Faster, Smarter Web Development',
          isbn: '1449309860',
          publishedDate: '2011-10-14',
          image:
            'http://ecx.images-amazon.com/images/I/51LpqnDq8-L._AA160_.jpg'
        }, {
          title: '20 Recipes for Programming PhoneGap:
          Cross-Platform Mobile Development for Android and iPhone',
          isbn: '1449319548',
          publishedDate: '2012-04-06',
          image:
            'http://ecx.images-amazon.com/images/I/51AkFkNeUxL._AA160_.jpg'
        }
      ]
    };
  </script>

```

```

        self.loadImage = function(element, index, data) {
            $('#image_' + index.isbn).attr('src', index.image);
        };

        self.formatDate = function(dateToFormat) {
            var months = new Array("January", "February", "March", "April",
                "May", "June", "July", "August", "September", "October",
                "November", "December");

            var d = new Date(dateToFormat);
            return months[d.getMonth()] + ' ' + d.getDate() + ', ' +
                d.getFullYear();
        };
    };

    var viewModel = new ViewModel();
    ko.applyBindings(viewModel);
</script>
</body>
</html>

```

Example 3-3 is quite similar to Example 3-2 in terms of structure, with several important changes.

First, the `foreach` data binding has been changed to include the `afterRender` callback. It will call the function `loadImage`.

Second, the table was updated to include a new column for the thumbnail. By default, I put a placeholder image in place. Then, using an attribute binding, I set the `id` of the image to include a unique identifier that will be used to swap out the image.

Third, the `books` array was updated to include a new property called `image` that contains the full path to the book image from Amazon.

And finally, the `loadImage` function was created. This uses a bit of jQuery to access the image by the uniquely created ID, and it changes the source of the image to the location provided in the `books` array.

For this example to function correctly, you need to [download the jQuery library](#). Once downloaded, place it inside the previously created `js` folder. For simplicity, I renamed the file `jquery.js`. Feel free to leave it as is, but be sure to update the location to the file in the HTML.

with Binding

The `with` binding is similar to the `foreach` binding in that it creates a new child context. Everything within the binding is now relative to the variable to which it is bound. The major difference between the two is that `with` is a single object with multiple properties, whereas the `foreach` repeats the HTML in the binding.

Example 3-4 extends the listing of books to be a single book detail page and leverages the `with` binding to prevent repetitive prefixes of the book variable.

Example 3-4. Book details using the `with` binding

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>
  <div id="book" data-bind="with: book">
    <h1 data-bind="text: title"></h1>
    <h2>Published on <span data-bind="text: $parent.formatDate(publishedDate)">
    </span></h2>
    <p data-bind="text: synopsis"></p>
  </div>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel(book) {
      var self = this;

      self.book = book;

      self.formatDate = function(dateToFormat) {
        var months = new Array("January", "February", "March", "April",
          "May", "June", "July", "August", "September", "October",
          "November", "December");

        var d = new Date(dateToFormat);
        return months[d.getMonth()] + ' ' + d.getDate() + ', ' +
          d.getFullYear();
      };

      var book = {
        title: 'Rapid Application Development With CakePHP',
        synopsis: '...',
        isbn: '1460954394',
        publishedDate: '2011-02-17'
      };

      var viewModel = new ViewModel(book);

      ko.applyBindings(viewModel);
    }
  </script>
</body>
</html>
```


In this example, a book object is created that contains a few additional properties that are not available on the listing page of books. This is passed to the constructor of the ViewModel and used within the with binding.

Unlike the foreach binding, the HTML is not repeated, as there is only a single book to display.

In comparison, as shown in [Example 3-5](#), if you didn't use the with data binding, the same result could be accomplished with the following HTML.

Example 3-5. Book details without the with binding

```
<div id="book">
  <h1 data-bind="text: book.title"></h1>
  <h2>Published on <span data-bind="text: formatDate(book.publishedDate)">
</span></h2>
  <p data-bind="text: book.synopsis"></p>
</div>
```

When the with binding is not used, notice how the HTML elements are data-bound using the full variable path (e.g., `book.title`). The other difference is that the `formatDate` function no longer needs to be prefixed with `$parent` because a child context was not created, and it is still in the root context.

In scenarios like this, using the with binding is a simple convenience to avoid repeatedly prefixing the child properties with the variable.

CHAPTER 4

Dynamically Changing Properties

So far, we've only touched on a small piece of the KnockoutJS framework. In this chapter, we'll start taking advantage of properties that change dynamically through user interaction.

Knockout calls these properties *observables*. When you define a variable or property as an observable, Knockout tracks when it changes. This can be used in a variety of different ways that we'll see in the next few chapters.

Defining an Observable

There are three different types of observables that are most commonly used. The first, shown in [Example 4-1](#), is an *observable variable*.

Example 4-1. Creating an observable variable

```
var myObservable = ko.observable();  
myObservable('Hello');  
  
alert(myObservable());
```

To create an observable, assign the `ko.observable` function to the variable. A default value can be specified in the constructor of the call. Knockout then converts your variable into a function and tracks when the value changes, in order to notify the UI elements associated with the variable.



Accessing an Observable

After an observable is defined, it needs to be called like a function in order to get or set its value. If you try setting it directly as if it were a variable, the observable would be destroyed.

In [Example 4-1](#), notice how the setting of the variable is done like a function. When accessing the value for the *alert* statement, this is also done. If you were to attempt to alert the observable without the brackets, it would output a large chunk of JavaScript code that contains the Knockout observable function implementation.

The second type of observable, as shown in [Example 4-2](#), is an *observable array*.

Example 4-2. Creating an observable array

```
var myObservableArray = ko.observableArray([]);
myObservableArray.push('Hello');
```

In [Example 4-2](#), the array is instantiated as an empty array by passing two square brackets in the constructor.

Just like observable variables, when elements are added or removed from the array, Knockout notifies elements that are subscribed to the notifications.

An observable array is great for use with tables where elements are being dynamically added and removed.

The final type of observable, as shown in [Example 4-3](#), is a *computed observable*. This is slightly different than the previous two types in that a computed observable is commonly used to combine one or more observables into a single object.

Example 4-3. Creating a computed observable

```
self.firstName = ko.observable('Steve');
self.lastName = ko.observable('Kennedy');

self.fullName = ko.computed(function() {
    return 'Hello ' + self.firstName() + ' ' + self.lastName();
});
```

Once the ViewModel is bound to Knockout, the computed function is executed. For each observable that is used within that function, it subscribes to any change events to that variable. When it changes, Knockout knows that the computed variable should be updated.

pureComputed Observables

As of Knockout version 3.2 (which this book uses), a new type of observable called *pureComputed* observable was introduced (see [Example 4-4](#)). It is quite similar to the *computed* observable with several performance and memory improvements. The name is borrowed from the **Pure function** programming term.

A common example of when to use this type of observable is in [Example 4-3](#) where the first and last name were concatenated into a computed observable.

Example 4-4. Creating a pureComputed observable

```
self.firstName = ko.observable('Steve');
self.lastName = ko.observable('Kennedy');

self.fullName = ko.pureComputed(function() {
    return 'Hello ' + self.firstName() + ' ' + self.lastName();
});
```

The observables are treated differently because when there are no elements listening for changes to the computed variable, they are placed in *sleeping* mode versus *listening* mode. While in sleeping mode, Knockout disposes all dependencies and re-evaluates the content when it is read—unlike listening mode, which manages references to all subscribers and ensures the value is up-to-date prior to first access.

The previous example follows both rules for being a pure function:

- Given the same input, it will output the same result.
- No side effects occur because of the function executed.

The second rule is probably the most important when deciding whether to use a computed or a pureComputed observable with Knockout. Within your observable, if you need to execute other code, then you should use a computed observable to ensure it is in listening mode instead of sleeping mode. See the difference in [Example 4-5](#).

Example 4-5. Computed versus pureComputed

```
<!DOCTYPE html>
<html>
<head>
    <title>Data Binding with KnockoutJS</title>
</head>
<body>

    <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
    <script>
        function ViewModel() {
            var self = this;
```

```

    self.firstName = ko.observable('Steve');
    self.lastName = ko.observable('Kennedy');

    self.pureComputedExecutions = 0;
    self.computedExecutions = 0;

    self.pureComputedFullName = ko.pureComputed(function() {
        // This is not recommended
        self.pureComputedExecutions++;
        return 'Hello ' + self.firstName() + ' ' + self.lastName();
    });

    self.computedFullName = ko.computed(function() {
        self.computedExecutions++;
        return 'Hello ' + self.firstName() + ' ' + self.lastName();
    });
};

var viewModel = new ViewModel();
ko.applyBindings(viewModel);

alert('Pure computed executions: ' + viewModel.pureComputedExecutions);
alert('Computed executions: ' + viewModel.computedExecutions);
</script>
</body>
</html>

```

Example 4-5 breaks the pure function rule by placing a side effect within it; however, it is demonstrating the difference between sleep and listening mode.

When this example is run, after the ViewModel has been bound to the view, two alert messages are shown that display the number of times the pureComputed and computed functions are called.

Because the computed observables are not data-bound to anything, the pureComputed is in sleeping mode; thus, the computed function has never been accessed, and the counter is at 0. However, the computed function is automatically evaluated on data binding to set its listeners, causing the counter to increase to 1.

If you were to data-bind to both computed observables, it would result in both counters being 1 because the pureComputed is now in listening mode as it has a subscriber.

Showing and Hiding Elements

Example 4-6 combines the use of an observable variable with a new data binding called visible. The visible data binding sets the CSS property display to either block or none depending on the results of the condition used in the binding.

In [Example 4-6](#), when the button is pressed by the user, it triggers the observable variable to change, which causes the UI to reveal HTML elements that were previously hidden.

