Knockout.js Succinctly

by Ryan Hodson

Syncfusion | Technology Resource Portal

Knockout.js Succinctly

By
Ryan Hodson

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Edited by

This publication was edited by Daniel Jebaraj, vice president, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Book

This book is intended for professional web developers who need to build dynamic, scalable user interfaces with minimal markup. Basic knowledge of HTML, CSS, and JavaScript is assumed. Experience with any particular JavaScript framework (e.g., jQuery, Prototype, MooTools, etc.) is not strictly required, though it wouldn't hurt.

The first two chapters provide a brief overview of the Knockout.js library. Chapter 3 discusses the data-oriented aspects of Knockout.js, and then Chapters 4 through 6 show you how to connect this data to HTML elements. The last two chapters of this book use jQuery's AJAX functionality to demonstrate how Knockout.js interacts with server-side applications and jQuery's animation features to add some flare to our data-driven interfaces. If you've never used jQuery before, don't worry—the examples are easily adapted to other frameworks.

Introduction

Creating data-driven user interfaces is one of the most complex jobs of a web developer. It requires careful management between the interface and its underlying data. For example, consider a simple shopping-cart interface for an e-commerce website. When the user deletes an item from the shopping cart, you have to remove the item from the underlying data set, remove the associated element from the shopping cart's HTML page, and update the total price. For all but the most trivial of applications, figuring out which HTML elements rely on a particular piece of data is an error-prone endeavor.

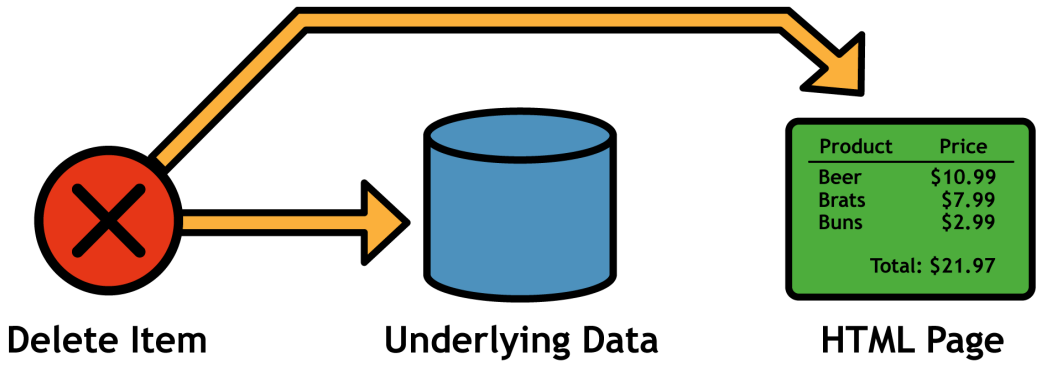

Figure 1: Manually tracking dependencies between HTML elements and their underlying data

The Knockout.js JavaScript library provides a cleaner way to manage these kinds of complex, data-driven interfaces. Instead of manually tracking which sections of the HTML page rely on the affected data, Knockout.js lets you create a direct connection between the underlying data and its presentation. After linking an HTML element with a particular data object, any changes to that object are *automatically* reflected in the DOM.

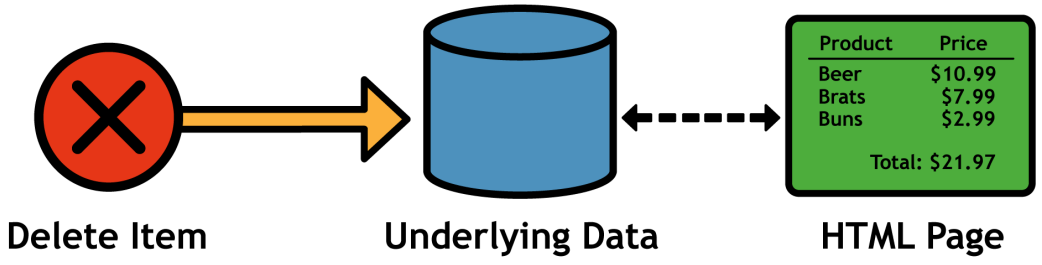

Figure 2: Automatically tracking dependencies using Knockout.js

This allows you to focus on the data behind your application. After you set up your HTML templates, you can work exclusively with JavaScript data objects. With Knockout.js, all you have to do to remove an item from the shopping cart is remove it from the JavaScript array that represents the user's shopping cart items. The corresponding HTML elements will automatically be removed from the page, and the total price recalculated.

Put another way, Knockout.js lets you design a self-updating display for your JavaScript objects.

Other Features

But, that's not all Knockout can do. In addition to automatic dependency tracking, it boasts several supporting features for the rapid development of responsive user interfaces…

Pure JavaScript

Knockout.js is a *client-side* library written entirely in JavaScript. This makes it compatible with virtually any server-side software, from ASP.NET to PHP, Django, Ruby on Rails, and even custom-built web frameworks.

When it comes to the front-end, Knockout.js connects the underlying data model to HTML elements by adding a single HTML attribute. This means it can be integrated into an existing project with minimal changes to your HTML, CSS, and other JavaScript libraries.

Extensible

While Knockout.js ships with almost two dozen bindings for defining how data is displayed, you may still find yourself in need of an application-specific behavior (e.g., a star-rating widget for user-submitted movie reviews). Fortunately, Knockout.js makes it easy to add your own bindings, giving you complete control over how your data is transformed into HTML. And, since these custom bindings are integrated into the core templating language, it's trivial to reuse widgets in other parts of your application.

Figure 3: Reusing a custom binding in several user interface components

Utility Functions

Knockout.js comes with several utility functions, including array filters, JSON parsing, and even a generic way to map data from the server to an HTML view. These utilities make it possible to turn large amounts of data into a dynamic user interface with just a few lines of code.

What Knockout.js is *Not*

Knockout.js is *not* meant to be a replacement for jQuery, Prototype, or MooTools. It doesn't attempt to provide animation, generic event handling, or AJAX functionality (however, Knockout.js can *parse* the data received from an AJAX call). Knockout.js is focused solely on designing scalable, data-driven user interfaces—how that underlying data is obtained is completely up to you.

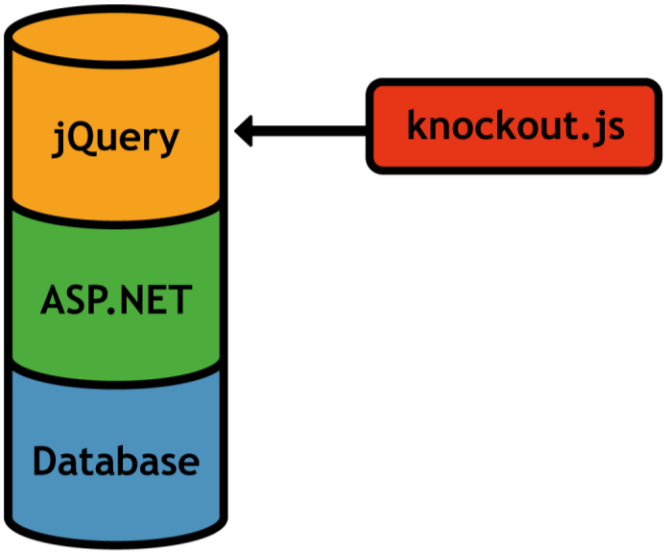

Figure 4: Knockout.js supplementing a full web application stack

This high level of specialization makes Knockout.js compatible with any other client-side and server-side technology, but it also means Knockout.js often requires the cooperation of a more full-featured JavaScript framework. In this sense, Knockout.js is more of a *supplement* to a traditional web application stack, rather than an integral part of it.

Chapter 1 Conceptual Overview

Knockout.js uses a Model-View-ViewModel (MVVM) design pattern, which is a variant of the classic Model-View-Controller (MVC) pattern. As in the MVC pattern, the **model** is your stored data, and the **view** is the visual representation of that data. But, instead of a controller, Knockout.js uses a **ViewModel** as the intermediary between the model and the view.

The ViewModel is a JavaScript representation of the model data, along with associated functions for manipulating the data. Knockout.js creates a direct connection between the ViewModel and the view, which is how it can detect changes to the underlying data and automatically update the relevant aspects of the user interface.

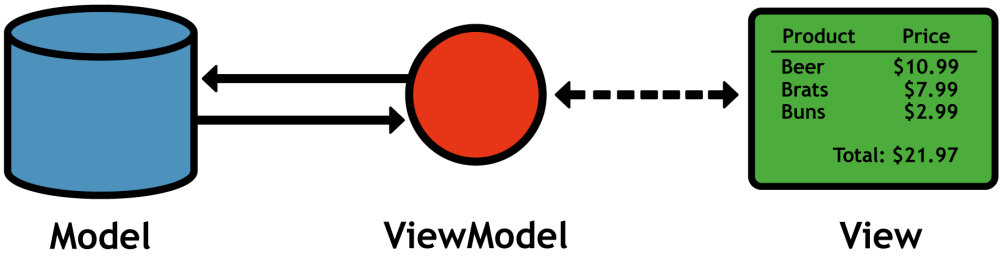

Figure 5: The Model-View-ViewModel design pattern

The MVVM components of our shopping cart example are listed as follows:

- **Model:** The contents of a user's shopping cart stored in a database, cookie, or some other persistent storage. Knockout.js doesn't care how your data is stored—it's up to you to communicate between your model storage and Knockout.js. Typically, you'll save and load your model data via an AJAX call.
- **View:** The HTML/CSS shopping cart page displayed to the user. After connecting the view to the ViewModel, it will automatically display new, deleted, and updated items when the ViewModel changes.
- **ViewModel:** A pure-JavaScript object representing the shopping cart, including a list of items and save/load methods for interacting with the model. After connecting your HTML view with the ViewModel, your application only needs to worry about manipulating this object (Knockout.js will take care of the view).

Observables

Knockout.js uses **observables** to track a ViewModel's properties. Conceptually, observables act just like normal JavaScript variables, but they let Knockout.js *observe* their changes and automatically update the relevant parts of the view.

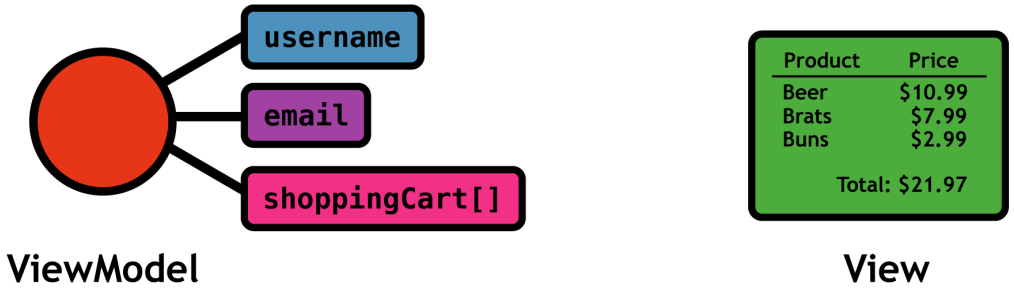

Figure 6: Using observables to expose ViewModel properties

Bindings

Observables only *expose* a ViewModel's properties. To connect a user interface component in the view to a particular observable, you have to **bind** an HTML element to it. After binding an element to an observable, Knockout.js is ready to display changes to the ViewModel automatically.

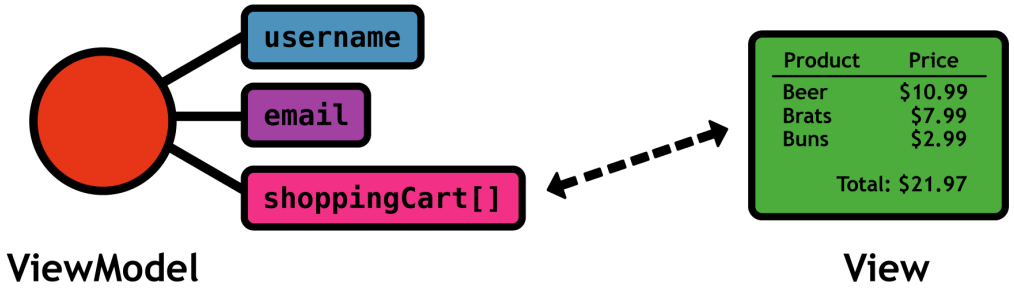

Figure 7: Binding a user interface component to an observable property

Knockout.js includes several built-in bindings that determine how the observable appears in the user interface. The most common type of binding is to simply display the value of the observed property, but it's also possible to change its appearance under certain conditions, or to call a method of the ViewModel when the user clicks the element. All of these use cases will be covered over the next few chapters.

