FREE eBook

LEARNING knockout.js

Free unaffiliated eBook created from
Stack Overflow contributors.

#knockout.j

s

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: knockout-js

It is an unofficial and free knockout.js ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official knockout.js.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with knockout.js

Remarks

This section provides an overview of what knockout.js is, and why a developer might want to use it.

It should also mention any large subjects within knockout.js, and link out to the related topics. Since the Documentation for knockout.js is new, you may need to create initial versions of those related topics.

Versions

Version	Notes	Release Date
3.4.2	Bug fixes	2017-03-06
3.4.1	Bug fixes	2016-11-08
3.4.0		2015-11-17
3.3.0		2015-02-18
3.2.0	Introduced `component` binding	2014-08-12
3.1.0		2014-05-14
3.0.0	See also: upgrade (from 2.x) notes	2013-10-25
2.3.0	Last 2.x release	2013-07-08
2.0.0		2011-12-21
1.2.1	Last 1.x release	2011-05-22
1.0.0		2010-07-05

Examples

Installation or Setup

Knockout is available on most JavaScript platforms, or as a standalone script.

Include as a script

You can download the script from it's download page, then include it in your page with a standard `script tag`

```
<script type='text/javascript' src='knockout-3.4.0.js'></script>
```

Using a CDN

You can also include knockout from either the Microsoft CDN, or CDNJS

```
<script type='text/javascript' src='//ajax.aspnetcdn.com/ajax/knockout/knockout-
3.4.0.js'></script>
```

OR

```
<script type='text/javascript' src='//cdnjs.cloudflare.com/ajax/libs/knockout/3.4.0/knockout-
min.js'></script>
```

Install from npm

```
npm install knockout
```

optionally you can add the `--save` parameter to keep it in your `package.json` file

Install from bower

```
bower install knockout
```

optionally you can add the `--save` parameter to keep it in your `bower.json` file

Install from NuGet

```
Install-Package knockoutjs
```

Getting Started: Hello world!

Creating an HTML document and enabling knockout.js

Create an HTML file and include `knockout.js` via a `<script>` tag.

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title>Hello world! (knockout.js)</title>
</head>
<body>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/knockout/3.4.0/knockout-
debug.js"></script>
</body>
</html>
```

Add a second `<script>` tag *under* the knockout script. In this script tag, we'll initialize a view model and apply *data binds* to our Document.

```
<script>
  var ViewModel = function() {
    this.greeting = ko.observable("Hello");
    this.name = ko.observable("World");

    this.appHeading = ko.pureComputed(function() {
      return this.greeting() + ", " + this.name() + "!";
    }, this);
  };

  var appVM = new ViewModel();

  ko.applyBindings(appVM);
</script>
```

Now, continue creating a *view* by adding some HTML to the body:

```
<section>
  <h1 data-bind="text: appHeading"></h1>
  <p>Greeting: <input data-bind="textInput: greeting" /></p>
  <p>Name: <input data-bind="textInput: name" /></p>
</section>
```

When the HTML document is opened and the scripts are executed, you'll see a page that says **Hello, World!**. When you change the words in the `<input>` elements, the `<h1>` text is automatically updated.

How the created file works

1. A debug version of knockout is loaded from an external source (cdnjs)

Code:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/knockout/3.4.0/knockout-
debug.js"></script>
```

2. A viewmodel instance is created that has *observable* properties. This means knockout is able to detect changes and update the UI accordingly.

Code:

```
var appVM = new ViewModel();
```

3. Knockout checks the DOM for `data-bind` attributes and updates the UI using the provided viewmodel.

Code:

```
ko.applyBindings(appVM);
```

4. When it encounters a `text` binding, knockout uses the property's value as it is defined in the bound viewmodel to inject a text node:

Code:

```
<h1 data-bind="text: appHeading"></h1>
```

Computed Observables

Computed observables are functions that can "watch" or "react" to other observables on the view model. The following example shows how you would display the total number of users and the average age.

*Note: The example below can also make use of **pureComputed()** (introduced in v3.2.0) since the function simply calculates something based on other view model properties and returns a value.*

```
<div>
  Total Users: <span data-bind="text: TotalUsers">2</span><br>
  Average Age: <span data-bind="text: UsersAverageAge">32</span>
</div>
```

```
var viewModel = function() {

    var self = this;

    this.Users = ko.observableArray([
        { Name: "John Doe", Age: 30 },
        { Name: "Jane Doe", Age: 34 }
    ]);

    this.TotalUsers = ko.computed(function() {
        return self.Users().length;
    });

    this.UsersAverageAge = ko.computed(function() {
        var totalAge = 0;
        self.Users().forEach(function(user) {
            totalAge += user.Age;
        });

        return totalAge / self.TotalUsers();
    });
};
```

```
ko.applyBindings(viewModel);
```

Read Getting started with knockout.js online: https://riptutorial.com/knockout-js/topic/799/getting-started-with-knockout-js

Chapter 2: AJAX requests and binding

Examples

Sample AJAX request w/ binding

Page.html

```
<div data-bind="foreach: blogs">
    <br />
    <span data-bind="text: entryPostedDate"></span>
    <br />
    <h3>
        <a data-bind="attr: { href: blogEntryLink }, text: title"></a>
    </h3>
    <br /><br />
    <span data-bind="html: body"></span>
    <br />
    <hr />
    <br />
</div>

<!--- include knockout and dependencies (Jquery) --->
<script type="text/javascript" src="blog.js"></script>
```

blog.js

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

    // Properties
    self.blogs = ko.observableArray([]);

    // consists of entryPostedDate, blogEntryLink, title, body
    var blogApi = "/api/blog";