Example 4-6. Using the visible data bind with an observable

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>
  <button type="button" data-bind="click: updateObservable">Click me</button>

  <div data-bind="visible: showExtraData" style="display: none">
    Now you see me!
  </div>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel() {
      var self = this;

      self.showExtraData = ko.observable(false);

      self.updateObservable = function() {
        self.showExtraData(!self.showExtraData());
      };

      var viewModel = new ViewModel();
      ko.applyBindings(viewModel);
    }
  </script>
</body>
</html>
```

Let's start with the HTML. A button is created and data-bound to an event, click. When the user clicks the button, Knockout calls the function `updateObservable`.

Below the button is a `div` tag. This element is using the visible data binding. When the result of the condition is `false`, the element is hidden. When it changes to `true`, it becomes visible.



CSS Style display: none

You might notice that in the above HTML, I also added `style="display: none"` to the div. I did this to prevent a flicker when the page is being loaded. A flicker would occur because without the `display: none` style, the user would temporarily see the contents within the div element. When the Knockout bindings execute, the element would be hidden.

Next, let's explore the JavaScript ViewModel. The first thing I did was create an observable variable called `showExtraData` and default it to `false` (hidden by default). And finally, the `updateObservable` function is defined, which Knockout calls when the button is clicked. This function simply inverts the value of the `showExtraData` variable. The first time the button is clicked, the HTML within the div is revealed. If it is clicked again, the text is hidden again.

Adding and Removing Elements

The `if` and `ifnot` data bindings are quite similar to the previous visible data binding. The difference between the two is that, unlike the visible binding setting a CSS style to show or hide the element, `if` and `ifnot` physically add or remove the elements from the Document Object Model (DOM). See [Example 4-7](#).

Example 4-7. If data binding with an observable

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>
  <button type="button" data-bind="click: updateObservable">Click me</button>

  <!-- ko if: showExtraData -->
  <div>
    Now you see me!
  </div>
  <!-- /ko -->

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel() {
      var self = this;

      self.showExtraData = ko.observable(false);

      self.updateObservable = function() {
```



```

        self.showExtraData(!self.showExtraData());
    };
};

var viewModel = new ViewModel();
ko.applyBindings(viewModel);
</script>
</body>
</html>

```

Example 4-7 is nearly identical to **Example 4-6**; in fact, the JavaScript ViewModel is untouched. The HTML contains the slight difference. Instead of the `div` containing a data binding for `visible`, I've used the `if` data binding inside an HTML comment.

If you inspect the HTML when the page is loaded, you will notice that the `div` tag is nowhere to be found. Clicking the button will cause the HTML to be dynamically inserted into the DOM.



Adding and Removing from the DOM

If an element is being added to the DOM because of a user interaction that had one or more JavaScript side effects performed on it during the original page load, these would need to be executed after adding them back to the DOM.

For example, if you have a field that is linked to a jQuery date picker, JavaScript is required to initialize it. This needs to be executed after the element is added to the DOM.

In a scenario like this, it might be more prudent to use the `visible` data binding because the elements will remain in the DOM and can be initialized upon the document load.

I find using the `if` and `ifnot` data bindings very convenient when I want to remove content that I only want the user to be able to access in specific scenarios.

Example 4-8. ifnot data binding

```
ifnot: !showExtraData()
```

Example 4-7 could be accomplished using `ifnot` by updating the binding to invert the condition. Instead of adding the elements when the condition evaluates to `true`, the `ifnot` adds the elements when the condition evaluates to `false`.



The Use of Brackets and Observables

In case you didn't notice, in the previous example, I had to execute the `showExtraData` observable like it was a function by adding brackets at the end. All of the previous examples did not require the brackets because Knockout intuitively knew to execute the observable; however, because I added the `!` to the state when false, I had to tell Knockout to execute the observable and then apply the `!` statement.

In [Chapter 2](#), when demonstrating conditional data binding, if the `id` property were an observable, the example would be updated as follows: `id() == 0`, thus telling Knockout to execute the `id` observable and compare it to the value 0.

When to Use Observables

Observables are extremely powerful, and as Peter Parker is told by his Uncle Ben: “With great power comes great responsibility.” (Yes, I must always use this Spider-Man quote when this comes up.)

When an observable is first defined, Knockout initializes the variable as a function to track any changes to the value. This contains a certain amount of processing overhead.

It's important to make conscious decisions about what you define as an observable. If the value can potentially change (either programmatically or from user interaction), an observable property is completely valid. However, if you have four properties and only two of them will change, and the other properties are needed but will never change, there is no need to make them observed (see [Example 4-9](#)).

Example 4-9. Object that is partially observed

```
function ViewModel(person) {
    var self = this;

    self.person = {
        id: person.id,
        firstName: ko.observable(person.firstName),
        lastName: ko.observable(person.lastName),
        status: person.status
    };
};

var person = {
    id: 1,
    firstName: 'Steve',
    lastName: 'Kennedy',
```

```
        status: 'Active'  
    };  
  
    var viewModel = new ViewModel(person);  
    ko.applyBindings(viewModel);
```

Example 4-9 implies that I can only change the first and last name of a person object. The `id` and `status` cannot be changed; however, they can still be used in a Knockout data binding for display purposes.

CHAPTER 5

Working with Forms

Form Data Bindings

Knockout provides several different bindings that work with specific form elements.

- The `value` binding is used with `input`, `select`, and `textarea` form inputs.
- The `textInput` binding is also used with `input` and `textarea` and is quite similar to the `value` binding. When the `textInput` is used, the observable updates with every user interaction, as opposed to the `value` binding, which defaults to updating when the form element changes (typically when the field loses focus). See [Example 5-1](#).
- The `checked` binding is used with checkboxes and radio buttons.
- The `options` binding is used on the `select` form input to populate the list of options available in the drop-down list.
- The `selectedOptions` binding is also used with the `select` form input; more specifically when you are using a multiselect list. This is commonly bound to an observable array, as opposed to an observable variable.
- The `enable` and `disable` bindings work with all form inputs to either enable or disable the form element when the condition results to true or false, respectively.

All of these bindings are what Knockout calls *two-way bindings*. This means that when the form element is changed, your ViewModel property is updated and if you programmatically change the observable, the form element it is binded to will be automatically updated.

Example 5-1. The textInput binding

```

<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <textarea data-bind="textInput: myText"></textarea><br/>
  Characters remaining: <span data-bind="text: charactersRemaining"></span>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel() {
      var self = this;

      self.myText = ko.observable('');

      self.maxCharacters = 140;

      self.charactersRemaining = ko.computed(function() {
        return self.maxCharacters - self.myText().length;
      });
    };

    var viewModel = new ViewModel();
    ko.applyBindings(viewModel);
  </script>
</body>
</html>

```

This example uses the `textInput` binding on a `textarea`. This is bound to the `myText` observable. Just beneath the text box, the total remaining characters are displayed. A `span` element is bound to a computed observable called `charactersRemaining`. Each time the `myText` property is changed, this function is called and automatically updates the remaining characters.

If you were to try the previous example but replace the `textInput` binding with the `value` binding, you would not see the character count updated with each keystroke.

This can be fixed because the `value` binding offers a secondary property called `valueUpdate`. This can be set to any other event that Knockout should listen to in order to trigger updates.

Example 5-2 tells Knockout to update the value when the event `afterkeydown` is triggered, which will provide more instantaneous updates. By default, when using the `value` binding, Knockout is only listening for the `onChange` event.

Example 5-2. Using the value binding with a valueUpdate

```
<textarea data-bind="value: myText, valueUpdate: 'afterkeydown'"></textarea><br/>
```

I advise caution in changing the `valueUpdate` option. While instantaneous updates are nice, many scenarios do not require this. If Knockout is unnecessarily updating observables, application performance will suffer.

Event Data Bindings

In [Chapter 4](#), the first Knockout binding event was used (`click`). Knockout provides several other events that are commonly used with forms:

- The `submit` binding is used on the form element and is triggered when a form is submitted.
- The `click` binding is commonly used on buttons and links, but can be applied to any DOM element that is visible.
- The `hasFocus` binding is commonly used on input elements and is triggered when the DOM element receives user focus.
- The event binding allows you to specify any other DOM event (including the `click` and `submit` bindings), such as `mouseover`, `keypress`, etc.

[Example 5-3](#) is quite similar to a previous example where a list of books are displayed from the array. The example has been trimmed down to just include a title and image.

Example 5-3. A mouseover event

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <img id="current_book" />

  <ul>
    <!-- ko foreach: books -->
      <li data-bind="text: title, event: { mouseover: $parent.loadImage }"></li>
    <!-- /ko -->
  </ul>

  <script type='text/javascript' src='js/jquery.js'></script>
  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel() {
      var self = this;
```