Summary

The Model-View-ViewModel design pattern, observables, and bindings provide the foundation for the Knockout.js library. Once you understand these concepts, learning Knockout.js is simply a matter of figuring out how to access observables and manipulate them via the various built-in bindings. In the next chapter, we'll take our first concrete look at these concepts by building a simple "Hello, World!" application.

Chapter 2 Hello, Knockout.js

This chapter is designed to be a high-level survey of Knockout.js' main components. By implementing a concrete sample application, we'll see how Knockout's ViewModel, view, observables, and bindings interact to create a dynamic user interface.

First, we'll create a simple HTML page to hold all of our code, then we'll define a ViewModel object, expose some properties, and even add an interactive binding so that we can react to user clicks.

Download Knockout.js

Before we start writing any code, download the latest copy of Knockout.js from the downloads page at GitHub.com. As of this writing, the most recent version is 2.1.0. After that, we're ready to add the library to an HTML page.

Samples

The samples in this book are available at https://bitbucket.org/syncfusion/knockoutjs_succinctly.

The HTML

Let's start with a standard HTML page. In the same folder as your Knockout.js library, create a new file called `index.html`, and add the following. Make sure to change `knockout-2.1.0.js` to the file name of the Knockout.js library you downloaded.

Sample code: item1.htm

```
<html lang='en'>
<head>
  <title>Hello, Knockout.js</title>
  <meta charset='utf-8' />
  <link rel='stylesheet' href='style.css' />
</head>
<body>
  <h1>Hello, Knockout.js</h1>
  <p>Bill's Shopping Cart</p>

  <script type='text/javascript' src='knockout-2.1.0.js'></script>
</body>
</html>
```

This is a basic HTML 5 webpage that includes the Knockout.js library at the bottom of `<body>`; although, like any external script, you can include it anywhere you want (inside `<head>` is the other common option). The `style.css` style sheet isn't actually necessary

for any of the examples in this book, but it will make them much easier on the eyes. It can be found in Appendix A, or downloaded from https://bitbucket.org/syncfusion/knockoutjs_succinctly. If you open the page in a web browser, you should see the following:

Figure 8: Basic webpage

Defining the ViewModel

Since we're not working with any persistent data yet, we don't have a model to work with. Instead we'll skip right to the ViewModel. Until Chapter 7, we're really just using a View-ViewModel pattern.

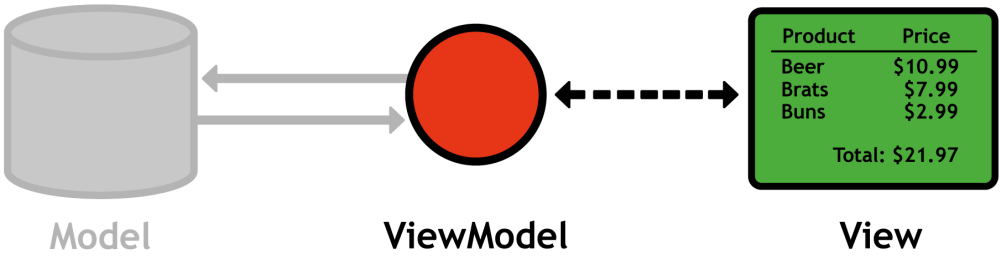

Figure 9: Focusing on the view and ViewModel for the time being

Remember, a ViewModel is a pure JavaScript representation of your model data. To start out, we'll just use a native JavaScript object as our ViewModel. Underneath the `<script>` tag that includes Knockout.js, add the following:

Sample code: item2.htm

```
<script type='text/javascript'>
  var personViewModel = {
    firstName: "John",
    lastName: "Smith"
  };
  ko.applyBindings(personViewModel);
</script>
```

```
</body>
```

This creates a “person” named John Smith, and the `ko.applyBindings()` method tells Knockout.js to use the object as the ViewModel for the page.

Of course, if you reload the page, it will still display “Bill’s Shopping Cart.” For Knockout.js to update the view based on the ViewModel, we need to **bind** an HTML element to the `personViewModel` object.

Binding an HTML Element

Knockout.js uses a special `data-bind` attribute to bind HTML elements to the ViewModel. Replace `Bill` in the `<p>` tag with an empty `<span>` element, as follows:

Sample code: item2.htm

```
<p><span data-bind='text: firstName'></span>'s Shopping Cart</p>
```

The value of the `data-bind` attribute tells Knockout.js what to display in the element. In this case, the `text` binding tells Knockout.js to display the `firstName` property of the ViewModel. Now, when you reload the page, Knockout.js will replace the contents of the `<span>` with `personViewModel.firstName`. As a result, you should see “John’s Shopping Cart” in your browser:

Figure 10: Screenshot of our first bound view component

Similarly, if you change the `data-bind` attribute to `text: lastName`, it will display “Smith’s Shopping Cart.” As you can see, binding an element is really just defining an HTML template for your ViewModel.

Observable Properties

So, we have a ViewModel that can be displayed in an HTML element, but watch what happens when we try to change the property. After calling `ko.applyBindings()`, assign a new value to `personViewModel.firstName`:

```
    ko.applyBindings(personViewModel);
    personViewModel.firstName = "Ryan";
```

Knockout.js *won't* automatically update the view, and the page will still read "John's Shopping Cart." This is because we haven't exposed the `firstName` property to Knockout.js. Any properties that you want Knockout.js to track must be **observable**. We can make our ViewModel's properties observable by changing `personViewModel` to the following:

Sample code: item3.htm

```
var personViewModel = {
  firstName: ko.observable("John"),
  lastName: ko.observable("Smith")
};
```

Instead of directly assigning values to `firstName` and `lastName`, we use `ko.observable()` to add the properties to Knockout.js' automatic dependency tracker. When we change the `firstName` property, Knockout.js should update the HTML element to match:

```
ko.applyBindings(personViewModel);
personViewModel.firstName("Ryan");
```

Accessing Observables

You've probably noticed that observables are actually functions—not variables. To get the value of an observable, you call it without any arguments, and to set the value, you pass the value as an argument. This behavior is summarized as follows:

- **Getting:** Use `obj.firstName()` instead of `obj.firstName`
- **Setting:** Use `obj.firstName("Mary")` instead of `obj.firstName = "Mary"`

Adapting to these new accessors can be somewhat counterintuitive for beginners to Knockout.js. Be very careful not to accidentally assign a value to an observable property with the = operator. This will overwrite the observable, causing Knockout.js to stop automatically updating the view.

Using Custom Objects

Our generic `personViewModel` object and its observable properties work just fine for this simple example, but remember that ViewModels can also define methods for interacting with their data. For this reason, ViewModels are often defined as custom classes instead of generic JavaScript objects. Let's go ahead and replace `personViewModel` with a user-defined object:

Sample code: item4.htm

```
function PersonViewModel() {
  this.firstName = ko.observable("John");
```

```
    this.lastName = ko.observable("Smith");
};
ko.applyBindings(new PersonViewModel());
```

This is the canonical way to define a ViewModel and activate Knockout.js. Now, we can add a custom method, like so:

```
function PersonViewModel() {
    this.firstName = ko.observable("John");
    this.lastName = ko.observable("Smith");
    this.checkout = function() {
        alert("Trying to check out!");
    };
};
```

Combining data and methods in a single object is one of the defining features of the MVVM pattern. It provides an intuitive way to interact with data. For example, when you're ready to check out simply call the `checkout()` method on the ViewModel. Knockout.js even provides bindings to do this directly from the view.

Interactive Bindings

Our last step in this chapter will be to add a checkout button to call the `checkout()` method we just defined. This is a very brief introduction to Knockout.js's interactive bindings, but it provides some useful functionality that we'll need in the next chapter. Underneath the `<p>` tag, add the following button:

```
<button data-bind='click: checkout'>Checkout</button>
```

Instead of a `text` binding that displays the value of a property, the `click` binding calls a method when the user clicks the element. In our case, it calls the `checkout()` method of our ViewModel, and you should see an alert message pop up.

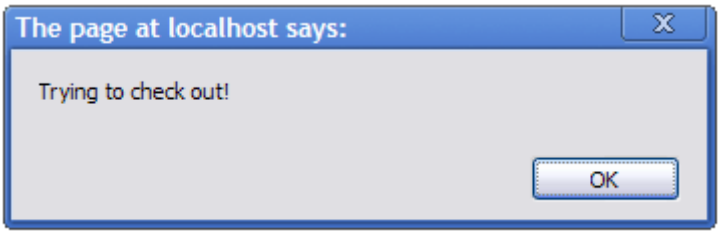

Figure 11: Alert message created after clicking the Checkout button

Knockout.js' full suite of interactive bindings will be covered in Chapter 6.

Summary

This chapter walked through the core aspects of Knockout.js. As we've seen, there are three steps to setting up a Knockout.js-based web application:

1. Creating a ViewModel object and registering it with Knockout.js.
2. Binding an HTML element to one of the ViewModel's properties.
3. Using observables to expose properties to Knockout.js

You can think of binding view elements to observable properties as building an HTML template for a JavaScript object. After the template is set up, you can completely forget about the HTML and focus solely on the ViewModel data behind the application. This is the whole point of Knockout.js.

In the next chapter, we'll explore the real power behind Knockout.js' automatic dependency tracker by creating observables that rely on other properties, as well as observable arrays to hold lists of data.

Chapter 3 Observables

We've seen how observable properties let Knockout.js automatically update HTML elements when underlying data changes, but this is only the beginning of their utility. Knockout.js also comes with two more ways of exposing ViewModel properties: computed observables and observable arrays. Together, these open up a whole new world of possibilities for data-driven user interfaces.

Computed observables let you create properties that are dynamically generated. This means you can combine several normal observables into a single property, and Knockout.js will still keep the view up-to-date whenever any of the underlying values change.

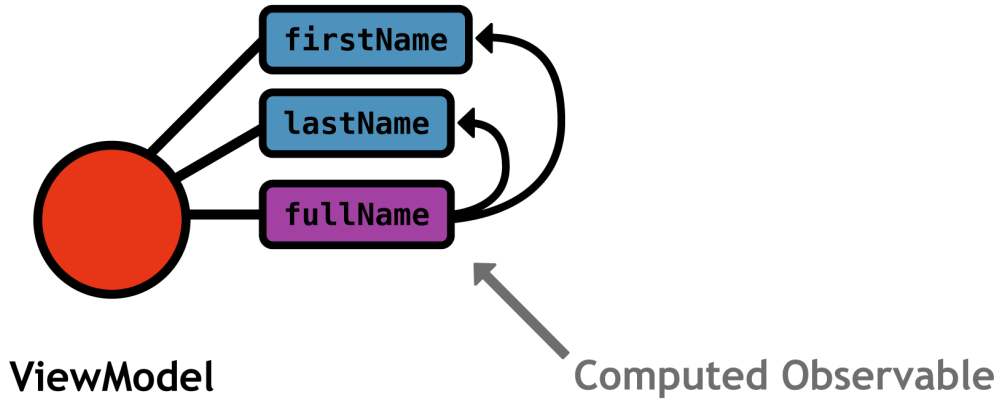

Figure 12: A computed observable dependent on two normal observables

Observable arrays combine the power of Knockout.js' observables with native JavaScript arrays. Like native arrays, they contain lists of items that you can manipulate. But since they're observable, Knockout.js automatically updates any associated HTML elements whenever items are added or removed.

