

[Advertise Here](#)

# Appearance Bindings

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

[« Previous](#)[Next »](#)

In the previous lesson, 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 lesson, 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:

```
1 | <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:

```
1 function Product(name, price, tags, discount) {  
2  
3     this.formattedName = ko.computed(function() {  
4         return "<strong>" + this.name() + "</strong>";  
5     }, this);  
6 }
```

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

```
1 <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:

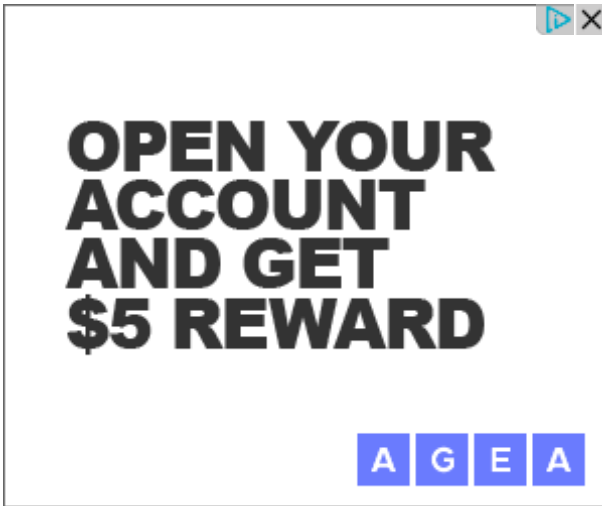
```
1 | <td data-bind='visible: discount() > 0' style='color: red'>
```

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:

```
1 | <!-- Using if binding: -->
2 | <td data-bind="if: discount() > 0" style="color: red"></td>
3 |
4 | <!-- Using visible binding: -->
5 | <td data-bind='visible: discount() > 0' style='color: red; display:
6 |   You saved <span data-bind='text: formattedDiscount'></span>!!!
7 | </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:

```
1 <td data-bind='if: discount() > 0' style='color: red'>
2   You saved <span data-bind='text: formattedDiscount,
3                     css: {supersaver: discount() > .15}'></span>!!!
4 </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).

```
1 .supersaver {
2   font-size: 1.2em;
3   font-weight: bold;
4 }
```

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:

```
1 | You saved <span data-bind='text: formattedDiscount,  
2 | style: {fontWeight: discount() > .15 ? "bold"
```

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

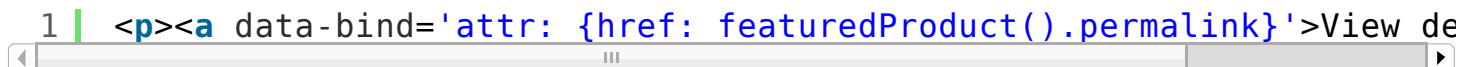
```
1 | <td style='color: red; font-weight: bold'>
```

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:

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



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 lesson 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).

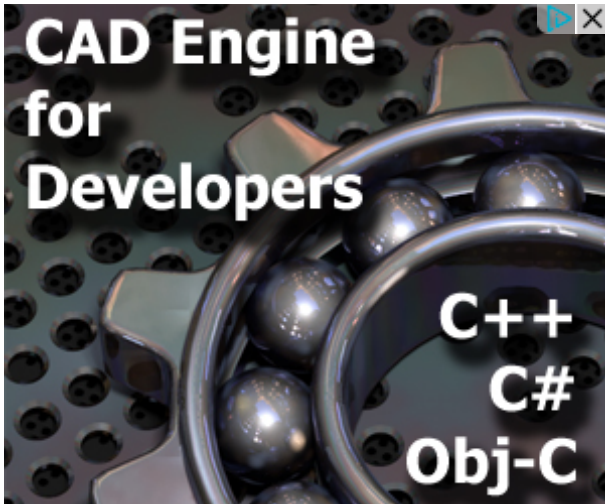
The appearance bindings presented in this lesson provide all the tools you need to *display* your data, but they don't let us add any user interaction to our view components. In the next lesson, we'll take a look at how Knockout.js manages form

fields.

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

Like

Be the first of your friends to like this.



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](#)