```

self.books = [
  {
    title: 'Rapid Application Development With CakePHP',
    image:
      'http://ecx.images-amazon.com/images/I/41JC54HEroL._AA160_.jpg'
  }, {
    title: '20 Recipes for Programming MVC 3:
    Faster, Smarter Web Development',
    image:
      'http://ecx.images-amazon.com/images/I/51LpqnDq8-L._AA160_.jpg'
  }, {
    title: '20 Recipes for Programming PhoneGap:
    Cross-Platform Mobile Development for Android and iPhone',
    image:
      'http://ecx.images-amazon.com/images/I/51AkFkNeUxL._AA160_.jpg'
  }
];

self.loadImage = function(data, event) {
  $('#current_book').attr('src', data.image);
};

var viewModel = new ViewModel();
ko.applyBindings(viewModel);
</script>
</body>
</html>

```

The books are output in an unordered list. The `li` element is bound to the `title` property. After the text binding, an event binding is used and listens for the mouse over event.

When this event is triggered, it calls the `loadImage` function. This function then populates an image with the value from the `image` property.

Listening for Changes

When observables were introduced, I mentioned how Knockout sends events that notify listeners about changes to the value. Knockout provides a useful *subscribe* function that can be added to an observable. Then, each time the observable is changed, the function is called, allowing programmatic changes to that observable.

Subscribing to observables allows you to easily perform side effects to other properties in your ViewModel when the property's value changes (see [Example 5-4](#)). The following example demonstrates this with two drop-down lists. When the observed value from the first list changes, it dynamically updates the observed value of the second list.

The first example contains a list of countries to select from. It uses the previously mentioned data bindings to populate a select element option, define the value used for the text, and define the values in the drop-down.

The second list contains available states for the selected country. By default, this will be hidden until a country that contains an array of states is selected.

Example 5-4. Subscribing to an observable value change

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <select data-bind="options: availableCountries, optionsText: 'name',
optionsValue: 'id', optionsCaption: 'Select a country...',
value: selectedCountry"></select>

  <select data-bind="options: availableStates, optionsText: 'name',
optionsValue: 'id', value: selectedState, visible: availableStates().length >
0" style="display:none"></select>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel() {
      var self = this;

      self.selectedCountry = ko.observable();
      self.selectedState = ko.observable();

      self.availableCountries = ko.observableArray([
        {
          id: 1, name: 'United States', states: [
            { id: 1, name: 'Alabama' },
            // ...
          ]
        },
        {
          id: 2, name: 'Canada', states: [
            { id: 53, name: 'Alberta' },
            // ...
          ]
        }
      ]);

      self.availableStates = ko.observableArray([]);

      self.selectedCountry.subscribe(function() {
        self.availableStates([]);
```

```

        for (var i = 0; i < self.availableCountries().length; i++) {
            if (self.availableCountries()[i].id == self.selectedCountry()) {
                self.availableStates(self.availableCountries()[i].states);
                break;
            }
        }
    });

};

var viewModel = new ViewModel();
ko.applyBindings(viewModel);
</script>
</body>
</html>

```

The array of countries was populated with an `id`, `name`, and `states` property. The array of states are used to populate the available states when that country is selected from the drop-down list.

This is accomplished by subscribing to the `selectedCountry` observable to which the first drop-down list is bound. When the value changes, the function within `subscribe` is called. This function clears the list of states, and then loops through the available countries until it finds the selected country (comparing the `id` property to the `availableCountry` value). When a match is found, the available states are populated.

Unobtrusive Form Validation

A chapter dedicated to working with forms would be incomplete if it didn't cover validation of the data within a form. Common form validation includes ensuring that required data is populated prior to submitting. It can also include validation of the content based on the data type; for example, an email address requires very specific formatting.

By implementing unobtrusive form validation, this can all occur prior to the form being submitted. There is an excellent jQuery plugin that allows for easy additions of unobtrusive client-side form validation.

Example 5-5 demonstrates how to set up and include several simple form data binds, including the `submit` data bind.

To begin, visit the [jQuery Validation Plugin](#) homepage and download the plugin. This example uses version 1.13. This plugin also requires the jQuery library to be installed. This can be downloaded from the [jQuery site](#) or the *lib* folder in the jQuery validation download.

Copy these two files into the previously created `js` folder where the Knockout library currently resides.

Example 5-5. Unobtrusive form validation

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <form data-bind="submit: validate">
    <input type="text" data-bind="value: firstName"
      placeholder="Enter your first name" required /><br/>
    <input type="text" data-bind="value: lastName"
      placeholder="Enter your last name" required /><br/>
    <input type="email" data-bind="value: email"
      placeholder="Enter your email" required /><br/>
    <input type="submit" />
  </form>

  <script type="text/javascript" src='js/jquery.js'></script>
  <script type="text/javascript" src='js/jquery.validate.min.js'></script>
  <script type="text/javascript" src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel() {
      var self = this;

      self.firstName = ko.observable();
      self.lastName = ko.observable();
      self.email = ko.observable();

      self.validate = function(form) {
        return $(form).validate();
      };

      var viewModel = new ViewModel();
      ko.applyBindings(viewModel);
    }
  </script>
</body>
</html>
```

Example 5-5 creates a form that collects a first name, last name, and email address. These fields are all data-bound to corresponding fields in the ViewModel.

The form also contains a data binding to the `submit` binding that tells Knockout to call the `validate` function that has been created in the ViewModel.

The `validate` function accepts the form as an input parameter. It then calls the `validate` function on the form that the unobtrusive validation automatically added to the form. If the validation succeeds, the form will be submitted; otherwise, an error will be displayed to the user to correct the information.

All three form fields have been tagged `required` and because I specified the email field as type `email`, the form validation ensures that I have a valid email address.

CHAPTER 6

Multiple ViewModels, Custom Data Bindings, and Templates

Binding Multiple ViewModels

Whether you are simply looking to keep your ViewModels nice and clean, or you have a shared ViewModel that appears on each page, there will come a time when you wish to bind multiple ViewModels on a single page.

Knockout makes this quite easy. There is an optional second parameter that can be passed into the `ko.applyBindings` function that tells Knockout to limit its binding within this block of HTML.

Example 6-1 almost looks the same as some of the previous examples where a simple name was being bound. There are a few subtle differences that get this to work.

Example 6-1. Binding two ViewModels on a single page

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <div id="viewModel1">
    <h1 data-bind="text: name"></h1>
  </div>

  <div id="viewModel2">
    <h1 data-bind="text: name"></h1>
  </div>
```

```

<script type='text/javascript' src='js/knockout-3.2.0.js'></script>
<script>
    function ViewModel(name) {
        var self = this;

        self.name = name;
    };

    var viewModel1 = new ViewModel('Steve Kennedy');
    ko.applyBindings(viewModel1, document.getElementById('viewModel1'));

    var viewModel2 = new ViewModel('Mike Wilson');
    ko.applyBindings(viewModel2, document.getElementById('viewModel2'));
</script>
</body>
</html>

```

First, there are two `h1` tags placed inside their own `div` tag. This `div` tag contains an `id` attribute that is important for this example to function properly.

In the JavaScript, the same `ViewModel` is instantiated twice, each with a different name as the input. This variable is then passed to the `ko.applyBindings` as normal; however, this time a second parameter is passed as well. The `div` element is passed so Knockout will only bind within that element.

You can confirm this by copying one of the `h1` tags and placing it outside of the `div`. No text will be displayed because Knockout does not perform any data bindings.

Binding to a WYSIWYG Editor

Knockout comes with many great, built-in bindings; however, there are times when they don't quite solve the problem at hand. Luckily, Knockout allows you to build a custom binding that can be bound to any HTML property.

A perfect example of this is a WYSIWYG editor such as the very popular (and free) TinyMCE.

TinyMCE converts any ordinary `textarea` into a sophisticated HTML editor. Unfortunately, TinyMCE doesn't actually change the contents of the `textarea` to which we would normally apply the `value` binding.

In [Example 6-2](#), `textarea` will be bound to a custom Knockout binding called `tinymce`. When the user works with the editor, a live preview is bound to a `div` tag using the `html` binding. A button is also bound to a function that will reset the contents of the editor to nothing.

To begin, visit the [TinyMCE download page](#) and get the version that is built with the jQuery package. This book uses version 4.1.5.

Example 6-2. textarea bound to TinyMCE

```
<!DOCTYPE html>
<html><head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>
  <form>
    <textarea data-bind="tinymce: htmlText"></textarea>
  </form>

  <button type="button" data-bind="click: resetContent">Reset content</button>

  <h2>Preview</h2>
  <div data-bind="html: htmlText"></div>

  <script type='text/javascript' src='js/jquery.js'></script>
  <script type='text/javascript' src='js/tinymce/jquery.tinymce.min.js'></script>
  <script type='text/javascript' src='js/tinymce/tinymce.min.js'></script>
  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script type='text/javascript' src='js/kobinding.js'></script>    <script>
    function ViewModel() {
      var self = this;

      self.htmlText = ko.observable();

      self.resetContent = function() {
        self.htmlText('');
      };
    };

    var viewModel = new ViewModel();
    ko.applyBindings(viewModel);
  </script>
</body>
</html>
```

So far, this looks identical to many previous examples shown. A `textarea` is bound to a property called `htmlText`. The button is bound to a function called `resetContent`, which simply sets the `htmlText` property to an empty string. The preview is a `div` that is data-bound via the `html` binding to the same `htmlText` property.

Several scripts are included: jQuery, TinyMCE, and the newly created Knockout binding. A new file called *kobinding.js* should be created and placed inside the previously created *js* folder. The contents of this file are shown in [Example 6-3](#).

Example 6-3. The *tinymce* binding, *kobinding.js*

```
ko.bindingHandlers.tinymce = {
  init: function (element, valueAccessor, allBindingsAccessor) {
    var tinymceOptions = { setup: function (editor) {
      editor.on('change', function (event) {
        valueAccessor()(event.target.getContent());
      });
    }
  };

  $( element ).text( valueAccessor()() );

  setTimeout( function() {
    $( element ).tinymce(tinymceOptions);
  }, 0 );

  ko.utils['domNodeDisposal'].addDisposeCallback( element, function() {
    $( element ).tinymce().remove();
  });
},
'update': function( element, valueAccessor, allBindings ) {
  var tinymce = $( element ).tinymce(), value = valueAccessor()();

  if ( tinymce ) {
    if ( tinymce.getContent() !== value ) {
      tinymce.setContent( value );
    }
  }
}
};
```

To begin, you'll notice that two properties are defined for the *tinymce* binding: *init* and *update*.

The *init* property is called upon first initialization (i.e., when *ko.applyBindings* is first called). This does a few things. It sets up the TinyMCE options and binds the textarea to TinyMCE.

In the setup of TinyMCE, I created an inline setup function that listens for the *change* event that is triggered by TinyMCE. Every time TinyMCE fires the *change* event, the observable that is bound to the custom binding is updated with the contents of the TinyMCE editor.

The *update* property is triggered whenever the observable changes via code (i.e., in the previous example, from clicking the reset button). Inside this function, it gets the TinyMCE editor that the observable is bound to and sets the content of TinyMCE to the current value of the observable.



valueAccessor()

If you are wondering why two sets of parentheses are being used, the input property `valueAccessor` is a JavaScript function which (when executed) provides access to the current model property value. This is the first set of parentheses.

The second set of parentheses is required because the `model` property is an observed property, meaning it is a function and needs to be executed to access the current value.