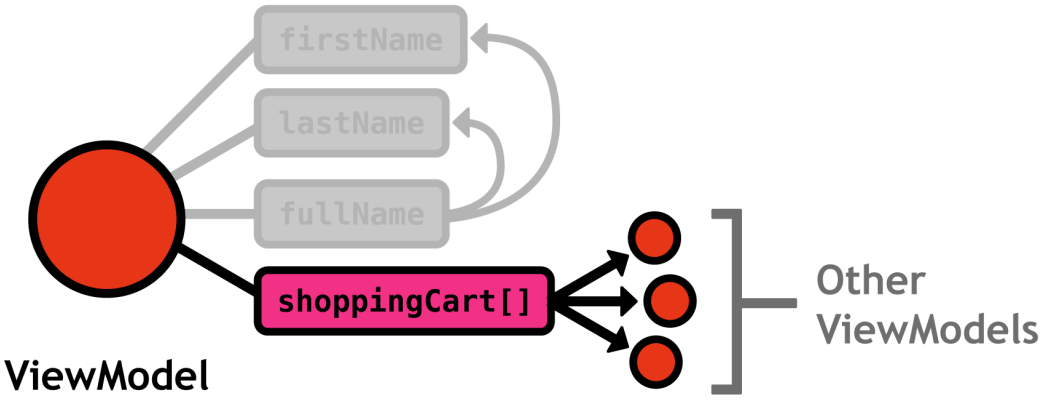

Figure 13: An observable array containing other ViewModels

The ability to combine observables, along with the ability to work with lists of items, provides all the data structures you'll need in a ViewModel. This chapter introduces both topics with a simple shopping cart interface.

Computed Observables

First, we'll start with a simple computed observable. Underneath the `firstName` and `lastName` observables in `PersonViewModel`, create the `fullName` computed observable:

Sample code: item5.htm

```
    this.fullName = ko.computed(function() {
      return this.firstName() + " " + this.lastName();
    }, this);
```

This defines an anonymous function that returns the person's full name whenever `PersonViewModel.fullName` is accessed. Dynamically generating the full name from the existing components (`firstName` and `lastName`) prevents us from storing redundant data, but that's only half the battle. We need to pass this function to `ko.computed()` to create a computed observable. This tells Knockout.js that it needs to update any HTML elements bound to the `fullName` property whenever either `firstName` or `lastName` change.

Let's make sure our computed observable works by binding the "John's Shopping Cart" line to `fullName` instead of `firstName`:

```
<p><span data-bind='text: fullName'></span>'s Shopping Cart</p>
```

Now your page should read "John Smith's Shopping Cart." Next, let's make sure that Knockout.js keeps this HTML element in sync when we change one of the underlying properties. After binding an instance of `PersonViewModel`, try changing its `firstName` property:

```
    var vm = new PersonViewModel();
    ko.applyBindings(vm);
    vm.firstName("Mary");
```

This should change the line to "Mary Smith's Shopping Cart." Again, remember that reading or setting observables should be done with function calls, not the assignment (=) operator.

Computed observables provide many of the same benefits as Knockout.js' automatic synchronization of the view. Instead of having to keep track of which properties rely on

other parts of the ViewModel, computed observables let you build your application around atomic properties and delegate dependency tracking to Knockout.js.

Observable Arrays

Observable arrays let Knockout.js track lists of items. We'll explore this by creating a shopping cart display page for our user. First, we need to create a custom object for representing products. At the top of our script, before defining `PersonViewModel`, add the following object definition:

Sample code: item6.htm

```
function Product(name, price) {
  this.name = ko.observable(name);
  this.price = ko.observable(price);
}
```

This is just a simple data object to store a few properties. Note that it's possible to give multiple objects observable properties, and Knockout.js will manage all of the interdependencies on its own. In other words, it's possible to create relationships between *multiple* ViewModels in a single application.

Next, we're going to create a few instances of our new `Product` class and add them to the user's virtual shopping cart. Inside of `PersonViewModel`, define a new observable property called `shoppingCart`:

```
this.shoppingCart = ko.observableArray([
  new Product("Beer", 10.99),
  new Product("Brats", 7.99),
  new Product("Buns", 1.49)
]);
```

This is a native JavaScript array containing three products wrapped in an observable array so Knockout.js can track when items are added and removed. But, before we start manipulating the objects, let's update our view so we can see the contents of the `shoppingCart` property. Underneath the `<p>` tag, add the following:

Sample code: item6.htm

```
<table>
  <thead><tr>
    <th>Product</th>
    <th>Price</th>
  </tr></thead>
  <tbody data-bind='foreach: shoppingCart'>
    <tr>
```

```
        <td data-bind='text: name'></td>
        <td data-bind='text: price'></td>
      </tr>
    </tbody>
  </table>
```

This is a typical HTML 5 table containing a column for product names and another for product prices. This example also introduces a new binding called `foreach`. When Knockout.js encounters `foreach: shoppingCart`, it loops through each item in the ViewModel's `shoppingCart` property. Any markup inside of the loop is evaluated in the context of each item, so `text: name` actually refers to `shoppingCart[i].name`. The result is a table of items alongside their prices:

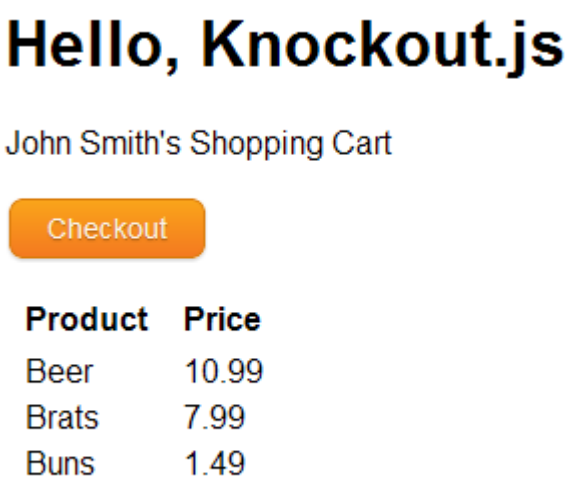

Figure 14: Screenshot of the rendered product listing

The details of the `foreach` binding are outside the scope of this chapter. The next chapter provides an in-depth discussion of `foreach`, and it also introduces Knockout.js' other control-flow bindings. For now, let's get back to observable arrays.

Adding Items

The whole point of using observable arrays is to let Knockout.js synchronize the view whenever we add or remove items. For example, we can define a method on our ViewModel that adds a new item, like so:

Sample code: item7.htm

```
    this.addProduct = function() {
       this.shoppingCart.push(new Product("More Beer", 10.99));
    };
```

Then, we can create a button to call the method so we can add items at run time and see Knockout.js keep the list up-to-date. Next to the checkout button in the view code, add the following:

```
  <button data-bind='click: addProduct'>Add Beer</button>
```

When you click this button, the ViewModel's `addProduct()` method is executed. And, since `shoppingCart` is an observable array, Knockout.js inserts another `<tr>` element to display the new item. Letting Knockout.js keep track of list items like this is much less error-prone than trying to manually update the `<table>` whenever we change the underlying array.

It's also worth pointing out that Knockout.js always makes the *minimal* amount of changes necessary to synchronize the user interface. Instead of regenerating the entire list every time an item is added or removed, Knockout.js tracks which parts of the DOM are affected and updates *only* those elements. This built-in optimization makes it possible to scale up your application to hundreds or even thousands of items without sacrificing responsiveness.

Deleting Items

Similarly, Knockout.js can also delete items from an observable array via the `remove()` method. Inside of the `PersonViewModel` definition, add another method for removing items:

Sample code: item8.htm

```
      this.removeProduct = function(product) {
        this.shoppingCart.remove(product);
      };
```

Then, add a delete button for each item in the `<tbody>` loop:

```
        <tr>
          <td data-bind='text: name'></td>
          <td data-bind='text: price'></td>
          <td><button data-bind='click:
$root.removeProduct'>Remove</button></td>
        </tr>
```

Because we're in the `foreach` context, we had to use the `$root` reference to access our ViewModel instead of the current item in the loop. If we tried to call `removeProduct()` without this reference, Knockout.js would have attempted to call the method on the `Product` class, which doesn't exist. All of the available binding contexts for `foreach` are covered in the next chapter.

The fact that we're in a `foreach` loop also messes up the `this` reference in `removeProduct()`, so clicking a **Remove** button will actually throw a `TypeError`. We can use a common JavaScript trick to resolve these kinds of scope issues. At the top of the `PersonViewModel` definition, assign `this` to a new variable called `self`:

```
function PersonViewModel() {
  var self = this;
  ...
```

Then, use `self` instead of `this` in the `removeProduct()` method:

```
this.removeProduct = function(product) {
  self.shoppingCart.remove(product);
};
```

You should now be able to manipulate our observable array with the **Add Beer** and **Remove** buttons. Also note that Knockout.js automatically adds the current item in the loop as the first parameter to `removeProduct()`.

Destroying Items

The `remove()` method is useful for real-time manipulation of lists, but it can prove troublesome once you start trying to send data from the ViewModel to a server-side script.

For example, consider the task of saving the shopping cart to a database every time the user added or deleted an item. With `remove()`, the item is removed *immediately*, so all you can do is send your server the new list in its entirety—it's impossible to determine which items where added or removed. You either have to save the entire list, or manually figure out the difference between the previous version stored in the database and the new one passed in from the AJAX request.

Neither of these options is particularly efficient, especially considering Knockout.js knows precisely which items were removed. To remedy this situation, observable arrays include a `destroy()` method. Try changing `PersonViewModel.removeProduct()` to the following:

Sample code: item9.htm

```
this.removeProduct = function(product) {
  self.shoppingCart.destroy(product);
  alert(self.shoppingCart().length);
};
```

Now when you click the **Remove** button, Knockout.js *won't* remove the item from the underlying array. This is shown in the alert message, which should *not* decrease when

you click “Remove.” Instead of altering the list, the `destroy()` method adds a `_destroy` property to the product and sets it to `true`. You can display this property by adding another alert message:

```
        alert(product._destroy);
```

The `_destroy` property makes it possible to sort through an observable list and pull out only items that have been deleted. Then, you can send *only* those items to a server-side script to be deleted. This is a much more efficient way to manage lists when working with AJAX requests.

Note that the `foreach` loop is aware of this convention, and still removes the associated `<tr>` element from the view, even though the item remains in the underlying array.

Other Array Methods

Internally, observable arrays are just like normal observable properties, except they are backed by a native JavaScript array instead of a string, number, or object. Like normal observables, you can access the underlying value by calling the observable array without any properties:

Sample code: item10.htm

```
      this.debugItems = function() {
        var message = "";
        var nativeArray = this.shoppingCart();
        for (var i=0; i<nativeArray.length; i++) {
          message += nativeArray[i].name + "\n";
        }
        alert(message);
      };
```

Calling this method will loop through the native list’s items, and it also provides access to the native JavaScript array methods like `push()`, `pop()`, `shift()`, `sort()`, etc.

However, Knockout.js defines *its own* versions of these methods on the observable array object. For example, earlier in this chapter, we used `shoppingCart.push()` to add an item instead of `shoppingCart().push()`. The former calls Knockout.js’ version, and the latter calls `push()` on the native JavaScript array.

It’s usually a much better idea to use Knockout.js’ array methods instead of accessing the underlying array directly because it allows Knockout.js to automatically update any dependent view components. The complete list of observable array methods provided by Knockout.js follows. Most of these act exactly like their native JavaScript counterparts.

- `push()`
- `pop()`

- `unshift()`
- `shift()`
- `slice()`
- `remove()`
- `removeAll()`
- `destroy()`
- `destroyAll()`
- `sort()`
- `reversed()`
- `indexOf()`

Summary

In this chapter, we saw how computed observables can be used to combine normal observables into compound properties that Knockout.js can track. We also worked with observable arrays, which are a way for Knockout.js to synchronize lists of data in the ViewModel with HTML components.

Together, atomic, computed, and array observables provide all the underlying data types you'll ever need for a typical user interface. Computed observables and observable arrays make Knockout.js a great option for rapid prototyping. They let you put all of your complex functionality one place, and then let Knockout.js take care of the rest.

For example, it would be trivial to create a computed observable that calculates the total price of each item in the `shoppingCart` list and displays it at the bottom of the page. Once you create that functionality, you can reuse it *anywhere* you need the total price (e.g., an AJAX request) just by accessing a ViewModel property.

The next chapter introduces control-flow bindings. The `foreach` binding that we used in this chapter is probably the most common control-flow tool, but Knockout.js also includes a few more bindings for fine-grained control over our HTML view components.

Chapter 4 Control-Flow Bindings

As we've seen in previous chapters, designing a view for a ViewModel is like creating an HTML template for a JavaScript object. An integral part of any templating system is the ability to control the flow of template execution. The ability to loop through lists of data and include or exclude visual elements based on certain conditions makes it possible to minimize markup and gives you complete control over how your data is displayed.

We've already seen how the `foreach` binding can loop through an observable array, but Knockout.js also includes two logical bindings: `if` and `ifnot`. In addition, its `with` binding lets you manually alter the scope of template blocks.

