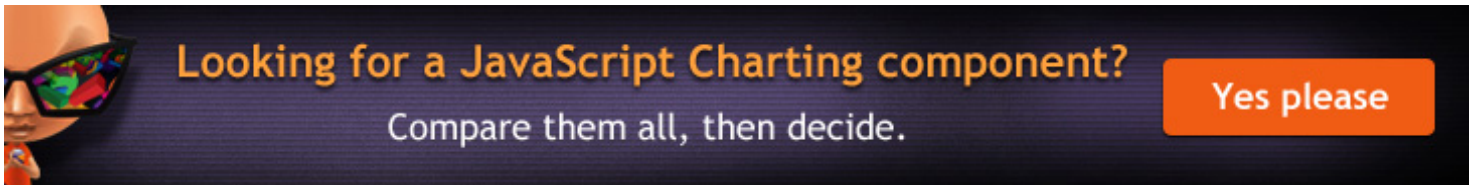


[Advertise Here](#)

# Interactive Bindings

[Ryan Hodson](#) on Jan 12th 2013 with [1 Comment](#)

## Tutorial Details

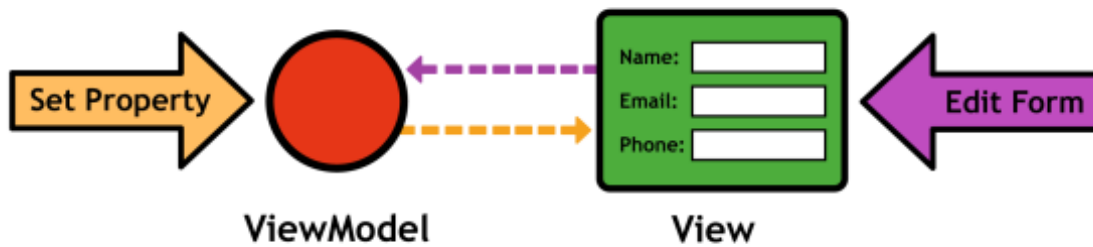
- 
- **Difficulty:** Intermediate
- **Completion Time:** 30 Minutes

[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 7 of 9 in the [Knockout Succinctly](#) Session – [Show All](#)*

[« Previous](#)[Next »](#)

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 lesson, 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 lesson 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:

```
1  <html lang='en'>
2  <head>
3    <title>Interactive Bindings</title>
4    <meta charset='utf-8' />
5    <link rel='stylesheet' href='../style.css' />
6  </head>
7  <body>
8    <h2>
9
10   <form action="#" method="post">
11     <!-- ToDo -->
12   </form>
13
14   <script src='knockout-2.1.0.js'></script>
15   <script>
16     function PersonViewModel() {
17       var self = this;
18       this.firstName = ko.observable("John");
19       this.lastName = ko.observable("Smith");
20     }
21
22     ko.applyBindings(new PersonViewModel());
23   </script>
24 </body>
25 </html>
```

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

---

## 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:

```
1 | <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:

```
1 | this.saveUserData = function(model, event) {  
2 |     alert(model.firstName() + " is trying to checkout!");  
3 |     if (event.ctrlKey) {  
4 |         alert("He was holding down the Control key for some reason.");  
5 |     }  
6 | };
```

In this particular example, `model` refers to the top-level ViewModel instance, and `event` is the DOM event triggered 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 [Lesson 3](#).

---

## The value Binding

The value binding is very similar to the text binding we've been using throughout this series. 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>`):

```
1 | <p>First name: <input data-bind='value: firstName' /></p>  
2 | <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:

```
1 <p>
2   <button data-bind='click: displayName'>
3     Display Name
4   </button>
5   <button data-bind='click: setName'>
6     Set Name
7   </button>
8 </p>
```

The handler methods should look something like the following:

```
1 this.displayName = function() {
2   alert(this.firstName());
3 };
4 this.setName = function() {
5   this.firstName("Bob");
6 };
```

Clicking **Display Name** will read the ViewModel's `email` 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:

```
1 | <p data-bind='event: {mouseover: showDetails, mouseout: hideDetails'
2 |   First name: <input data-bind='value: firstName' />
3 | </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:

```
1 | this.showDetails = function(target, event) {
2 |   alert("Mouse over");
3 | };
4 | this.hideDetails = function(target, event) {
5 |   alert("Mouse out");
6 | };
```

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`:

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


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

```
1 | this.showDetails = function(target, event) {
2 |   this.details(true);
```

```
3   };  
4   this.hideDetails = function(target, event) {  
5       this.details(false);  
6   };
```

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

```
1   <p data-bind='event: {mouseover: showDetails, mouseout: hideDetails'  
2       First name: <input data-bind='value: firstName' />  
3       <span data-bind='visible: details'>Your given name</span>  
4   </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:

```
1  this.details = ko.observable("");
2
3  this.showDetails = function(target, event, details) {
4      this.details(details);
5  }
6  this.hideDetails = function(target, event) {
7      this.details("");
8  }
```

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():

```
1  <p data-bind='event: {mouseover: function(data, event) {
2      showDetails(data, event, "firstName")
3      }, 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:

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

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:

```
1  <p data-bind='event: {mouseover: function(data, event) {
2      showDetails(data, event, "lastName")
3      }, mouseout: hideDetails}'>
4
5  Last name: <input data-bind='value: lastName' />
6  <span data-bind='visible: details() == "lastName"'>Your surname</sp
```



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 [Lesson 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`:

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

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

```
1 | <p>
2 |   Primary phone: <input data-bind='value: primaryPhone' />
3 | </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:

```
1 | <p>  
2 |   Secondary phone: <input data-bind='value: secondaryPhone,  
3 |                       enable: primaryPhone' />  
4 | </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:

```
1 | <p>Annoy me with special offers: <input data-bind='checked: annoyMe
```



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:

```
1 | 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:



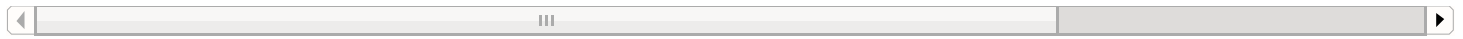
Figure 23: Connecting an observable array with multiple check boxes

For instance, consider the following observable:

```
1 | 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:

```
1 | <p>Annoy me with special offers: <input data-bind='checked: annoyMe  
2 | <div data-bind='visible: annoyMe'>  
3 |   <div>  
4 |     <input data-bind='checked: annoyTimes'  
5 |       value='morning'  
6 |       type='checkbox' />
```



This uses the `annoyMe` property from the previous lesson 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:

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

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

```
1 | <input data-bind='checked: annoyTimes'  
2 |           value='morning'  
3 |           type='radio'  
4 |           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:

```
1 | this.annoyTimes = ko.observableArray([
2 |   'In the morning',
3 |   'In the afternoon',
4 |   'In the evening'
5 | ]);
```

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

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

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 lesson:

```
1 | <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:

```
1 | 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 lesson:

```
1 | this.products = ko.observableArray([  
2 |   {name: 'Beer', price: 10.99},  
3 |   {name: 'Brats', price: 7.99},  
4 |   {name: 'Buns', price: 2.99}  
5 | ]);
```

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:

```
1 | <select data-bind='options: products,  
2 |         optionsText: "name",  
3 |         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:

```
1 <select data-bind='options: products,  
2         optionsText: "name",  
3         selectedOptions: favoriteProducts'  
4         size='3'  
5         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:

```
1 var brats = {name: 'Brats', price: 7.99};  
2 this.products = ko.observableArray([  
3     {name: 'Beer', price: 10.99},  
4     brats,  
5     {name: 'Buns', price: 2.99}  
6 ]);  
7 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:

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

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

```
1 | 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 lesson 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 lesson 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 lesson addresses this issue by integrating Knockout.js with jQuery's AJAX functionality.

This lesson represents a chapter from [Knockout Succinctly](#), a free eBook from the team at [Syncfusion](#).

Like

One person likes this. Be the first of your friends.





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](#).

**Note:** Want to add some source code? Type `<pre><code>` before it and `</code></pre>` after it. [Find out more](#)