Binding to a Knockout Template

Code organization and code reuse are important factors on almost any size project. Knockout helps with organizing or reusing your HTML templates by creating a Knockout template.

Even if you don't use a template, the underlying Knockout functionality converts the HTML structure in `foreach`, `with`, and `if` bindings into one.

In [Example 6-4](#), I am going to update an example from [Chapter 3](#) that outputs a list of books in a table. The HTML that is repeated within the `foreach` binding will be extracted into a template, allowing for easy reuse.

Example 6-4. Using a template with a foreach binding

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <table>
    <thead>
      <tr>
        <th>Title</th>
        <th>ISBN</th>
        <th>Published</th>
      </tr>
    </thead>
    <tbody data-bind="template: { name: 'book-template', foreach: books }">
    </tbody>
  </table>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>

  <script type="text/html" id="book-template">
    <tr>
      <td data-bind="text: title"></td>
```

```

        <td data-bind="text: isbn"></td>
        <td data-bind="text: $parent.formatDate(publishedDate)"></td>
    </tr>
</script>

<script>
    function ViewModel() {
        var self = this;

        self.books = [
            {
                title: 'Rapid Application Development With CakePHP',
                isbn: '1460954394',
                publishedDate: '2011-02-17'
            }, {
                title: '20 Recipes for Programming MVC 3:
                Faster, Smarter Web Development',
                isbn: '1449309860',
                publishedDate: '2011-10-14'
            }, {
                title: '20 Recipes for Programming PhoneGap:
                Cross-Platform Mobile Development for Android and iPhone',
                isbn: '1449319548',
                publishedDate: '2012-04-06'
            }
        ];

        self.formatDate = function(dateToFormat) {
            var months = new Array("January", "February", "March", "April",
            "May", "June", "July", "August", "September", "October",
            "November", "December");

            var d = new Date(dateToFormat);

            return months[d.getMonth()] + ' ' + d.getDate() + ', ' +
            d.getFullYear();
        };
    };

    var viewModel = new ViewModel();
    ko.applyBindings(viewModel);
</script>
</body>
</html>

```

This example is almost identical to the earlier example with two key modifications.

First, the foreach binding is now wrapped within a template binding. The template binding tells Knockout to call the book-template for each element in the foreach binding.

Second, the HTML code that was previously within the `foreach` binding has been moved inside a `script` tag that contains a different script type of `text/html`, and the `id` attribute is set to `book-template`.

Templates provide an excellent way to improve organization and reusability of blocks of HTML code. The larger your project, the more important this will become!

Enhancing Observables

Extending Observables

Observables by themselves can accomplish quite a bit. Of course, there are times with every framework that it simply cannot do everything you want. Knockout provides the ability to write custom functions to extend any observable.

Example 7-1 will extend (pun intended) the previous example that counted characters to prevent the text input from entering more than the allotted amount.

Example 7-1. Extending an observable

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <textarea data-bind="textInput: myText"></textarea><br/>
  Characters remaining: <span data-bind="text: charactersRemaining"></span>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script type='text/javascript' src='js/maxCharacters.js'></script>
  <script>
    function ViewModel() {
      var self = this;

      self.maxCharacters = 140;

      self.myText = ko.observable('')
        .extend({maxCharacters: self.maxCharacters});
```

```

        self.charactersRemaining = ko.computed(function() {
            return self.maxCharacters - self.myText().length;
        });
    };

    var viewModel = new ViewModel();
    ko.applyBindings(viewModel);
</script>
</body>
</html>

```

In the previous example, the `myText` observable has been extended to call the `maxCharacters` function, passing in a variable that contains the max characters to allow (see [Example 7-2](#)).

The `maxCharacters` function has been placed inside a file called *maxCharacters.js* and placed within the previously created *js* folder. This file is included immediately after the Knockout framework.

Example 7-2. Max characters extend function

```

ko.extenders.maxCharacters = function(target, max) {
    var result = ko.pureComputed({
        read: target,
        write: function(newValue) {
            var current = target();

            // only write the newValue if it less or equal to the
            // max characters allowed
            if (newValue.length <= max)
                target(newValue);
            // set it back to the previous value
            else {
                target(current);
                target.notifySubscribers(current);
            }
        }
    }).extend({ notify: 'always' });

    return result;
};

```

The extender function looks similar to computed observables in that it deals with both the read and write of the observable that is being extended.

Whenever the observable is about to change, Knockout calls the `write` function. This function checks if the new value is less than the max characters allowed. If it is, it writes the value into the observable.

If it's not, it resets the observable back to the previous value. After this is done, it tells the observable to notify all subscribers that the value has changed. This is required because technically the value hasn't changed, but by telling the subscribers that it has, it causes Knockout to update the contents of the text box and remove the newly added characters when the value exceeds the maximum length.

Adding Custom Functions to Observables

Similar to extending observables, you can also attach a custom function to them. Knockout uses inheritance for observable types, so where you attach your function will determine what types of observables have access to it.

At the top of the hierarchy is the `subscribable` class; all types of observables inherit from this class. If you wish to add a custom function that is available to any observable type, you would attach it to this class.

From `subscribable`, the `observable` property and `computed observable` branch off. The `observableArray` type further extends the `observable` property. This means if you attach a function to the `computed`, it will only be available to that type. However, if you add it to `observable`, it will also be available to the `observableArray`.

In [Example 7-3](#), the books are back. This time, a list of books will be displayed in an unordered list. Each book will contain a checkbox that is data-bound to the `owned` observable property of the book.

A function is then added to the `observableArray` called `booksOwned` that will create a new list from the main `books` array that only contains books where the `owned` property is `true`.

This list is then displayed under a separate unordered list that shows all of the books you own.

Example 7-3. Adding a function to an `observableArray`

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  List of books
  <ul>
    <!-- ko foreach: books -->
    <li>
      <input type="checkbox" data-bind="attr: { id: isbn }, checked: owned "/>
      <label data-bind="attr: { for: isbn }, text: title"></label>
```

```

        </li>
    <!-- /ko -->
</ul>

Books you own
<ul>
    <!-- ko foreach: booksOwned -->
        <li data-bind="text: title"></li>
    <!-- /ko -->
</ul>

<script type='text/javascript' src='js/knockout-3.2.0.js'></script>
<script type='text/javascript' src='js/booksOwned.js'></script>
<script>
    function ViewModel() {
        var self = this;

        self.books = ko.observableArray([
            {
                title: 'Rapid Application Development With CakePHP',
                isbn: '1460954394',
                owned: ko.observable(false)
            }, {
                title: '20 Recipes for Programming MVC 3:
                Faster, Smarter Web Development',
                isbn: '1449309860',
                owned: ko.observable(false)
            }, {
                title: '20 Recipes for Programming PhoneGap:
                Cross-Platform Mobile Development for Android and iPhone',
                isbn: '1449319548',
                owned: ko.observable(false)
            }
        ]);

        self.booksOwned = self.books.booksOwned('owned', true);
    };

    var viewModel = new ViewModel();
    ko.applyBindings(viewModel);
</script>
</body>
</html>

```

When the checkbox is selected, it will set the owned property to true. Inside the ViewModel, another array is defined that is called booksOwned (see [Example 7-4](#)). This is a subset of the books array that calls the booksOwned custom function that has been added to the observableArray. This function accepts two inputs, the property that will be used in the comparison and the required value to be included in the new list.

The previous example includes a new script called *booksOwned.js* that is placed within the previously created *js* folder.

Example 7-4. booksOwned custom function

```
ko.observableArray.fn.booksOwned = function(property, value) {
  return ko.pureComputed(function() {
    var allItems = this();
    var matchingItems = [];

    for (var i = 0; i < allItems.length; i++) {
      var current = allItems[i];

      if (ko.unwrap(current[property]) === value)
        matchingItems.push(current);
    }

    return matchingItems;
  }, this);
};
```

This function creates two arrays: *allItems* and *matchingItems*. *allItems* contains the original array. This array is then looped through, and using the inputs to this function, the element in the array is compared to the expected value. If there is a match, it is added to the second array, *matchingItems*. This array is then returned.

Because this function is wrapped within a computed function, it is set up to subscribe to any changes to the *observableArray* or any properties within that array that are being observed. So whenever either of those change, this function is executed, and the list of books owned is updated.

Rate-Limiting Observables

There might be a scenario where you want to delay responding to an observable that has changed. Knockout contains a built-in extended function that allows you to limit how often subscribers to the observable are notified.

Because there might be several different ways you wish to limit, Knockout has provided a few different methods.

In its simplest form, as shown in [Example 7-5](#), you can extend the observable and set a timeout value (in milliseconds) that Knockout should wait.

Example 7-5. Delay updates by one second

```
myObservable.extend({ rateLimit: 1000 });
```

This example can also be extended (no pun intended this time) to not simply wait the given milliseconds, but instead wait a given milliseconds after changes have stopped. See [Example 7-6](#).

Example 7-6. Update one second after changes have stopped

```
myObservable.extend( {
  rateLimit: {
    timeout: 1000,
    method: "notifyWhenChangesStop"
  }
});
```

In [Example 7-7](#), I thought it would be neat to create our own autocomplete. By using the `rateLimit` when the value changes on delay, a list of programming languages will be searched. Any matches will be added to an `observableArray` and displayed in an unordered list.

Example 7-7. Autocomplete using Knockout

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <label for="tags">Filter a tag: </label>
  <input id="tags" data-bind="textInput: tag"><br/><br/>

  Tags Matching:
  <ul>
    <!-- ko foreach: matchedTags -->
      <li data-bind="text: $data"></li>
    <!-- /ko -->
  </ul>

  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel() {
      var self = this;

      self.availableTags = [
        "ActionScript", "AppleScript", "Asp",
        "BASIC", "C", "C++", "Clojure",
        "COBOL", "ColdFusion", "Erlang",
        "Fortran", "Groovy", "Haskell",
        "Java", "JavaScript", "Lisp",
        "Perl", "PHP", "Python",
        "Ruby", "Scala", "Scheme"
```

```

];

self.matchedTags = ko.observableArray([]);

self.tag = ko.observable().extend({
  rateLimit: {
    timeout: 1000,
    method: "notifyWhenChangesStop"
  }
});

self.tag.subscribe(function(value) {
  self.matchedTags.removeAll();

  if (value !== '') {
    for (var i = 0; i < self.availableTags.length; i++) {
      if (self.availableTags[i].toLowerCase().indexOf(value) >= 0)
        self.matchedTags.push(self.availableTags[i]);
    }
  }
});

};

var viewModel = new ViewModel();
ko.applyBindings(viewModel);
</script>
</body>
</html>

```

In the previous example, two arrays are created. One contains the list of tags to search. The other is an `observableArray` that will be updated and contains tags that match what the user types.