This chapter introduces Knockout.js' control-flow bindings by extending the shopping cart example from the previous chapter. We'll also explore some of the nuances of `foreach` that were glossed over in the last chapter.

The foreach Binding

Let's start by taking a closer look at our existing `foreach` loop:

Sample code: item010.htm

```
    <tbody data-bind='foreach: shoppingCart'>
      <tr>
        <td data-bind='text: name'></td>
        <td data-bind='text: price'></td>
        <td><button data-bind='click:
$root.removeProduct'>Remove</button></td>
      </tr>
    </tbody>
```

When Knockout.js encounters `foreach` in the `data-bind` attribute, it iterates through the `shoppingCart` array and uses each item it finds for the **binding context** of the contained markup. This binding context is how Knockout.js manages the scope of loops. In this case, it's why we can use the `name` and `price` properties without referring to an instance of `Product`.

Working with Binding Contexts

Using each item in an array as the new binding context is a convenient way to create loops, but this behavior also makes it impossible to refer to objects outside of the current item in the iteration. For this reason, Knockout.js makes several special properties available in each binding context. Note that all of these properties are only available in the *view*, not the ViewModel.

The $root Property

The `$root` context always refers to the top-level ViewModel, regardless of loops or other changes in scope. As we saw in the previous chapter, this makes it possible to access top-level methods for manipulating the ViewModel.

The $data Property

The `$data` property in a binding context refers to the ViewModel object for the current context. It's a lot like the `this` keyword in a JavaScript object. For example, inside of our `foreach: shoppingCart` loop, `$data` refers to the current list item. As a result, the following code works exactly as it would without using `$data`:

```
        <td data-bind='text: $data.name'></td>
        <td data-bind='text: $data.price'></td>
```

This might seem like a trivial property, but it's indispensable when you're iterating through arrays that contain atomic values like strings or numbers. For example, we can store a list of strings representing tags for each product:

Sample code: item011.htm

```
function Product(name, price, tags) {
  this.name = ko.observable(name);
  this.price = ko.observable(price);
  tags = typeof(tags) !== 'undefined' ? tags : [];
  this.tags = ko.observableArray(tags);
}
```

Then, define some tags for one of the products in the `shoppingCart` array:

```
        new Product("Buns", 1.49, ['Baked goods', 'Hot dogs']);
```

Now, we can see the `$data` context in action. In the `<table>` containing our shopping cart items, add a `<td>` element containing a `<ul>` list iterating through the `tags` array:

```
<tbody data-bind='foreach: shoppingCart'>
  <tr>
    <td data-bind='text: name'></td>
    <td data-bind='text: price'></td>
    <td> <!-- Add a list of tags. -->
      <ul data-bind='foreach: tags'>
        <li data-bind='text: $data'></li>
      </ul>
    </td>
```

```
        <td><button data-bind='click:
$root.removeProduct'>Remove</button></td>
      </tr>
    </tbody>
  </table>
```

Inside of the `foreach: tags` loop, Knockout.js uses the native strings “Baked goods” and “Hot dogs” as the binding context. But, since we want to access the actual strings instead of their *properties*, we need the `$data` object.

The $index Property

Inside of a `foreach` loop, the `$index` property contains the current item’s index in the array. Like most things in Knockout.js, the value of `$index` will update automatically whenever you add or delete an item from the associated observable array. This is a useful property if you need to display the index of each item, like so:

Sample code: item012.htm

```
        <td data-bind='text: $index'></td>
```

The $parent Property

The `$parent` property refers to the parent ViewModel object. Typically, you’ll only need this when you’re working with nested loops and you need to access properties in the outer loop. For example, if you need to access the `Product` instance from the inside of the `foreach: tags` loop, you could use the `$parent` property:

Sample code: item013.htm

```
            <ul data-bind="foreach: tags">
                <li>
                    <span data-bind="text: $parent.name"></span> -
<span data-bind="text: $data"></span>
                </li>

            </ul>
```

Between observable arrays, the `foreach` binding, and the binding context properties discussed previously, you should have all the tools you need to leverage arrays in your Knockout.js web applications.

Discounted Products

Before we move on to the conditional bindings, we’re going to add a `discount` property to our `Product` class:

Sample code: item014.htm

```
    function Product(name, price, tags, discount) {
      ...
      discount = typeof(discount) !== 'undefined' ? discount : 0;
      this.discount = ko.observable(discount);
      this.formattedDiscount = ko.computed(function() {
        return (this.discount() * 100) + "%";
      }, this);
    }
```

This gives us a condition we can check with Knockout.js' logical bindings. First, we make the `discount` parameter optional, giving it a default value of `0`. Then, we create an observable for the discount so Knockout.js can track its changes. Finally, we define a computed observable that returns a user-friendly version of the discount percentage.

Let's go ahead and add a 20% discount to the first item in `PersonViewModel.shoppingCart`:

```
      this.shoppingCart = ko.observableArray([
        new Product("Beer", 10.99, null, .20),
        new Product("Brats", 7.99),
        new Product("Buns", 1.49, ['Baked goods', 'Hot dogs']);
      ]);
```

The if and ifnot Bindings

The `if` binding is a conditional binding. If the parameter you pass evaluates to true, the contained HTML will be displayed, otherwise it's removed from the DOM. For instance, try adding the following cell to the `<table>` containing the shopping cart items, right before the "Remove" button.

```
        <td data-bind='if: discount() > 0' style='color: red'>
          You saved <span data-bind='text:
formattedDiscount'></span>!!!
        </td>
```

Everything inside the `<td>` element will only appear for items that have a discount greater than `0`. Plus, since `discount` is an observable, Knockout.js will automatically re-evaluate the condition whenever it changes. This is just one more way Knockout.js helps you focus on the data driving your application.

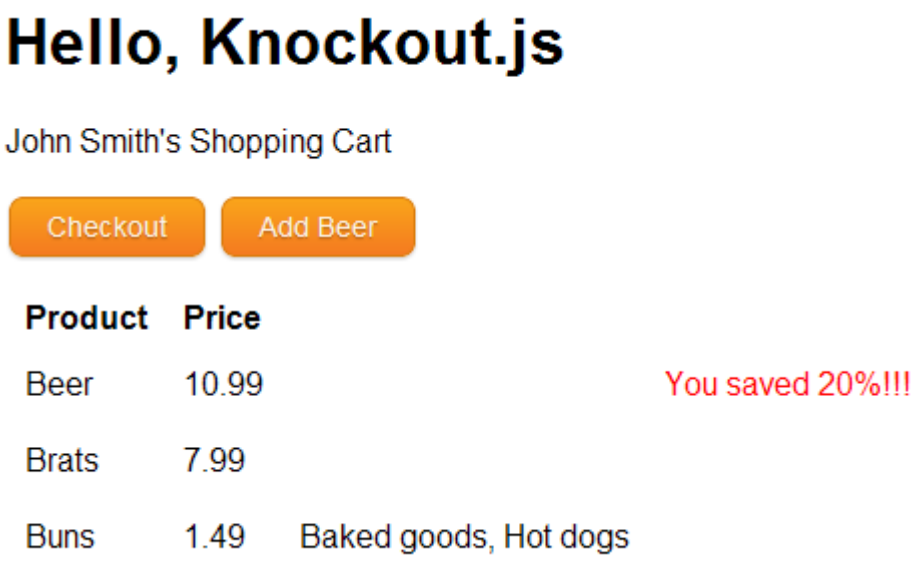

Figure 15: Conditionally rendering a discount for each product

You can use *any* JavaScript expression as the condition: Knockout.js will try to evaluate the string as JavaScript code and use the result to show or hide the element. As you might have guessed, the `ifnot` binding simply negates the expression.

The with Binding

The `with` binding can be used to manually declare the scope of a particular block. Try adding the following snippet towards the top of your view, before the "Checkout" and "Add Beer" buttons:

Sample code: item015.htm

```
<p data-bind='with: featuredProduct'>
  Do you need <strong data-bind='text: name'></strong>? <br />
  Get one now for only <strong data-bind='text: price'></strong>.
</p>
```

Inside of the `with` block, Knockout.js uses `PersonViewModel.featuredProduct` as the binding context. So, the `text: name` and `text: price` bindings work as expected without a reference to their parent object.

Of course, for the previous HTML to work, you'll need to define a `featuredProduct` property on `PersonViewModel`:

```
    var featured = new Product("Acme BBQ Sauce", 3.99);
    this.featuredProduct = ko.observable(featured);
```

Summary

This chapter presented the `foreach`, `if`, `ifnot`, and `with` bindings. These control-flow bindings give you complete control over how your ViewModel is displayed in a view.

It's important to realize the relationship between Knockout.js' bindings and observables. Technically, the two are entirely independent. As we saw at the very beginning of this book, you can use a normal object with native JavaScript properties (i.e. *not* observables) as your ViewModel, and Knockout.js will render the view's bindings correctly. However, Knockout.js will only process the template the first time around—without observables, it can't automatically update the view when the underlying data changes. Seeing as how this is the whole point of Knockout.js, you'll typically see bindings refer to *observable* properties, like our `foreach: shoppingCart` binding in the previous examples.

Now that we can control the logic behind our view templates, we can move on to controlling the appearance of individual HTML elements. The next chapter digs into the fun part of Knockout.js: appearance bindings.

Chapter 5 Appearance Bindings

In the previous chapter, we saw how Knockout.js' control-flow bindings provide a basic templating system for view code. Control-flow bindings provide the visual structure for your application, but a full-fledged templating system needs more than just structure. Knockout.js' appearance bindings give you precise control over the styles and formatting of individual elements.

As of this writing, Knockout.js ships with six bindings for controlling the appearance of HTML elements:

- `text: <value>`—Set the contents of an element.
- `html: <value>`—Set the HTML contents of an element.
- `visible: <condition>`—Show or hide an element based on certain conditions.
- `css: <object>`—Add CSS classes to an element.
- `style: <object>`—Define the `style` attribute of an element.
- `attr: <object>`—Add arbitrary attributes to an element.

Like all Knockout.js bindings, appearance bindings always occur inside of the `data-bind` attribute of an HTML element. But unlike the control-flow bindings of the previous chapter, appearance bindings only affect their associated element—they do *not* alter template blocks or change the binding context.

The text Binding

The `text` binding is the bread and butter of Knockout.js. As we've already seen, the `text` binding displays the value of a property inside of an HTML element:

```
        <td data-bind='text: name'></td>
```

You should really only use the `text` binding on text-level elements (e.g., `<a>`, `<em>`, `<span>`, etc.), although technically it can be applied to any HTML element. As its parameter, the `text` binding takes any data type, and it casts it to a string before rendering it. The `text` binding will escape HTML entities, so it can be used to safely display user-generated content.

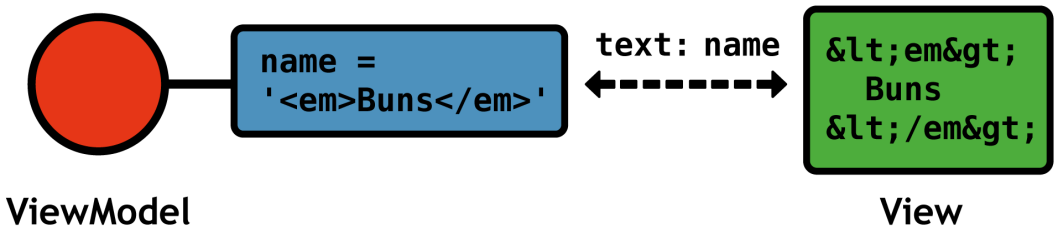

Figure 16: The `text` binding automatically escaping HTML entities in the view

It's also worth pointing out that Knockout.js manages cross-browser issues behind the scenes. For IE, it uses the `innerText` property, and for Firefox and related browsers it uses `textContent`.

The html Binding

The `html` binding allows you to render a string as HTML markup. This can be useful if you want to dynamically generate markup in a ViewModel and display it in your template. For example, you could define a computed observable called `formattedName` on our `Product` object that contains some HTML:

```
function Product(name, price, tags, discount) {
  ...
  this.formattedName = ko.computed(function() {
    return "<strong>" + this.name() + "</strong>";
  }, this);
}
```

Then, you could render the formatted name with the `html` binding:

```
<span data-bind='html: featuredProduct().formattedName'></span>
```

While this defeats the goal of separating content from presentation, the `html` binding can prove to be a versatile tool when used judiciously.

