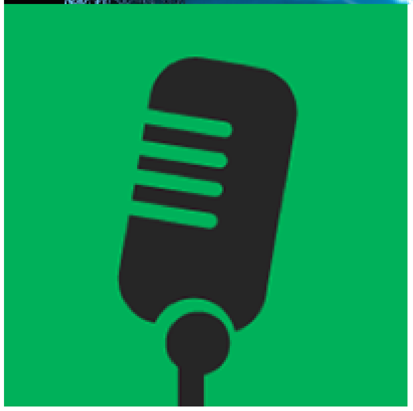

Control-Flow Binding

Ryan Hodson on Jan 12th 2013 with 0 Comments

Tutorial Details

-
- **Difficulty:** Intermediate
- **Completion Time:** 30 Minutes

View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

This entry is part 5 of 9 in the ***Knockout Succinctly*** *Session – Show All*

« PreviousNext »

As we've seen in previous lessons, 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 lesson introduces Knockout.js' control–flow bindings by extending the shopping cart example from the previous lesson. We'll also explore some of the nuances of `foreach` that were glossed over in the last lesson.

The `foreach` Binding

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

```
<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</butt
  </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 lesson, 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:

```
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<
```

```
            </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:

```
<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:

```
<ul data-bind="foreach: tags">
```

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

```
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:

```
<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 lesson 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 series, 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 lesson digs into the fun part of Knockout.js: appearance bindings.

This lesson represents a chapter from *Knockout Succinctly*, a free eBook from the team at Syncfusion.

Tags: knockout

By Ryan Hodson

Ryan Hodson has worked in a diverse collection of programming fields, building everything from websites to e-publishing platforms, touch-screen thermostats, and natural language processing tools. These experiences have led to a love of exploring new software and a proficiency in several languages and frameworks.In 2012, Ryan founded an independent publishing firm called RyPress and published his first book, Ry's Friendly Guide to Git. Since then, he has worked as a freelance technical writer for well-known software companies, including Syncfusion and Atlassian. Ryan continues to publish high-quality software tutorials via RyPress.com.