A tag observable is also created and extended with a `rateLimit` of one second after it stops changing. This property is then subscribed to. When the property changes (after the delay), this function is called. It clears the `matchedTags` array and loops through the list of tags and adds any matches to the array.

If you run this example and type *a* into the text box that is data-bound to the delayed observed property, after one second the list of matched tags will be populated with several matches.

It's not quite as pretty as jQuery's autocomplete, but I like the better rate limiting that it provides!

Server-Side Interaction

Knockout doesn't include any functions to perform server-side interaction. It is completely up to you. You can use jQuery, vanilla JavaScript, or any other library that you prefer.

I suggest using jQuery because some of the previous examples are already using it, which means it is most likely already included in your project, thus adding no additional override.

Sending Data

When it comes to sending data, I typically use a few different formats depending on whether I'm interacting with a **REST API** or if I'm **POSTing** a standard form. There are two major differences when changing between types:

1. The content-type of the **AJAX** request is different. The default form when performing an AJAX request using jQuery is *application/x-www-form-urlencoded*. This would be used for a standard form POST. A typical content-type when working with a REST API would be *application/json* when working in **JSON**.
2. The content of the form is also different. When performing a standard form POST, the data would be serialized as a standard JavaScript key/value pair of data. Whereas, when interacting with a JSON REST API, the form data would be serialized to JSON. Knockout does provide helpful functions to unwrap observables and convert them to the necessary types that will be explored in the upcoming examples.

In **Example 8-1**, I will create a standard form that leverages several Knockout bindings. When the form is submitted, it will be done via AJAX using the standard form POST.

Example 8-1. Form POST via AJAX

```

<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>
  <form data-bind="submit: validateAndSave, with: person">
    <input type="text" data-bind="value: firstName"
      placeholder="Enter your first name" required /><br/>
    <input type="text" data-bind="value: lastName"
      placeholder="Enter your last name" required /><br/>
    <input type="email" data-bind="value: email"
      placeholder="Enter your email" required /><br/>
    <input type="submit" />
  </form>

  <script type="text/javascript" src="js/jquery.js"></script>
  <script type="text/javascript" src="js/jquery.validate.min.js"></script>
  <script type="text/javascript" src="js/knockout-3.2.0.js"></script>
  <script>
    function ViewModel() {
      var self = this;

      self.person = {
        firstName: ko.observable(),
        lastName: ko.observable(),
        email: ko.observable()
      };

      self.validateAndSave = function(form) {
        if (!$.validate(form).valid()) return;

        $.ajax({
          url: 'myform.aspx',
          data: ko.toJSON(self.person),
          type: 'POST',
          contentType: 'application/x-www-form-urlencoded'
        }).success(self.successSave).error(self.errorSave);
      };

      self.successSave = function() {
        alert('Success!');
      };

      self.errorSave = function() {
        alert('Error!');
      };
    };

    var viewModel = new ViewModel();
  </script>

```

```

        ko.applyBindings(viewModel);
    </script>
</body>
</html>

```

This example is quite similar to the example shown in [Chapter 5](#). A few key changes have been made. First, the previous observable elements for the fields have been converted into a person object. This is done to make it easier to submit the entire object via AJAX. This can be seen inside of the `validateAndSave` function where the data being supplied via AJAX is using the Knockout function `ko.toJSON` to convert the person object into a usable JavaScript key/value pair.

If the AJAX call is successful, the `successSave` function will be executed, which will display a success message in an alert dialog. Similarly, if an error occurs, the `errorSave` function will be called, and an error message displayed.



Running the Example

The most likely result of running the above example will cause the error dialog to appear because the form is attempting to submit to a page called *myform.aspx*, which does not exist.

The only other significant changes to the previous example are the `form` tag data bindings. Previously, it was bound to a `validate` function; it is now bound to a `validateAndSave` function (a slightly more verbose name). It also leverages the `with` binding to the person object to avoid prefixing each input element to that object.

[Example 8-1](#) is submitting a standard form POST via AJAX as described earlier. To convert this example to submit the form via JSON, two minor changes need to be made to the AJAX call inside the `validateAndSave` function. See [Example 8-2](#).

Example 8-2. Form post via JSON

```

$.ajax({
    url: 'myform.aspx',
    data: ko.toJSON(self.person),
    type: 'POST',
    contentType: 'application/json'
})

```

Instead of using the `ko.toJSON` function, it is replaced with the very similar `ko.toJSON` function. As the function name indicates, this converts the person object into a JSON representation of it. The other minor change is that `contentType` is now set to `application/json`.



ko.toJS and ko.toJSON

The `ko.toJS` and `ko.toJSON` functions recursively go through the entire object and unwrap any properties or arrays from being an observed property to a regular JavaScript property.

Receiving Data

Receiving data is quite similar to sending data, but in reverse. When you're sending data to a server, you typically use observables to capture and send data. When receiving data, it's quite typical to use properties or observables to store the response from a server.

To demonstrate this, [Example 8-3](#) will convert the autocomplete example created in [Chapter 7](#). This time, the `availableTags` array will not be included directly within the `ViewModel`; the data will be loaded via the server.

Example 8-3. Server-side auto complete data

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <label for="tags">Filter a tag: </label>
  <input id="tags" data-bind="textInput: tag"><br/><br/>

  Tags Matching:
  <ul>
    <!-- ko foreach: matchedTags -->
    <li data-bind="text: $data"></li>
    <!-- /ko -->
  </ul>

  <script type='text/javascript' src='js/jquery.js'></script>
  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel() {
      var self = this;

      self.matchedTags = ko.observableArray([]);
      self.availableTags = [];
      self.tag = ko.observable().extend({
        rateLimit: {
          timeout: 1000,
          method: "notifyWhenChangesStop"
        }
      })
    }
  </script>
```



```

});

self.tag.subscribe(function(value) {
    self.matchedTags.removeAll();

    if (value !== '') {
        for (var i = 0; i < self.availableTags.length; i++) {
            if (self.availableTags[i].toLowerCase().indexOf(value) >= 0)
                self.matchedTags.push(self.availableTags[i]);
        }
    }
});

$.ajax({
    url: 'tags.html',
    type: 'GET',
    contentType: 'application/json',
    dataType: 'json'
}).success(function(data) {
    self.availableTags = data;
});

};

var viewModel = new ViewModel();
ko.applyBindings(viewModel);
</script>
</body>
</html>

```

This example is nearly identical to the previous example with the exception that the `availableTags` is populated with the response of the newly added AJAX call.

The AJAX request calls `tags.html` (shown in [Example 8-4](#)), which contains the same list of tags from the earlier example. Because the AJAX call was told the response would be JSON, the array of words can be stored directly in the `availableTags` array.

Example 8-4. Tags.html

```

[
    "ActionScript", "AppleScript", "Asp",
    "BASIC", "C", "C++", "Clojure",
    "COBOL", "ColdFusion", "Erlang",
    "Fortran", "Groovy", "Haskell",
    "Java", "JavaScript", "Lisp",
    "Perl", "PHP", "Python",
    "Ruby", "Scala", "Scheme"
]

```

If your server contains a language that has database access, this would be a perfect example to implement fetching the list of tags from a nonstatic file.



May Require a Web Server

The previous example might require a web server to execute. If you were previously running the examples as HTML files (e.g., the URL was `file://`), you might receive an error when performing the AJAX request.

For development purposes, I recommend setting up **XAMPP server**, which is compatible with Windows, Linux, and Mac OS X. Once set up, the sample files can be placed within the `htdocs` directory where XAMPP is installed.

In **Example 8-4**, the list of available tags is being loaded once, cached locally, and searched each time. Often, when performing this type of operation, it might be on a much larger data set where looping through an array is not practical.

An example like this could be altered quite easily to move the matching of tags to the server and taking the results and setting the `matchedTags` observable array from the response.

Example 8-5 will partially demonstrate this by pretending the response to the AJAX request contains only tags that match our query.

Example 8-5. Autocomplete that updates the matchedTags

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <label for="tags">Filter a tag: </label>
  <input id="tags" data-bind="textInput: tag"><br/><br/>

  Tags Matching:
  <ul>
    <!-- ko foreach: matchedTags -->
    <li data-bind="text: $data"></li>
    <!-- /ko -->
  </ul>

  <script type='text/javascript' src='js/jquery.js'></script>
  <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
  <script>
    function ViewModel() {
      var self = this;

      self.matchedTags = ko.observableArray([]);
      self.tag = ko.observable().extend({
```

```

        rateLimit: {
            timeout: 1000,
            method: "notifyWhenChangesStop"
        }
    });

    self.tag.subscribe(function(value) {
        self.matchedTags.removeAll();

        if (value !== '') {
            $.ajax({
                url: 'tags.html',
                type: 'GET',
                contentType: 'application/json',
                dataType: 'json'
            }).success(function(data) {
                self.matchedTags(data);
            });
        }
    });
};

var viewModel = new ViewModel();
ko.applyBindings(viewModel);
</script>
</body>
</html>

```

The previous example no longer contains the `availableTags` and the loop to perform the match; instead, whatever the results are from the AJAX request are directly assigned to the `matchedTags` observable array.