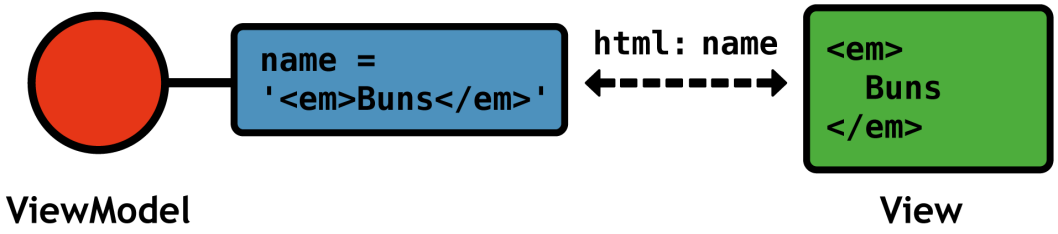

Figure 17: The `html` binding rendering HTML entities in the view

Whenever you render dynamic HTML—whether via the `html` binding or ASP.NET—always make sure that the markup has been validated. If you need to display content that can't be trusted, you should use the `text` binding instead of `html`.

In the previous snippet, also notice that `featuredProduct` is an observable, so the underlying object must be referenced with an empty function call instead of directly accessing the property with `featuredProduct.formattedName`. Again, this is a common mistake for Knockout.js beginners.

The visible Binding

Much like the `if` and `ifnot` bindings, the `visible` binding lets you show or hide an element based on certain conditions. But, instead of completely removing the element from the DOM, the `visible` binding simply adds a `display: none` declaration to the element's `style` attribute. For example, we can change our existing `if` binding to a `visible` binding:

```
        <td data-bind='visible: discount() > 0' style='color: red'>
```

The resulting HTML for both the `if` and the `visible` versions is shown in the following code sample. This example assumes the condition evaluates to `false`:

```
<!-- Using if binding: -->
<td data-bind="if: discount() > 0" style="color: red"></td>

<!-- Using visible binding: -->
<td data-bind='visible: discount() > 0'
    style='color: red; display: none'>
  You saved <span data-bind='text: formattedDiscount'></span>!!!
</td>
```

Deciding when to use `visible` versus `if` is largely determined by context. In this case, it's actually better to use the `if` binding so the empty `<td>` creates an equal number of columns for each row.

This binding takes the same parameter as the `if` and `ifnot` bindings. The condition can be a property of your ViewModel, a JavaScript expression, or a function that returns a Boolean.

The css Binding

The `css` binding lets you define CSS classes for HTML elements based on certain conditions. Instead of taking a condition as its parameter, it takes an object containing CSS class names as property names and conditions for applying the class as values. This is best explained with an example.

Let's say you want to draw extra attention to a product's discount when it's more than 15% off. One way to do this would be to add a `css` binding to the "You save __%" message inside of the `<table>` that displays all of our shopping cart items:

Sample code: item016.htm

```
        <td data-bind='if: discount() > 0' style='color: red'>
          You saved <span data-bind='text: formattedDiscount,
                     css: {supersaver: discount() > .15}'></span>!!!
```

```
        </td>
```

First, you'll notice that it's possible to add multiple bindings to a single `data-bind` attribute by separating them with commas. Second, the `css` binding takes the `{supersaver: discount() > .15}` object as its argument. This is like a mapping that defines when a CSS class should be added to the element. In this case, the `.supersaver` class will be added whenever the product's discount is greater than 15%, and removed otherwise. The actual CSS defining the `.supersaver` rule can be defined anywhere in the page (i.e. an external or internal style sheet).

```
.supersaver {
  font-size: 1.2em;
  font-weight: bold;
}
```

If you add a 10% discount to the second product, you should see our `css` binding in action:

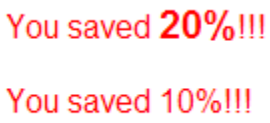

Figure 18: The `css` binding applying a class when `discount() > .15`

The condition contained in the object's property is the same as the `if`, `ifnot`, and `visible` bindings' parameter. It can be a property, a JavaScript expression, or a function.

The style Binding

The `style` binding provides the same functionality as the `css` binding, except it manipulates the element's `style` attribute instead of adding or removing classes. Since inline styles require a key-value pair, the syntax for this binding's parameter is slightly different, too:

```
        You saved <span data-bind='text: formattedDiscount,
                    style: {fontWeight: discount() > .15 ? "bold" :
"normal"}'></span>!!!
```

If the product's discount is greater than 15%, Knockout.js will render this element as the following:

```
        <td style='color: red; font-weight: bold'>
```

But, if it's less than 15%, it will have a `font-weight` of `normal`. Note that the `style` binding can be used in conjunction with an element's existing `style` attribute.

The attr Binding

The `attr` binding lets you dynamically define attributes on an HTML element using ViewModel properties. For example, if our `Product` class had a `permalink` property, we could generate a link to individual product pages with:

```
  <p><a data-bind='attr: {href: featuredProduct().permalink}'>View
details</a></p>
```

This adds an `href` attribute to the `<a>` tag pointing to whatever is stored in the `permalink` property. And of course, if `permalink` is an observable, you can leverage all the benefits of Knockout.js' automatic dependency tracking. Since permalinks are typically stored with the data object in persistent storage (e.g., a blog entry), dynamically generating links in this fashion can be very convenient.

But, the `attr` binding can do more than just create links. It lets you add *any* attribute to an HTML element. This opens up all kinds of doors for integrating your Knockout.js templates with other DOM libraries.

Summary

This chapter introduced Knockout.js' appearance bindings. Many of these bindings change an HTML element when a particular condition has been met. Defining these conditions directly in the binding is an intuitive way to design templates, and it keeps view-centric code outside of the ViewModel.

Remember, Knockout.js' goal is to let you focus on the data behind your application by automatically synchronizing the view whenever the data changes. Once you've defined your bindings, you never have to worry about them again (unless you change the structure of your ViewModel, of course).

The appearance bindings presented in this chapter provide all the tools you need to *display* your data, but they don't let us add any user interaction to our view components. In the next chapter, we'll take a look at how Knockout.js manages form fields.

Chapter 6 Interactive Bindings

Form elements are the conventional way to interact with users through a webpage. Working with forms in Knockout.js is much the same as working with appearance bindings. But, since users can edit form fields, Knockout.js manages updates *in both directions*. This means that interactive bindings are *two-way*. They can be set programmatically and the view will update accordingly, *or* they can be set by the view and read programmatically.

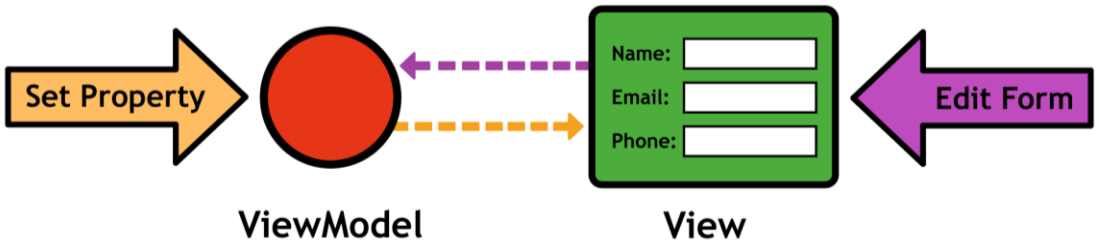

Figure 19: Knockout.js propagating changes in both directions

For example, you can set the value of a text input field from the ViewModel and it will be displayed in the view. But, the user typing something into the input field causes the associated property on the ViewModel to update, too. The point is, Knockout.js always makes sure that the view and the ViewModel are synchronized.

Knockout.js includes 11 bindings for interacting with the user:

- `click: <method>`—Call a ViewModel method when the element is clicked.
- `value: <property>`—Link a form element's value to a ViewModel property.
- `event: <object>`—Call a method when a user-initiated event occurs.
- `submit: <method>`—Call a method when a form is submitted.
- `enable: <property>`—Enable a form element based on a certain condition.
- `disable: <property>`—Disable a form element based on a certain condition.
- `checked: <property>`—Link a radio button or check box to a ViewModel property.
- `options: <array>`—Define a `<select>` element with a ViewModel array.
- `selectedOptions: <array>`—Define the active elements in a `<select>` field.
- `hasfocus: <property>`—Define whether or not the element is focused.

Like the appearance bindings presented in the previous chapter, these are all defined in the data-bind attribute of an HTML element. Some of them (like the `click` binding) work on any element, but others (like `checked`) can only be used with specific elements.

One of the major benefits of using Knockout.js to manage HTML forms is that you *still* only have to worry about the data. Whenever the user changes a form element's value, your ViewModel will automatically reflect the update. This makes it very easy to integrate user input into the rest of your application.

An HTML Form

This chapter uses a new HTML page for the running example. Instead of a shopping cart display page, we'll be working with a registration form for new customers. Create a new HTML file called `interactive-bindings.html` and add the following:

Sample code: item017.htm

```
<html lang='en'>
<head>
  <title>Interactive Bindings</title>
  <meta charset='utf-8' />
  <link rel='stylesheet' href='../style.css' />
</head>
<body>
  <h1>Interactive Bindings</h1>

  <form action="#" method="post">
    <!-- ToDo -->
  </form>

  <script type='text/javascript' src='knockout-2.1.0.js'></script>
  <script type='text/javascript'>
    function PersonViewModel() {
      var self = this;
      this.firstName = ko.observable("John");
      this.lastName = ko.observable("Smith");
    }

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

This is a simplified version of what we've been working with throughout the book. In this chapter, we'll only be worrying about *configuring* form elements. Processing form submissions is left for the next chapter.

The click Binding

The `click` binding is one of the simplest interactive bindings. It just calls a method of your ViewModel when the user clicks the element. For example, add the following button inside of the `<form>` element:

```
<p><button data-bind='click: saveUserData'>Submit</button></p>
```

When the user clicks the button, Knockout.js calls the `saveUserData()` method on `PersonViewModel`. In addition, it passes two parameters to the handler method: the current model and the DOM event. A `saveUserData()` method utilizing both of these parameters would look something like:

```
      this.saveUserData = function(model, event) {
        alert(model.firstName() + " is trying to checkout!");
        if (event.ctrlKey) {
          alert("He was holding down the Control key for some
reason.");
        }
      };
```

In this particular example, `model` refers to the top-level ViewModel instance, and `event` is the DOM event trigged by the user's click. The model argument will always be the *current* ViewModel, which makes it possible to access individual list items in a `foreach` loop. This is how we implemented the `removeProduct()` method in Chapter 3.

The value Binding

The `value` binding is very similar to the text binding we've been using throughout this book. The key difference is that it can be changed by the *user*, and the ViewModel will update accordingly. For instance, we can link the `firstName` and `lastName` observables with an input field by adding the following HTML to the form (before the `<button>`):

```
    <p>First name: <input data-bind='value: firstName' /></p>
    <p>Last name: <input data-bind='value: lastName' /></p>
```

The `value: firstName` binding makes sure that the `<input>` element's text is always the same as the ViewModel's `firstName` property, regardless of whether it's changed by the user or by your application. The same goes for the `lastName` property.

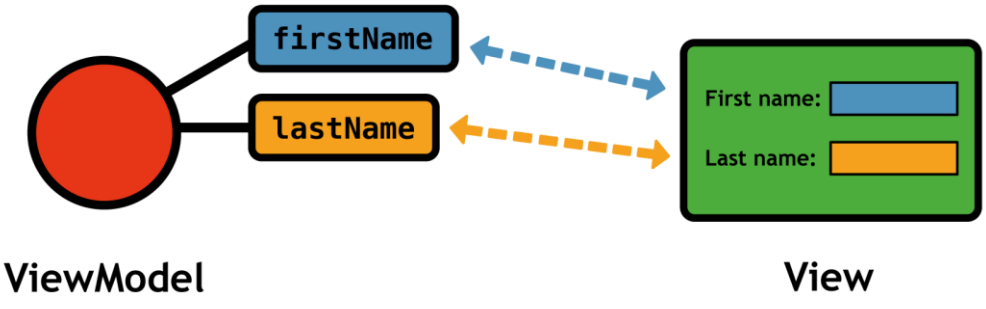

Figure 20: Two-way connections between observables and form fields

We can examine this further by including a button for displaying the user's name and another to set it programmatically. This lets us see how the `value` binding works from both ends:

```
<p>
  <button data-bind='click: displayName'>
    Display Name
  </button>
  <button data-bind='click: setName'>
    Set Name
  </button>
</p>
```

The handler methods should look something like the following:

Sample code: item019.htm

```
this.displayName = function() {
  alert(this.firstName());
};
this.setName = function() {
  this.firstName("Bob");
};
```

Clicking **Display Name** will read the ViewModel's `firstName` property, which should match the `<input>` element, even if it has been edited by the user. The **Set Name** button sets the value of the ViewModel's property, causing the `<input>` element to update. The behavior of the latter is essentially the same as a normal `text` binding.

Once again, the whole point behind this two-way synchronization is to let you focus on your data. After you set up a `value` binding, you can completely forget about HTML form elements. Simply get or set the associated property on the ViewModel and Knockout.js will take care of the rest.

We won't be needing the `displayName` and `setName` methods or their respective buttons, so you can go ahead and delete them if you like.

The event Binding

The `event` binding lets you listen for arbitrary DOM events on any HTML element. It's like a generic version of the `click` binding. But, because it can listen for multiple events, it requires an object to map events to methods (this is similar to the `attr` binding's parameter). For example, we can listen for `mouseover` and `mouseout` events on the first `<input>` element with the following:

```
    <p data-bind='event: {mouseover: showDetails, mouseout:
hideDetails}'>
      First name: <input data-bind='value: firstName' />
    </p>
```

When the user fires a `mouseover` event, Knockout.js calls the `showDetails()` method of our ViewModel. Likewise, when he or she leaves the element, `hideDetails()` is called. Both of these take the same parameters as the `click` binding's handlers: the target of the event and the event object itself. Let's implement these methods now:

```
      this.showDetails = function(target, event) {
          alert("Mouse over");
      };
      this.hideDetails = function(target, event) {
          alert("Mouse out");
      };
```

Now, when you interact with the **First name** field, you should see both messages pop up. But, instead of just displaying an alert message, let's show some extra information for each form field when the user rolls over it. For this, we need another observable on `PersonViewModel`:

```
      this.details = ko.observable(false);
```

The `details` property acts as a toggle, which we can switch on and off with our event handler methods:

```
      this.showDetails = function(target, event) {
        this.details(true);
      };
      this.hideDetails = function(target, event) {
        this.details(false);
      };
```

Then we can combine the toggle with the `visible` binding to show or hide form field details in the view:

Sample code: item020.htm

```
    <p data-bind='event: {mouseover: showDetails, mouseout:
hideDetails}'>
      First name: <input data-bind='value: firstName' />
```

```
    <span data-bind='visible: details'>Your given name</span>
  </p>
```

The contents of the `<span>` should appear whenever you mouse over the **First name** field and disappear when you mouse out. This is pretty close to our desired functionality, but things get more complicated once we want to display details for more than one form field. Since we only have one toggle variable, displaying details is an all-or-nothing proposition—either details are displayed for *all* of the fields, or for none of them.

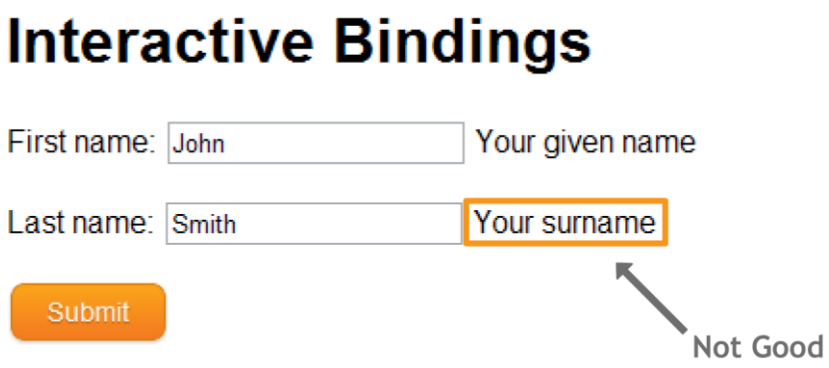

Figure 21: Toggling all form field details simultaneously

One way to fix this is by passing a custom parameter to the handler function.

Event Handlers with Custom Parameters

It's possible to pass custom parameters from the view into the event handler. This means you can access arbitrary information from the view into the ViewModel. In our case, we'll use a custom parameter to identify which form field should display its details. Instead of a toggle, the `details` observable will contain a string representing the selected element. First, we'll make some slight alterations in the ViewModel:

```
this.details = ko.observable("");

this.showDetails = function(target, event, details) {
  this.details(details);
}
this.hideDetails = function(target, event) {
  this.details("");
}
```

The only big change here is the addition of a `details` parameter to the `showDetails()` method. We don't need a custom parameter for the `hideDetails()` function since it just clears the details `observable`.

Next, we'll use a function literal in the `event` binding to pass the custom parameter to `showDetails()`:

```
    <p data-bind='event: {mouseover: function(data, event) {
         showDetails(data, event, "firstName")
       }, mouseout: hideDetails}'>
```

The function literal for `mouseover` is a wrapper for our `showDetails()` handler, providing a straightforward means to pass in extra information. The `mouseout` handler remains unchanged. Finally, we need to update the `<span>` containing the details:

```
      <span data-bind='visible: details() == "firstName"'>Your given
name</span>
```

The **First name** form field should display its detailed description when you mouse over and hide when you mouse out, just like it did in the previous section. Only now, it's possible to add details to more than one field by changing the custom parameter. For example, you can enable details for the **Last name** input element with:

```
    <p data-bind='event: {mouseover: function(data, event) {
         showDetails(data, event, "lastName")
       }, mouseout: hideDetails}'>
      Last name: <input data-bind='value: lastName' />
      <span data-bind='visible: details() == "lastName"'>Your
surname</span>
```

Event bindings can be a little bit complicated to set up, but once you understand how they work, they enable limitless possibilities for reactive design. The `event` binding can even connect to jQuery's animation functionality, which is discussed in Chapter 8. For now, we'll finish exploring the rest of Knockout.js' interactive bindings. Fortunately for us, none of them are nearly as complicated as `event` bindings.

The enable/disable Bindings

The `enable` and `disable` bindings can be used to enable or disable form fields based on certain conditions. For example, let's say you wanted to record a primary and a secondary phone number for each user. These could be stored as normal observables on `PersonViewModel`:

```
      this.primaryPhone = ko.observable("");
      this.secondaryPhone = ko.observable("");
```

The `primaryPhone` observable can be linked to a form field with a normal `value` binding:

```
    <p>
      Primary phone: <input data-bind='value: primaryPhone' />
    </p>
```

However, it doesn't make much sense to enter a secondary phone number without specifying a primary one, so we activate the `<input>` for the secondary phone number only if `primaryPhone` is not empty:

```
    <p>
      Secondary phone: <input data-bind='value: secondaryPhone,
          enable: primaryPhone' />
    </p>
```

Now users will only be able to interact with the **Secondary phone** field if they've entered a value for `primaryPhone`. The `disable` binding is a convenient way to negate the condition, but otherwise works exactly like `enable`.

The checked Binding

`checked` is a versatile binding that exhibits different behaviors depending on how you use it. In general, the `checked` binding is used to select and deselect HTML's checkable form elements: check boxes and radio buttons.

Simple Check Boxes

Let's start with a straightforward check box:

```
    <p>Annoy me with special offers: <input data-bind='checked:
annoyMe' type='checkbox' /></p>
```

This adds a check box to our form and links it to the `annoyMe` property of the ViewModel. As always, this is a two-way connection. When the user selects or deselects the box, Knockout.js updates the ViewModel, and when you set the value of the ViewModel property, it updates the view. Don't forget to define the `annoyMe` observable:

```
      this.annoyMe = ko.observable(true);
```

Using the `checked` binding in this fashion is like creating a one-to-one relationship between a single check box and a Boolean observable.

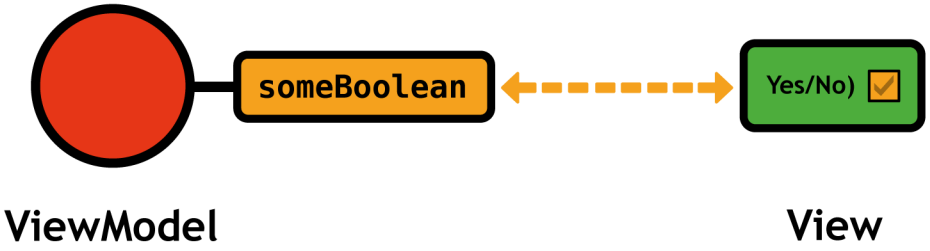

Figure 22: Connecting a Boolean observable with a single check box

Check-box Arrays

It's also possible to use the `checked` binding with arrays. When you bind a check box to an observable array, the selected boxes correspond to elements contained in the array, as shown in the following figure:

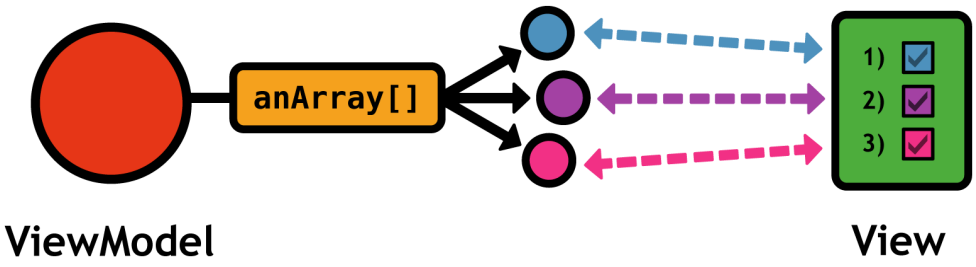

Figure 23: Connecting an observable array with multiple check boxes

For instance, consider the following observable:

```
this.annoyTimes = ko.observableArray(['morning', 'evening']);
```

We can connect the items in this observable array to check boxes using the `value` attribute on each `<input>` element:

Sample code: item022.htm

```
    <p>Annoy me with special offers: <input data-bind='checked:
annoyMe' type='checkbox' /></p>
    <div data-bind='visible: annoyMe'>
      <div>
        <input data-bind='checked: annoyTimes'
```

```
            value='morning'
            type='checkbox' />
      In the morning
    </div>
    <div>
      <input data-bind='checked: annoyTimes'
            value='afternoon'
            type='checkbox' />
      In the afternoon
    </div>
    <div>
      <input data-bind='checked: annoyTimes'
            value='evening'
            type='checkbox' />
      In the evening
    </div>
  </div>
```

This uses the `annoyMe` property from the previous chapter to toggle a list of check boxes for selecting when it would be a good time to be annoyed. Since `value='morning'` is on the first check box, it will be selected whenever the `"morning"` string is in the `annoyTimes` array. The same goes for the other check boxes. `"morning"` and `"evening"` are the initial contents of the array, so you should see something like the following in your webpage:

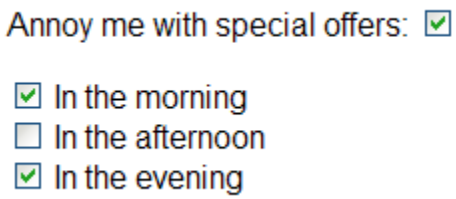

Figure 24: Check boxes displaying the initial state of the `annoyTimes` observable array

And since we're using an *observable* array, the connection is two-way—deselecting any of the boxes will remove the corresponding string from the `annoyTimes` array.

Radio Buttons

The last context for the `checked` binding is in a radio button group. Instead of a Boolean or an array, radio buttons connect their `value` attribute to a string property in the ViewModel. For example, we can turn our check-box array into a radio button group by first changing the `annoyTimes` observable to a string:

```
    this.annoyTimes = ko.observable('morning');
```

Then, all we have to do is turn the `<input>` elements into radio buttons:

```
<input data-bind='checked: annoyTimes'
       value='morning'
       type='radio'
       name='annoyGroup' />
```

Each `<input>` should have `"radio"` as its `type` and `"annoyGroup"` as its `name`. The latter doesn't have anything to do with Knockout.js—it just adds all of them to the same HTML radio button group. Now, the `value` attribute of the selected radio button will always be stored in the `annoyTimes` property.

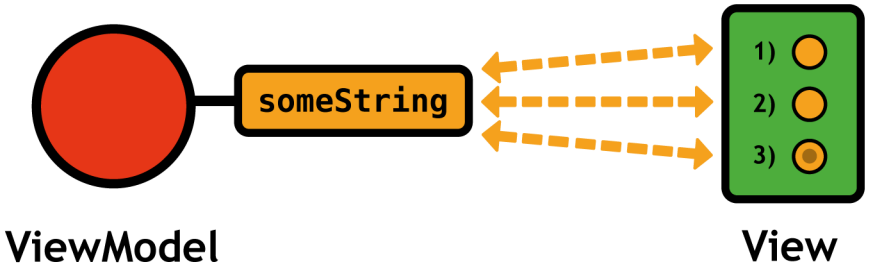

Figure 25: Connecting an observable string with multiple radio buttons

The options Binding

The `options` binding defines the contents of a `<select>` element. This can take the form of either a drop-down list or a multi-select list. First, we'll take a look at drop-down lists. Let's edit the `annoyTimes` property one more time:

```
this.annoyTimes = ko.observableArray([
  'In the morning',
  'In the afternoon',
  'In the evening'
]);
```

Then we can bind it to a `<select>` field with:

```
<div data-bind='visible: annoyMe'>
  <select data-bind='options: annoyTimes'></select>
</div>
```

You should now have a drop-down list instead of a radio button group, but it's no use having such a list if you can't figure out which item is selected. For this, we can reuse the `value` binding from earlier in the chapter:

```
      <select data-bind='options: annoyTimes, value:
selectedTime'></select>
```

This determines which property on the ViewModel contains the selected string. We still need to define this property:

Sample code: item024.htm

```
      this.selectedTime = ko.observable('In the afternoon');
```

Again, this relationship goes both ways. Setting the value of `selectedTime` will change the selected item in the drop-down list, and vice versa.

Using Objects as Options

Combining the options and the value bindings give you all the tools you need to work with drop-down lists that contain strings. However, it's often much more convenient to select entire JavaScript objects using a drop-down list. For example, the following defines a list of products reminiscent of the previous chapter:

```
      this.products = ko.observableArray([
        {name: 'Beer', price: 10.99},
        {name: 'Brats', price: 7.99},
        {name: 'Buns', price: 2.99}
      ]);
```

When you try to create a `<select>` element out of this, all of your objects will be rendered as `[object Object]`:

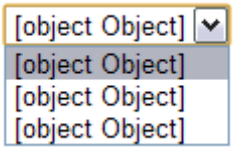

Figure 26: Attempting to use objects with the `options` binding

Fortunately, Knockout.js lets you pass an `optionsText` parameter to define the object property to render in the `<select>` element:

```
      <select data-bind='options: products,
```

```
        optionsText: "name",
        value: favoriteProduct'></select>
```

For this snippet to work, you'll also have to define a `favoriteProduct` observable on your ViewModel. Knockout.js will populate this property with an *object* from `PersonViewModel.products`—not a string like it did in the previous section.

The selectedOptions Binding

The other rendering possibility for HTML's `<select>` element is a multi-select list. Configuring a multi-select list is much like creating a drop-down list, except that instead of *one* selected item, you have an *array* of selected items. So, instead of using a `value` binding to store the selection, you use the `selectedOptions` binding:

```
<select data-bind='options: products,
      optionsText: "name",
      selectedOptions: favoriteProducts'
    size='3'
    multiple='true'></select>
```

The `size` attribute defines the number of visible options, and `multiple='true'` turns it into a multi-select list. Instead of a string property, `favoriteProducts` should point to an array:

Sample code: item025.htm

```
var brats = {name: 'Brats', price: 7.99};
this.products = ko.observableArray([
    {name: 'Beer', price: 10.99},
    brats,
    {name: 'Buns', price: 2.99}
]);
this.favoriteProducts = ko.observableArray([brats]);
```

Note that we needed to provide the same object reference (`brats`) to both `products` and `favoriteProducts` for Knockout.js to initialize the selection correctly.

The hasfocus Binding

And so, we come to our final interactive binding: `hasfocus`. This aptly named binding lets you manually set the focus of an interactive element using a ViewModel property. If, for some strange reason, you'd like the "Primary phone" field to be the initial focus, you can add a `hasfocus` binding, like so:

```
<p>
  Primary phone: <input data-bind='value: primaryPhone,
    hasfocus: phoneHasFocus' />
</p>
```

Then you can add a Boolean observable to tell Knockout.js to give it focus:

Sample code: item026.htm

```
    this.phoneHasFocus = ko.observable(true);
```

By setting this property elsewhere in your application, you can precisely control the flow of focus in your forms. In addition, you can use `hasfocus` to track the user's progress through multiple form fields.

Summary

This chapter covered interactive bindings, which leverage Knockout.js' automatic dependency tracking against HTML's form fields. Unlike appearance bindings, interactive bindings are *two-way* bindings—changes to the user interface components are automatically reflected in the ViewModel, and assignments to ViewModel properties trigger Knockout.js to update the view accordingly.

Interactive bindings, appearance bindings, and control-flow bindings compose Knockout.js' templating toolkit. Their common goal is to provide a data-centric interface for your web applications. Once you define the presentation of your data using these bindings, all you have to worry about is manipulating the underlying ViewModel. This is a much more robust way to develop dynamic web applications.

This chapter discussed forms from the perspective of the view and the ViewModel. Interactive bindings are an intuitive, scalable method for accessing user input, but we have yet to discuss how to get this data out of the front-end and into a server-side script. The next chapter addresses this issue by integrating Knockout.js with jQuery's AJAX functionality.

Chapter 7 Accessing External Data

For most web applications, collecting user input is relatively useless if you can't pass that data along to a server. In this chapter, we'll learn how to send and receive information from a server using AJAX requests. This puts the *model* back into the Model-View-ViewModel design pattern underpinning Knockout.js.

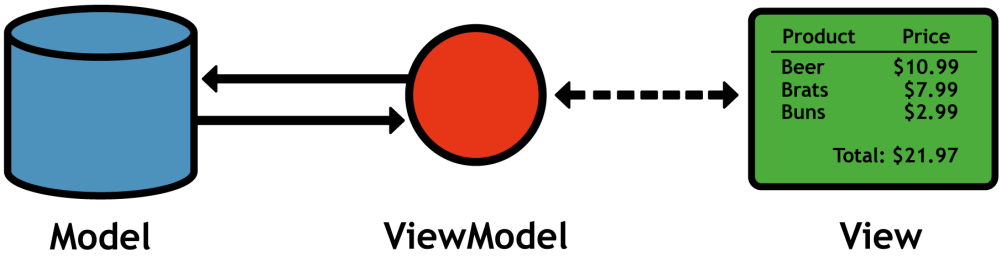

Figure 27: Adding the model back into our MVVM pattern

Remember that Knockout.js is designed to be compatible with any other client-side or server-side technology. This book uses jQuery's `$.getJSON()` and `$.post()` functions, but you're free to use any JavaScript framework that can send and receive JSON data. Similarly, the server-side scripting language is completely up to you. Instead of presenting back-end code samples, this chapter simply includes the JSON data expected by Knockout.js. Generating this output should be trivial to implement in any modern scripting language.

A New HTML Form

We're going to use a fresh HTML page to experiment with Knockout.js/AJAX integration. Since this page will have to interact with some server-side code, make sure it's accessible from the document root of your local server. We'll start out with something similar to the previous chapter:

```
<html lang='en'>
<head>
  <title>External Data</title>
  <meta charset='utf-8' />
  <link rel='stylesheet' href='style.css' />
</head>
<body>
  <h1>External Data</h1>

  <form action="#" method="post">
    <p>First name: <input data-bind='value: firstName' /></p>
    <p>Last name: <input data-bind='value: lastName' /></p>
```

```
    <div>
      Your favorite food:
      <select data-bind='options: activities,
          value: favoriteHobby'></select>
    </div>
    <p><button data-bind='click: loadUserData'>Load Data</button></p>
  </form>

  <script type='text/javascript' src='knockout-2.1.0.js'></script>
  <script type='text/javascript' src='jquery-1.7.2.js'></script>
  <script type='text/javascript'>
    function PersonViewModel() {
      var self = this;
      self.firstName = ko.observable("");
      self.lastName = ko.observable("");
      self.activities = ko.observableArray([]);
      self.favoriteHobby = ko.observable("");
    }

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

This is a basic form with a few `<input>` fields so we can see how to send and receive information from the server. Notice that we also include the jQuery library before our custom `<script>` element.

Loading Data

You probably noticed that unlike previous chapters, all of our observables are empty. Instead of hard-coding data into our ViewModel, we're going to load it from a server using jQuery's `$.getJSON()` method. First, let's make a button for loading data (typically, you would automatically load the data when your application starts up, but this way we can see how everything works step-by-step):

```
    <p><button data-bind='click: loadUserData'>Load Data</button></p>
```

The handler for this button uses `$.getJSON()` to call a server-side script:

```
      self.loadUserData = function() {
        $.getJSON("/get-user-data", function(data) {
          alert(data.firstName);
        });
      }
```

The `/get-user-data` string should be the path to the script. Again, as long as it can encode and decode JSON, any server-side language can be used with Knockout.js. For our example, it should return a JSON-formatted string that looks something like the following:

```
{"firstName":"John",
 "lastName":"Smith",
 "activities":[
    "Golf",
    "Kayaking",
    "Web Development"],
  "favoriteHobby":"Golf"
}
```

The `$.getJson()` method automatically translates this string back into a JavaScript object and passes it to the handler method via the `data` parameter. It's trivial to update our ViewModel with the new information:

```
      self.loadUserData = function() {
        $.getJSON("/get-user-data", function(data) {
          self.firstName(data.firstName);
          self.lastName(data.lastName);
          self.activities(data.activities);
          self.favoriteHobby(data.favoriteHobby);
        });
      }
```

After clicking the **Load Data** button, `$.getJSON()` loads data from the server and uses it to update all of our ViewModel's observables. As always, Knockout.js automatically updates the form fields to match.

Saving Data

For normal web applications, saving data is a simple matter of converting objects to JSON and sending it to the server with something like jQuery's `$.post()` method. Things are somewhat more complicated for Knockout.js applications. It's not possible to use a standard JSON serializer to convert the object to a string because ViewModels use observables instead of normal JavaScript properties. Remember that observables are actually functions, so trying to serialize them and send the result to a server would have unexpected results.

Fortunately, Knockout.js provides a simple solution to this problem: the `ko.toJSON()` utility function. Passing an object to `ko.toJSON()` replaces all of the object's observable properties with their current value and returns the result as a JSON string.

Create another button called “Save Data” and point it to a `saveUserData()` method on the ViewModel. Then, you can see the JSON generated by `ko.toJSON()` with the following:

```
self.saveUserData = function() {
  alert(ko.toJSON(self));
}
```

Clicking this button should display the current data in your form fields transformed into a JSON string. Now that we’ve gotten rid of all our observables, we can send this to the server for processing:

```
self.saveUserData = function() {
  var data_to_send = {userData: ko.toJSON(self)};
  $.post("/save-user-data", data_to_send, function(data) {
    alert("Your data has been posted to the server!");
  });
}
```

This sends the JSON string representing your ViewModel to a script called `/save-user-data` using the POST method. As a result, your script should find the string under a `userData` entry in its POST dictionary. You can then deserialize the JSON string into an object, save it into your database, or do whatever kind of server-side processing you need to do.

Mapping Data to ViewModels

The loading and saving mechanisms covered in the previous two sections provide everything you need to create rich user interfaces backed by an arbitrary server-side scripting language. However, manually mapping loaded data to observables can become quite tedious if you’re working with more than just a few properties.

The `mapping` plug-in for Knockout.js solves this problem by letting you automatically map JSON objects loaded from the server to ViewModel observables. In essence, `mapping` is a generic version of our `saveUserData()` and `loadUserData()` methods.

The `mapping` plug-in is released as a separate project, so we’ll need to download it and include it in our HTML page before using it:

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

Next, we’re going to completely replace our `PersonViewModel`. In its place, we’ll use jQuery’s `$.getJSON()` method to load some initial data from the server and let the

`mapping` plug-in dynamically generate observables. Replace the entire custom `<script>` element with the following:

```
<script type='text/javascript'>
  $.getJSON("/get-user-data", function(data) {
    var viewModel = ko.mapping.fromJS(data);
    ko.applyBindings(viewModel);
  });
</script>
```

When our application loads, it immediately makes an AJAX request for the initial user data. Your server-side script for `/get-intial-data` should return the same thing as the sample JSON output from the Loading Data section of this chapter. Once the data is loaded, we create a ViewModel via `ko.mapping.fromJS()`. This takes the native JavaScript object generated by the script and turns each property into an observable. Aside from the `saveUserData()` and `loadUserData()` methods, this dynamically generated ViewModel has the exact same functionality as `PersonViewModel`.