Using our imaginations for a minute, when you type in the tags box, if the *tags.html* page were performing the filtering logic, the `matchedTags` would be correctly filtered each time you typed based on the dynamic response from the server.

CHAPTER 9

The Mapping Plugin

Knockout allows developers to extend the framework by creating plugins, and the original developer has created one called *mapping*. The purpose of this plugin is to automatically map a standard JavaScript object or JSON data to a new object with each property being an observable object.

This can be extremely useful in several different situations. Where we had been creating objects and manually defining each property as observable, the mapping plugin will do this automatically.

To get started, [download the additional JavaScript file](#). I saved mine into the existing *js* folder and called it *knockout.mapping.js*, and that is how these next examples will reference the file.

Map from an Object

In [Example 9-1](#), I convert an example from [Chapter 3](#) that demonstrates the with binding on a single book object. In the original example, a book object is passed to the *ViewModel* via the constructor. This is then directly assigned to the internal book property and used for displaying with Knockout bindings. However, because it is using the mapping plugin, all of the properties (*title*, *synopsis*, and *published Date*) are now being observed.

Example 9-1. Mapping from an object

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
```

```

<body>

<div id="book" data-bind="with: book">
  <h1 data-bind="text: title"></h1>
  <h2>Published on <span data-bind="text: $parent.formatDate(publishedDate())">
</span></h2>
  <p data-bind="text: synopsis"></p>
</div>

<script type='text/javascript' src='js/knockout-3.2.0.js'></script>
<script type='text/javascript' src='js/knockout.mapping.js'></script>
<script>
  function ViewModel(book) {
    var self = this;

    self.book = ko.mapping.fromJS(book);

    self.formatDate = function(dateToFormat) {
      var months = new Array("January", "February", "March", "April",
        "May", "June", "July", "August", "September", "October",
        "November", "December");

      var d = new Date(dateToFormat);

      return months[d.getMonth()] + ' ' + d.getDate() + ', ' +
        d.getFullYear();
    };
  };

  var book = {
    title: 'Rapid Application Development With CakePHP',
    synopsis: '...',
    isbn: '1460954394',
    publishedDate: '2011-02-17'
  };

  var viewModel = new ViewModel(book);
  ko.applyBindings(viewModel);
</script>
</body>
</html>

```

In the original example, when displaying the published date via a function, the property was not being observed; in this example, it is, meaning when this function is called, the `publishedDate` requires parentheses to execute the function and return the observed value.

The other change is the `book` property is now being set with the `ko.mapping.fromJS` function, which converts each property into an observable.

A function like this can be extremely useful when setting up a form that is pre-populated from a large data object, which avoids the necessity of mapping each property manually.

Map from JSON (or a Server)

In the previous section, the data was being mapped from a JavaScript object. It's also quite common to want to map data from JSON via an AJAX request. [Example 9-2](#) will also extend an example from [Chapter 3](#). A list of books will be displayed in a table. The title of the book has been converted into a link. When the user selects the book, an AJAX request is made to fetch the book details. Using the mapping plugin, the response is mapped to an internal book property and used to dynamically display the book.

Example 9-2. Mapping from JSON

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <!-- ko if: hasFetchedBook -->
    <div id="book" data-bind="with: book">
      <img data-bind="attr: { src: image }" />
      <h1 data-bind="text: title"></h1>
      <h2>Published on
      <span data-bind="text: $parent.formatDate(publishedDate())">
      </span></h2>
      <p data-bind="text: synopsis"></p>
    </div>
  <!-- /ko -->

  <table>
    <thead>
      <tr>
        <th>Title</th>
        <th>ISBN</th>
        <th>Published</th>
      </tr>
    </thead>
    <tbody data-bind="foreach: books">
      <tr>
        <td><a href="#" data-bind="text: title, click: $parent.loadBook">
        </a></td>
        <td data-bind="text: isbn"></td>
        <td data-bind="text: $parent.formatDate(publishedDate)"></td>
      </tr>
```

```

</tbody>
</table>

<script type='text/javascript' src='js/jquery.js'></script>
<script type='text/javascript' src='js/knockout-3.2.0.js'></script>
<script type='text/javascript' src='js/knockout.mapping.js'></script>
<script>
    function ViewModel() {
        var self = this;

        self.books = [
            {
                title: 'Rapid Application Development With CakePHP',
                isbn: '1460954394',
                publishedDate: '2011-02-17'
            }, {
                title: '20 Recipes for Programming MVC 3:
                Faster, Smarter Web Development',
                isbn: '1449309860',
                publishedDate: '2011-10-14'
            }, {
                title: '20 Recipes for Programming PhoneGap:
                Cross-Platform Mobile Development for Android and iPhone',
                isbn: '1449319548',
                publishedDate: '2012-04-06'
            }
        ];

        self.book;
        self.hasFetchedBook = ko.observable(false);

        self.loadBook = function() {
            $.ajax({
                url: 'book.html',
                type: 'GET',
                contentType: 'application/json'
            }).success(function(data) {
                self.book = ko.mapping.fromJSON(data, self);
                self.hasFetchedBook(true);
            });
        };

        self.formatDate = function(dateToFormat) {
            var months = new Array("January", "February", "March", "April",
            "May", "June", "July", "August", "September", "October",
            "November", "December");

            var d = new Date(dateToFormat);

            return months[d.getMonth()] + ' ' + d.getDate() + ', ' +
            d.getFullYear();
        };
    };

```



```

    var viewModel = new ViewModel();
    ko.applyBindings(viewModel);
</script>
</body>
</html>

```

In this example, the title of the book within the table is an HTML link that is being data-bound to the `loadBook` function on click. This function performs an AJAX request to the `book.html` page (shown in [Example 9-3](#)). This contains similar data to the hardcoded book in the previous example. This function also sets the observable `hasFetchedBook` to `true` causing the HTML to dynamically render and show the book, image and all.

In the previous example, the book property was being mapped via a JavaScript object. In [Example 9-2](#), it's from JSON and thus uses the `ko.mapping.fromJSON` function instead.

Example 9-3. *book.html*

```

{
  "title": "Rapid Application Development With CakePHP",
  "synopsis": "...",
  "isbn": "1460954394",
  "publishedDate": "2011-02-17",
  "image": "http://ecx.images-amazon.com/images/I/41JC54HEroL._AA160_.jpg"
}

```

The above is the previous book object changed into valid JSON to be parsed by the mapping plugin.



The Same Book Every Time

Regardless of which book is selected, the same book will be returned every time in this example as there are no server-side logic capabilities to dynamically determine which book was selected. Of course, this example will not be difficult to extend, for example, to provide the book's ISBN number to the server.

Similar to the previous section, another great use of mapping from JSON would be to dynamically populate a form by fetching the data from the server, using the mapping plugin, and then binding the form to these elements.

Observing and Ignoring Specific Properties

When observables were first introduced, I hope I identified the importance of not unnecessarily making every property in an object observable. In [Example 9-1](#), the mapping plugin used the default behavior and converted everything into an observable. This is perfect when everything needs to be observed; however, there will be many times that this is not the case and the plugin provides options for scenarios like this.

In [Example 9-4](#), let's piece together several different things we've discussed but not yet fully implemented. This example demonstrates mapping data for a form, making only the necessary properties observed and using conditional functions to work as a form for adding and editing.

Example 9-4. Mapping plugin with options

```
<!DOCTYPE html>
<html>
<head>
  <title>Data Binding with KnockoutJS</title>
</head>
<body>

  <form data-bind="submit: validateAndSave, with: user">
    <input type="text" data-bind="value: firstName"
      placeholder="Enter your first name" required /><br/>
    <input type="text" data-bind="value: lastName"
      placeholder="Enter your last name" required /><br/>
    <input type="email" data-bind="value: email"
      placeholder="Enter your email" required /><br/>

    <input type="submit" data-bind="value: (id == 0) ? 'Create' : 'Update'" />
  </form>

  <script type="text/javascript" src="js/jquery.js"></script>
  <script type="text/javascript" src="js/jquery.validate.min.js"></script>
  <script type="text/javascript" src="js/knockout-3.2.0.js"></script>
  <script type="text/javascript" src="js/knockout.mapping.js"></script>
  <script>
    function ViewModel(user) {
      var self = this;

      var mapping = {
        'observe': [ "firstName", "lastName", "email" ],
        'ignore': [ "created" ]
      };

      self.user = ko.mapping.fromJS(user, mapping);
```

```

        self.validateAndSave = function(form) {
            if (!$(form).validate()) return;

            if (self.user.id == 0)
                alert("Creating a user");
            else
                alert("Updating a user");
        };

        var user = {
            id: 1,
            firstName: 'Mike',
            lastName: 'Wilson',
            email: 'mike@notarealaddress.com',
            created: '2014-10-12'
        };

        var viewModel = new ViewModel(user);
        ko.applyBindings(viewModel);
    </script>
</body>
</html>

```

This example accepts a `user` object in the constructor to the `ViewModel`. This object is then mapped to an internal `user` property that is bound to the form elements for editing.

The `user` object contains several additional properties that might be useful, but not editable within the form. To not make these objects observed unnecessarily, several mapping options are defined and passed into the mapping function.

First, an explicit list of properties that should be observed are defined in the `observe` option. These properties match the elements that are contained within the form. Second, the `ignore` option is used to not map the `created` property, as this is not required on the page.

CHAPTER 10

A Practical Example

For this final chapter, I thought it would be a good idea to see several of the things that have been demonstrated in small examples throughout the previous chapters. What better way to do that than with a shopping cart?

In previous examples, the HTML and JavaScript code have often been presented in a single example. Because this HTML and JavaScript is slightly larger than previous examples, it has been broken up into several small JavaScript files and one HTML file.

Building a Shopping Cart

Let's begin by defining the goals of the shopping cart:

- Include a category selection to allow easy filtering of products
- Display a list of products for a given category
- See how many items are currently in the shopping cart
- Add an item to the cart
- Remove an item from the cart
- Display the running total of items in the cart
- Include some simple animations when adding/removing from the cart

I think that covers the core features of a typical shopping cart. By looking at these features, I see an immediate need for three arrays:

1. List of categories with their available items.
2. List of available products for the active category.
3. List of items the user is interested in purchasing.

In addition to these three arrays, we want to keep track of the current running total.

Example 10-1 shows the HTML that covers the shopping cart's features.

Example 10-1. HTML for shopping cart

```

<!DOCTYPE html>
<html>
<head>
  <title>Yet Another Shopping Cart</title>
</head>
<body>

  <div style="float: left; width: 50%">

    <ul>
      <!-- ko foreach: categories -->
      <li><a href="#" data-bind=
        "click: $parent.showProducts, text: name"></a></li>
      <!-- /ko -->
    </ul>

    <ul>
      <!-- ko foreach: availableProducts -->
      <li>
        <!-- ko template: { name: 'item-template', data: $data } -->
        <!-- /ko -->
        <p><button data-bind="click: $parent.addItem">Add to cart</button></p>
      </li>
      <!-- /ko -->
    </ul>

  </div>
  <div style="float: left; width: 50%">

    <h1>Items in cart: <span data-bind="text: cartItems().length"></span></h1>

    <ul>
      <!-- ko foreach:
        { data: cartItems, afterAdd: fadeIn, beforeRemove: fadeOut } -->
      <li>
        <!-- ko template: { name: 'item-template', data: $data } -->
        <!-- /ko -->
        <p><button data-bind="click: $parent.removeItem">Remove</button>
        </p>
      </li>
      <!-- /ko -->
    </ul>

    <h3>Total: $<span data-bind="text: cartTotal"></span></h3>

  </div>

  <script type="text/html" id="item-template">
    <h2 data-bind="text: getTitle()"></h2>
    <p data-bind="html: getDescription()"></p>

```

```

        <p>Price: $<span data-bind="text: price"></span></p>
    </script>
    <script type='text/javascript' src='js/jquery.js'></script>
    <script type='text/javascript' src='js/jquery-ui.js'></script>
    <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
    <script type='text/javascript' src='js/classes.js'></script>
    <script type='text/javascript' src='js/data.js'></script>
    <script type='text/javascript' src='js/viewmodel.js'></script>
    <script>
        var viewModel = new ViewModel(categories);
        ko.applyBindings(viewModel);
    </script>
</body>
</html>

```

Using inline CSS, the output is split into two equal sides. On the left side, a list of categories is displayed in an unordered list. Beneath this list is a second unordered list that displays a list of available products after a category has been selected. By default, the `availableProducts` observable array is empty. This array is populated when a category is selected in the `showProducts` function that is data-bound to the click event.

On the right side of the screen, the cart items are displayed in another unordered list. You might notice that this `foreach` is different than the others because it is using the `afterAdd` and `beforeRemove` callback functions.

Because the output of the available products and selected cart items is the same, a template has been created to commonize the display of the item. The only difference among them is the button that is displayed below each item. When showing available products, the button allows the user to add an item and when showing the selected cart items, the button allows the user to remove the item.

Also on the right side, the total number of products added to the cart is shown, as well as the running cart total cost.

Finally, there are several JavaScript files that are included at the bottom of the HTML. A new library (**jQueryUI**) has been added to perform the simple animations when items are added or removed from the shopping cart. The other files (*classes.js*, *data.js*, *viewmodel.js*) are the divided files to better focus on each file in turn. See Examples 10-2, 10-3, and 10-4.

Example 10-2. *viewmodel.js*

```

function ViewModel(categories) {
    var self = this;

    self.categories = categories;

```

```

self.availableProducts = ko.observableArray([]);

self.cartItems = ko.observableArray([]);

self.cartTotal = ko.computed(function() {
    var total = 0;
    for (var i = 0; i < self.cartItems().length; i++)
        total += self.cartItems()[i].price;

    return total;
});

self.showProducts = function(category) {
    self.availableProducts(category.items);
};

self.addItem = function(item) {
    self.cartItems.push(item);
};

self.removeItem = function(item) {
    self.cartItems.remove(item);
};

self.fadeIn = function(element) {
    $(element).hide().fadeIn('slow');
};

self.fadeOut = function(element) {
    $(element).slideUp(function() {
        $(element).remove();
    });
};
};

```

The categories that are passed into the constructor will be shown in [Example 10-3](#) when the *data.js* file is displayed. The array is stored in the internal *categories* array. The *availableProducts* and *cartItems* are both defined as empty observable arrays that will be used to dynamically display the products for a given category and the items in the user's cart, respectively.

The *cartTotal* is a computed observable that loops through the items in the user's cart and adds the price to the running total. This function is executed each time the *cartItems* observable array is changed.

The *showProducts* function resets the *availableProducts* array with the list of items for the selected category.

The *addItem* and *removeItem* functions add and remove the selected item to and from the *cartItems* array, respectively.

And finally, the `fadeIn` and `fadeOut` functions leverage some basic jQueryUI animation functions to make the adding and removing of cart items a little fancy.

Example 10-3. data.js

```
var categories = [
  new Category(1, "Books", [
    new Book("1", "Book 1", "Book 1 Description", 1.99),
    new Book("2", "Book 2", "Book 2 Description", 2.99),
    new Book("3", "Book 3", "Book 3 Description", 3.99)
  ]),
  new Category(2, "Movies", [
    new Movie("Movie 1", "Movie 1 Description", 120, 4.99),
    new Movie("Movie 2", "Movie 2 Description", 160, 5.99),
    new Movie("Movie 3", "Movie 3 Description", 110, 6.99)
  ]),
  new Category(3, "Music", [
    new Album("Artist 1", "Album 1", 10, 4.99),
    new Album("Artist 2", "Album 2", 8, 7.99),
    new Album("Artist 3", "Album 3", 12, 6.99)
  ])
];
```

Three different categories are created: Books, Movies, and Music. Each category item and the category itself are created by creating new objects (Category, Book, Movie, and Album). These are shown with the *classes.js* file in [Example 10-4](#).

Example 10-4. classes.js

```
function Category(id, name, items) {
  this.id = id;
  this.name = name;
  this.items = items;
};

function Book(isbn, title, description, price) {
  var self = this;

  self.isbn = isbn;
  self.title = title;
  self.description = description;
  self.price = price;

  self.getTitle = function() {
    return self.title;
  };

  self.getDescription = function() {
    return self.description;
  };
};
```

```

};

function Movie(title, description, movie_length, price) {
  var self = this;

  self.title = title;
  self.description = description;
  self.movie_length = movie_length;
  self.price = price;

  self.getTitle = function() {
    return self.title;
  };

  self.getDescription = function() {
    return 'Summary: ' + self.description + '<br/>' +
      'Movie length: ' + self.movie_length;
  };
};

function Album(artist, name, number_of_songs, price) {
  var self = this;

  self.artist = artist;
  self.name = name;
  self.number_of_songs = number_of_songs;
  self.price = price;

  self.getTitle = function() {
    return self.name;
  };

  self.getDescription = function() {
    return 'Artist: ' + self.artist + '<br/>' +
      'Number of songs: ' + self.number_of_songs;
  };
};

```

The Category object accepts three parameters: id, name, and an array of items. These are the items that are used to populate the availableProducts observable array when a category is selected.

The remaining objects—Book, Movie, and Album—are slightly different in regards to what data they accept as parameters. This allows for the data to be structured differently depending on the type of product it is; however, each object has three shared properties: price, getTitle, and getDescription. Within each object, the getTitle and getDescription functions are used to determine how the output should be formatted when the item is displayed. This simplifies the original HTML to avoid conditional processing when displaying an item.

When this example is run, a list of three categories is displayed. When a category is selected, the list of products for that category will be displayed with a button to add that item to the cart. When the add button is clicked, a jQuery animation is used to fade in the product to the user's shopping cart. Both the item count and running total are updated to reflect the newly added item. When the item is added to the cart, there is a remove button. Clicking this button does the inverse of add; the item is removed with another jQuery animation, and the item count and cart total are updated again.

Although the example looks plain, the functionality is extremely slick-looking with Knockout automatically updating the user interface with each action the user takes.

I'll leave it to you to build the final form (data bindings and all) to collect the user's contact and credit card information to complete the checkout of their shopping cart!

CHAPTER 11

Next Steps

The Simplicity of Knockout

The last chapter signified the conclusion of the code examples in this book. While I was working on the plan for this book, I thought the last chapter would contain a significant amount of more code. A shopping cart is not a simple thing to build, as it includes many components such as browsing categories, products, and tracking items the user adds to his shopping cart. I really hope this demonstrates how KnockoutJS fits into your projects and makes maintaining a state across a single web page really easy. The final view model for the shopping cart was less than 40 lines of code (including line breaks for readability)! The HTML that performs the necessary data bindings was also less than 40 lines of code! The way Knockout brings this together with simple user interactions makes those 80 lines of code extremely powerful.

The focus of this book was about learning the ins and outs of Knockout. I think every feature was covered. If I were to advance the shopping cart example, I would focus on the design. The current implementation looks quite plain, but with a little bit of styling, it would quickly take on a whole new appearance with little to no changes to the JavaScript. Of course, design is not my strong suit, so I'll leave that to the designers!

Prior to using Knockout, I built heavy client-side user interfaces, and I can assure you that it involved significantly more than 80 lines of code. It involved complex jQuery to dynamically add and remove HTML elements, or adding and removing CSS classes. Compare that to Knockout, where it takes a few characters of data bindings to accomplish the same thing!

The User's Context

Both as a user of the Internet and a web developer, I'm constantly thinking of usability and user interfaces. One of the key focuses I pay close attention to is *the user's context*.

For example, it is quite common for additional data to be loaded via AJAX. I see two common approaches to this: loading the data and *adding* it below the existing content, or loading the data and *replacing* the existing content. The former perfectly maintains my context. In the latter example, when the content is replaced, it's extremely important that, as a user, my expected context is back to the start of the new content and not at the bottom of the new content where the action to load the content was.