At this point, we've only *initialized* our ViewModel with data from the server. The `mapping` plug-in also lets us *update* an existing ViewModel in the same fashion. Let's go ahead and add an explicit `loadUserData()` method back to the ViewModel:

```
    viewModel.loadUserData = function() {
      $.getJSON("/get-user-data", function(data) {
        ko.mapping.fromJS(data, viewModel);
      });
    }
```

In the old version of `loadUserData()`, we had to manually assign each data property to its respective observable. But now, the `mapping` plug-in does all of this for us. Note that passing the `data` object as the first argument to `ko.mapping.fromJS()` causes it to *update* the ViewModel instead of *initializing* it.

The `mapping` plug-in only relates to loading data, so `saveUserData()` remains unaffected except for the fact that it needs to be assigned to the `viewModel` object:

```
    viewModel.saveUserData = function() {
      var data_to_send = {userData: ko.toJSON(viewModel)};
      $.post("/save-user-data", data_to_send, function(data) {
        alert("Your data has been posted to the server!");
      });
    }
```

And now we should be back to where we started at the beginning of this section—both the **Load Data** and **Save Data** buttons should work, and Knockout.js should keep the view and ViewModel synchronized.

While not a necessary plug-in for all Knockout.js projects, the `mapping` plug-in does make it possible to scale up to complex objects without adding an extra line of code for every new property you add to your ViewModel.

Summary

In this chapter, we learned how Knockout.js can communicate with a server-side script. Most of the AJAX-related functionality came from the jQuery web framework, although Knockout.js does provide a neat utility function for converting its observables into native JavaScript properties. We also discussed the `mapping` plug-in, which provided a generic way to convert a native JavaScript object to a ViewModel with observable properties.

Remember, Knockout.js is a pure client-side library. It's only for connecting JavaScript objects (ViewModels) with HTML elements. Once you have this relationship set up, you can use any other technology you like to communicate with the server. On the client-side, you could replace jQuery with Dojo, Prototype, MooTools, or any other framework that supports AJAX requests. On the server-side, you have the choice of ASP.NET, PHP, Django, Ruby on Rails, Perl, JavaServer Pages…you get the idea. This separation of concerns makes Knockout.js an incredibly flexible user interface development tool.

Chapter 8 Animating Knockout.js

Knockout.js is *not* an animation library. All of Knockout.js' automatic updates are *immediately* applied whenever the underlying data changes. In order to animate any of its changes, we need to dig into Knockout.js' internals and manually create animated transitions using another JavaScript framework like jQuery or MooTools. This chapter sticks with jQuery's animation routines, but the concepts presented apply to other animation libraries as well.

Return of the Shopping Cart

For this chapter, we'll return to a simplified version of our shopping cart example. Create a new HTML file with the following contents. We won't be making any AJAX requests, so feel free to put this anywhere on your computer. We will, however, be using jQuery's animation routines, so be sure to include a link to your copy of the jQuery library.

```
<html lang='en'>
<head>
  <title>Animating Knockout.js</title>
  <meta charset='utf-8' />
  <link rel='stylesheet' href='style.css' />
</head>
<body>
  <h1>Animating Knockout.js</h1>
  <table>
    <thead><tr>
      <th>Product</th>
      <th>Price</th>
      <th></th>
    </tr></thead>
    <tbody data-bind='foreach: items'>
      <tr>
        <td data-bind='text: name'></td>
        <td data-bind='text: price'></td>
        <td><button data-bind='click:
$root.removeProduct'>Remove</button></td>
      </tr>
    </tbody>
  </table>

  <button data-bind='click: addProduct'>Add Beer</button>

  <script type='text/javascript' src='knockout-2.1.0.js'></script>
  <script src='jquery-1.7.2.js'></script>
  <script type='text/javascript'>
    function Product(name, price, tags, discount, details) {
      this.name = ko.observable(name);
```

```
      this.price = ko.observable(price);
    }
    function ShoppingCart() {
      var self = this;
      this.instructions = ko.observable("");
      this.hasInstructions = ko.observable(false);

      this.items = ko.observableArray([
        new Product("Beer", 10.99),
        new Product("Brats", 7.99),
        new Product("Buns", 1.49)
      ]);

      this.addProduct = function() {
        this.items.push(new Product("More Beer", 10.99));
      };

      this.removeProduct = function(product) {
        self.items.destroy(product);
      };

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

Hopefully, this is all review by now. We have an observable array containing a bunch of products, a `foreach` binding that displays each one of them, and a button to add more items to the shopping cart.

List Callbacks

Knockout.js is a powerful user interface library on its own, but once you combine it with the animation capabilities of a framework like jQuery or MooTools, you're ready to create truly stunning UIs with minimal markup. First, we'll take a look at animating lists, and then the next section presents a more generic way to animate view components.

The `foreach` binding has two callbacks named `beforeRemove` and `afterAdd`. These functions are executed before an item is removed from the list or after it's been added to the list, respectively. This gives us an opportunity to animate each item before Knockout.js manipulates the DOM. Add the callbacks to the `<tbody>` element like so:

```
    <tbody data-bind='foreach: {data: items,
      beforeRemove: hideProduct,
      afterAdd: showProduct}'>
```

Instead of a property, our `foreach` binding now takes an object literal as its parameter. The parameter's `data` property points to the array you would like to render, and the `beforeRemove` and `afterAdd` properties point to the desired callback functions. Next, we should define these callbacks on the `ShoppingCart` ViewModel:

```
    this.showProduct = function(element) {
      if (element.nodeType === 1) {
        $(element).hide().fadeIn();
      }
    };

    this.hideProduct = function(element) {
      if (element.nodeType === 1) {
       $(element).fadeOut(function() { $(element).remove(); });
      }
    };
```

The `showProduct()` callback uses jQuery to make new list items gradually fade in, and the `hideProduct()` callback fades them out, and then removes them from the DOM. Both functions take the affected DOM element as their first parameter (in this case, it's a `<tr>` element). The conditional statements make sure that we're working with a full-fledged element and not a mere text node.

The end result should be list items that smoothly transition into and out of the list. Of course, you're free to use any of jQuery's other transitions or perform custom post-processing in either of the callbacks.

Custom Bindings

The `foreach` callbacks work great for animating lists, but unfortunately other bindings don't provide this functionality. So, if we want to animate other parts of the user interface, we have to create *custom* bindings that have the animation built right into them.

Custom bindings work just like Knockout.js' default bindings. For example, consider the following form fields:

```
<div>
  <p>
    <input data-bind='checked: hasInstructions'
           type='checkbox' />
    Requires special handling instructions
  </p>
  <div>
    <textarea data-bind='visible: hasInstructions,
                         value: instructions'>
    </textarea>
```

```
    </div>
  </div>
```

The check box acts as a toggle for the `<textarea>`, but since we're using the `visible` binding, Knockout.js abruptly adds or removes it from the DOM. To provide a smooth transition for the `<textarea>`, we'll create a custom binding called `visibleFade`:

```
    <textarea data-bind='visibleFade: hasInstructions,
                         value: instructions'>
```

Of course, this won't work until we add the custom binding to Knockout.js. We can do this by adding an object defining the binding to `ko.bindingHandlers` as shown in the following code sample. This also happens to be where all of the built-in bindings are defined, too.

```
ko.bindingHandlers.visibleFade = {
  init: function(element, valueAccessor) {
    var value = valueAccessor();
    $(element).toggle(value());
  },
  update: function(element, valueAccessor) {
    var value = valueAccessor();
    value() ? $(element).fadeIn() : $(element).fadeOut();
  }
}
```

The `init` property specifies a function to call when Knockout.js first encounters the binding. This callback should define the initial state for the view component and perform necessary setup actions (e.g., registering event listeners). For `visibleFade`, all we have to do is show or hide the element based on the state of the ViewModel. We implemented this using jQuery's `toggle()` method.

The `element` parameter is the DOM element being bound, and `valueAccessor` is a function that will return the ViewModel property in question. In our example, `element` refers to `<textarea>`, and `valueAccessor()` returns a reference to the `hasInstructions` observable.

The `update` property specifies a function to execute whenever the associated observable changes, and our callback uses the value of `hasInstructions` to transition the `<textarea>` in the appropriate direction. Remember that you need to call the observable to get its current value (i.e. `value()`, not `value`). However, if `hasInstructions` were a normal JavaScript property instead of an observable, this would not be the case.

Summary

In this chapter, we discovered two methods of animating Knockout.js view components. First, we added callback methods to the `foreach` binding, which let us delegate the addition and removal of items to a user-defined function. This gave us the opportunity to integrate jQuery's animated transitions into our Knockout.js template. Then, we explored custom bindings as a means to animate arbitrary elements.

This chapter presented a common use case for custom bindings, but they are by no means limited to animating UI components. Custom bindings can also be used to filter data as it is collected, listen for custom events, or create reusable widgets like grids and paged content. If you can encapsulate a behavior into an `init` and an `update` function, you can turn it into a custom binding.

Chapter 9 Conclusion

Knockout.js is a pure JavaScript library that makes it incredibly easy to build dynamic, data-centric user interfaces. We learned how to expose ViewModel properties using observables, bind HTML elements to those observables, manage user input with interactive bindings, export that data to a server-side script, and animate components with custom bindings. Hopefully, you're more than ready to migrate this knowledge to your real-world web applications.

This book covered the vast majority of the Knockout.js API, but there are still a number of nuances left to discover. These topics include: custom bindings for aggregate data types, the `throttle` extender for asynchronous evaluation of computed observables, and manually subscribing to an observable's events. However, all of these are advanced topics that shouldn't be necessary for the typical web application. Nonetheless, Knockout.js provides a plethora of extensibility opportunities for you to explore.

Appendix A

```
body {
  margin: 20px;
  font-family: "Arial", "Helvetica", sans-serif;
}

button {
  display: inline-block;
  outline: none;
  cursor: pointer;
  text-align: center;
  text-decoration: none;
  font: 14px/100% Arial, Helvetica, sans-serif;
  padding: .5em 1.3em .5em;
  text-shadow: 0 1px 1px rgba(0,0,0,.3);
  -webkit-border-radius: .5em;
  -moz-border-radius: .5em;
  border-radius: .5em;
  -webkit-box-shadow: 0 1px 2px rgba(0,0,0,.2);
  -moz-box-shadow: 0 1px 2px rgba(0,0,0,.2);
  box-shadow: 0 1px 2px rgba(0,0,0,.2);

  color: #fef4e9;
  border: solid 1px #da7c0c;
  background: #f78d1d;
  background: -webkit-gradient(linear, left top, left bottom,
from(#faa51a), to(#f47a20));
  background: -moz-linear-gradient(top,  #faa51a,  #f47a20);
  filter:
progid:DXImageTransform.Microsoft.gradient(startColorstr='#faa51a',
endColorstr='#f47a20');
}

button:hover {
  text-decoration: none;
  background: #f47c20;
  background: -webkit-gradient(linear, left top, left bottom,
from(#f88e11), to(#f06015));
  background: -moz-linear-gradient(top,  #f88e11,  #f06015);
  filter:
progid:DXImageTransform.Microsoft.gradient(startColorstr='#f88e11',
endColorstr='#f06015');
}

button:active {
  position: relative;
  top: 1px;
```

```
  color: #fcd3a5;
  background: -webkit-gradient(linear, left top, left bottom,
from(#f47a20), to(#faa51a));
  background: -moz-linear-gradient(top,  #f47a20,  #faa51a);
  filter:
progid:DXImageTransform.Microsoft.gradient(startColorstr='#f47a20',
endColorstr='#faa51a');
}


table {
  padding-top: 1em;
}


th {
  text-align: left;
}


th, td {
  padding: .1em .5em;
}


td li, td ul {
  margin: 0;
  padding: 0;
}


td li {
  display: inline;
}


td li::after {
  content: ',';
}


td li:last-child::after {
  content: '';
}
```