Another common example is with a form. These are often submitted via AJAX (as shown in this book). When an error is displayed, it's important to think about where this message is shown. The user's probable context would be the bottom of the form where the submit button is. If an error were displayed on the top of the screen, it would probably go unnoticed by the user.

When building dynamic client-side applications, Knockout makes it very easy to keep any content on the screen up-to-date, especially involving user interaction. Though Knockout makes it easy, it's important that the context is maintained properly. Most often, no effort is required, but be sure to keep this in mind when altering large portions of content. This will cause the screen to change size, potentially losing the appropriate context.

Page Reloads for Content Updates

Full page reloads are a thing of the past! If content needs to be updated, Knockout gives you no excuses not to do it. You simply have to make that data observable and update it when it changes.

I had a great discussion about this the other day with a coworker. We had the option to:

- Reload the page and lose the user's context (and not be slick-looking)
- Perform a loop and perform an if/else statement within the loop to update the observed content

Not wanting to lose the user's context, the second option was chosen. Sure, it took 2 minutes to write the loop and conditional statement versus 15 seconds to write *location.reload*. However, the result of choosing option two is a much better user experience. Clicking the button causes the user interface to update, providing (almost) instantaneous feedback after the action occurs.

This also provides the ability for other neat user interface enhancements like animations. This was demonstrated within the shopping cart by fading in and sliding out cart items as they were added and removed, respectively. I find this to be a much nicer interface as opposed to being relocated to a new page to show the same information.

Conclusion

I hope this book has provided you with a lot of useful examples. As I thought about each feature, I attempted to think about how I use it on a daily basis in my projects. And where I could, I passed on the little intricacies that I've stumbled across while using Knockout inside a large project.

Index

Symbols

\$data variable, 11
 \$index variable, 12
 \$parent variable, 11, 15
 \$parents variable, 11, 15
 \$root variable, 11, 15

A

afterAdd method, 15
 afterMove method, 15
 afterRender method, 15
 AJAX, 55, 80
 alert statements, 22
 arrays
 in shopping cart example, 71
 observable arrays, 22
 selectedOptions binding and, 31

B

beforeMove method, 15
 beforeRemove method, 15
 brackets, 28

C

callback methods, 15-17
 CamelCase, 8
 checked binding, 31
 click binding, 33
 computed observables, 22
 conditional statements, 9, 28, 68
 CSS
 classes, binding of, 8
 display property, 24

display: none style, 26
 custom data bindings, 40
 custom functions, 47

D

data
 receiving, 58-61
 sending, 55-58
 data binding
 adding multiple, 8
 condition-based, 9, 28
 creating custom, 40
 CSS classes and styles, 8
 HTML attributes, 8
 HTML data, 7
 specifying with HTML comments, 1
 syntax for, 1
 two-way, 31
 data binding context
 foreach bindings, 12-15
 foreach callbacks (events), 15-17
 navigating between, 11
 parent/child hierarchy, 11
 root context, 11
 with bindings, 17-19
 data validation, 36
 data-bind attribute, 1, 7
 disable binding, 31
 display property, 24
 DOM (Document Object Model), 27, 33
 drop-down lists, 31, 34

E

elements

- adding/removing, 26
- form elements, 31-38
- showing/hiding, 24
- enable binding, 31
- event binding, 33
- events
 - foreach callbacks, 15-17
 - used with forms, 33

F

- foreach bindings, 12-15
- foreach callbacks (events), 15-17
- form elements, 33
- forms
 - adding to/editing, 68
 - bindings for form elements, 31
 - enabling/disabling text boxes, 31
 - event data bindings, 33
 - form validation, 36
 - listening for changes, 34
 - sending data from, 55-58
 - two-way bindings for, 31
 - working with, 31-38
- functions
 - adding custom to observables, 49
 - conditional, 68
 - creating custom, 47

H

- hasFocus binding, 33
- HTML
 - binding attributes, 8
 - binding HTML data, 7
 - data bindings via comments, 1
 - data-bind attribute, 1
 - displaying lists in HTML tables, 13
 - for shopping cart example, 71
 - repeating sections on arrays, 12
 - reusing templates, 43

I

- if/ifndef data bindings, 26
- inheritance, 49
- input element, 31, 33

J

- JavaScript, 4, 43, 63
- jQuery, 17, 55

- jQuery Validation Plugin, 36

K

- key/value pairs, 55
- keypress event, 33
- KnockoutJS
 - basic implementation
 - data binding example, 2
 - data binding syntax, 1
 - MVVM (Model-View-ViewModel), 3
 - overview of, 1
 - ViewModel creation, 3
 - ViewModel object orientation, 4
 - ViewModels with parameters, 5
 - benefits of, vii, 1
 - common uses of, 2
 - history of, vii
 - installing, vii
 - plugins for, 63
 - purpose of, 1
 - templates, 43
 - version changes, viii
- ko.applyBindings function, 3, 11, 14, 39
- ko.mapping.fromJS, 64
- ko.mapping.fromJSON function, 67
- ko.observable function, 21
- ko.toJS function, 57
- ko.toJSON function, 57

L

- listening mode, 23
- lists, 31

M

- mapping plugin
 - map from an object, 63
 - map from JSON, 65
 - purpose of, 63
- Model-View-Controller (MVC) pattern, 3
- Model-View-ViewModel (MVVM) pattern, 3
- mouseover event, 33
- Mozilla Developer Network (MDN), 4
- multiselect lists, 31

O

- object-oriented programming (OOP), 4
- observables
 - accessing, 22

- adding custom functions to, 49
- adding/removing elements, 26
- best use of, 28, 68
- combining into single object, 22
- computed observables, 22
- creating observable variables, 21
- defining, 21
- extending with custom functions, 47
- instantiating observable arrays, 22
- partial application of, 28
- pureComputed observables, 23
- purpose of, 21
- rate-limiting, 51-53
- selectedOptions binding and, 31
- showing/hiding elements, 24
- storing server responses with, 58
- subscribe function, 34
- updates with textInput vs. value bindings, 31
- use of brackets and, 28
- visible data binding, 24

onChange events, 32

options binding, 31

P

- parent/child hierarchy, 11
- parentheses, 43
- plugins, 63
 - (see also mapping plugin)
- POST request method, 55
- properties
 - dynamically changing, 21
 - (see also observables)
 - observing/ignoring specific, 68
 - storing server responses with, 58
- pureComputed observables, 23

R

- rate-limiting observables, 51-53
- REST APIs, 55

S

- screen flickers, preventing, 26
- select element, 31
- selectedOptions binding, 31
- self variable, 5
- server-side interaction
 - choices for, 55

- mapping from, 65
 - receiving data, 58-61
 - sending data, 55-58
- shopping cart example
 - addItem, 74
 - animated enhancements for, 81
 - arrays needed, 71
 - availableProducts, 74
 - benefits of KnockoutJS for, 79
 - cartItems, 74
 - cartTotal, 74
 - core features of, 71
 - current running total, 71
 - design improvements, 79
 - fadeIn/fadeOut functions, 75
 - HTML for, 71
 - output, 73
 - removeItem, 74
 - showProducts, 74
- showExtraData variable, 26, 28
- sleeping mode, 23
- style bindings, 8
- submit binding, 33
- submit data binding, 36
- subscribe function, 34

T

- templates, 43, 73
- text boxes, enabling/disabling, 31
- textarea element, 31, 40
- textInput binding, 31
- this variable, 5
- TinyMCE WYSIWYG editor, 40
- two-way bindings, 31

U

- unobtrusive form validation, 36
- updateObservable function, 25
- user interfaces (UIs)
 - animated enhancements for, 81
 - content updates, 80
 - dynamic, vii
 - role of KnockoutJS, 1
 - user's context, 80

V

- validate function, 37
- validation, of forms, 36

- value binding, 31
 - valueUpdate property, 32
 - var self = this, 5
 - variables
 - \$data variable, 11
 - \$index variable, 12
 - \$parent variable, 11, 15
 - \$parents variable, 11, 15
 - \$root variable, 11, 15
 - creating observable variables, 21
 - selectedOptions binding and, 31
 - showExtraData, 26
 - ViewModels
 - binding multiple, 39
 - creating, 3
 - displaying name property within, 2
 - dynamically displaying content within, 2
 - instantiating as a class, 14
 - object-oriented, 4
 - with parameters, 5
 - wrapping in a class, 5
 - visible data binding, 24, 26
- ## W
- with bindings, 17-19
 - WYSIWYG editors, 40
- ## X
- XAMPP server, 60

About the Author

The author of *20 Recipes for Programming MVC 3* and *20 Recipes for Programming PhoneGap* from O'Reilly Media, **Jamie Munro** has been developing websites and web applications for over 15 years. For the past 8 years, Jamie has been acting as a lead developer by mentoring younger developers to enhance their web development skills. Taking his love of mentoring people, Jamie began his writing career on his [personal blog](#) back in 2009. As Jamie's blog grew in success, he turned his writing passion to books about web development in hopes that his many years of experience could be passed on to you, the reader.

Colophon

The animal on the cover of *Knockout.js* is a black tree kangaroo (*Dendrolagus ursinus*), a marsupial that inhabits the tropical rainforests of New Guinea, far northeastern Queensland, and some of the islands in the region. The tree kangaroo is the only true arboreal member of the kangaroo family.

The black tree kangaroo is easily recognized by its long, tufted ears, which are unique among tree kangaroo species. Long, black fur covers the back and tail, while the fur on the underside is coffee-colored. Its distinctive face is more brown, with white or sometimes red cheeks and a white throat.

The black tree kangaroo is known to be mostly solitary and nocturnal, emerging after dark to feed on its preferred food of leaves and fruit.

Indigenous peoples throughout the tree kangaroo's range hunt the animals for food, sometimes using dogs to track them down. For several species, hunting alone has driven these marsupials close to extinction.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from *Riverside Natural History*. The cover fonts are URW Type-writer and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.