    // Load data
    $.getJSON(blogApi)
        .success(function (data) {
            self.blogs(data);
        });
    }
ko.applyBindings(new vm());
```

Note that JQuery was used (`$.getJSON(...)`) to perform the request. vanilla JavaScript can perform the same, albeit with more code.

"Loading section/notification" during AJAX request

Blog.html

```
<div data-bind="visible: isLoading()">
```

```
    Loading...
</div>

<div data-bind="visible: !isLoading(), foreach: blogs">
    <br />
    <span data-bind="text: entryPostedDate"></span>
    <br />
    <h3>
        <a data-bind="attr: { href: blogEntryLink }, text: title"></a>
    </h3>
    <br /><br />
    <span data-bind="html: body"></span>
    <br />
    <hr />
    <br />
</div>

<!--- include knockout and dependencies (JQuery) --->
<script type="text/javascript" src="blog.js"></script>
```

blog.js

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

    // Properties
    self.blogs = ko.observableArray([]);
    self.isLoading = ko.observable(true);

    // consists of entryPostedDate, blogEntryLink, title, body
    var blogApi = "/api/blog";

    // Load data
    $.getJSON(blogApi)
        .success(function (data) {
            self.blogs(data);
        })
        .complete(function () {
            self.isLoading(false); // on complete, set loading to false, which will hide
"Loading..." and show the content.
        });
}
ko.applyBindings(new vm());
```

Note that JQuery was used ($.getJSON(...)) to perform the request. vanilla JavaScript can perform the same, albeit with more code.

Read AJAX requests and binding online: https://riptutorial.com/knockout-js/topic/7538/ajax-requests-and-binding

Chapter 3: Bindings

Syntax

- `<!-- ko if:myObservable --><!-- /ko -->`
- `<i data-bind="if:myObservable"></i>`

Remarks

What a binding is

Essentially a binding or a data binding is a way to link your ViewModels to your Views(templates) and vice versa. KnockoutJS uses two-way data binding, which means changes to your ViewModel influence the View and changes to your View can influence the ViewModel.

Under the hood (short overview)

Bindings are just plugins (scripts) that allow you to solve a particular task. This task is more often than not is changing markup (html) according to your ViewModel.

For example, a `text` binding allows you to display text and dynamically change it whenever your ViewModel changes.

KnockoutJS comes with many powerful bindings and lets you extend it with your own custom bindings.

And most importantly bindings are not magical, they work according to a set of rules and anytime you are unsure of what a binding does, what parameters it takes or when it will update the view you can refer to source code of the binding.

Consider the following example of a custom binding:

```
ko.bindingHandlers.debug = {
    init: function (element, valueAccessor, allBindingsAccessor, viewModel, bindingContext) {
        ko.computed(function () {
            var value = ko.unwrap(valueAccessor());

            console.log({
                value: value,
                viewModel: viewModel,
                bindingContext: bindingContext
            });
        }, null, { disposeWhenNodeIsRemoved: element });
    }
};
```

0. A binding has a name - `debug` so you can use as follows:

```
data-bind="debug: 'foo'"
```

1. The `init` method is called once when the binding is initiated. The rest of the updates are handled by a anonymous computed which is disposed when `element` is removed.
2. The binding prints to console several things: the passed value in our example this value is `foo` (this value can also be observable since `ko.unwrap` method is used to read it), the current viewModel and bindingContext.
3. Whenever the passed value changes the binding will print updated information to console.
4. This binding cannot be used with virtual elements (in html comments), only on real elements, since `ko.virtualElements.allowedBindings.debug` flag is not set to true.

When to use parentheses

Without any additional plugins, KnockoutJS will only have live View updates for properties on the ViewModel that are *observable* (regular `observable`, but also `computed`, `pureComputed`, `observableArray`, etc). An observable would be created like this:

```
var vm = { name: ko.observable("John") };
```

In this case, `vm.name` is a *function* with two seperate "modes":

1. Getter: `vm.name()`, without arguments, will get the current value;
2. Setter: `vm.name("Johnnyboy")`, with an argument, will set a new value.

In the built-in data-bindings you can *always* use the getter form, and you can *sometimes* actually *omit* the parentheses, and the binding will effectively "add" them for you. So these are equivalent:

```
<p data-bind="text: name"></p> ... will work
<p data-bind="text: name()"></p> ... works too
```

But this will fail:

```
<p data-bind="text: 'Hello, ' + name + '!'"></p> ... FAILS!
```

Because as soon as you want to "do" something before passing a value to a data-binding, including value comparisons, you need to properly "get" the values for all observables, e.g.:

```
<p data-bind="text: 'Hello, ' + name() + '!'"></p> ... works
```

See also this Q&A for more details.

Examples

If / ifnot

You can use the `if` binding to determine whether or not the child elements of the node should be

created.

```
<div class="product-info">
    <h2>  Product1  </h2>
    <img src="../products/product1.jpg"/>
    <span data-bind="if:featured">
        <span class="featured"></span>
    </span>
    <span data-bind="ifnot:inStock">
        <span class="out-of-stock"></span>
    </span>
</div>

<script>
    ko.applyBindings({
        product: {
            featured: ko.observable(true),
            inStock: ko.observable(false)
        }
    });
</script>
```

The inverse of the `if` binding is `ifnot`

```
<div data-bind="ifnot: someProperty">...</div>
```

is equivalent to

```
<div data-bind="if: !someProperty()">...</div>
```

Sometimes, you'll wan't to control the presence of elements without having to create a container (typically for `<li>` elements in a `<ul>` or `<option>` elements inside a `<select>`)

Knockout enables this with containerless control flow syntax based on comment tags like so:

```
<select>
  <option value="0">fixed option</option>
<!-- ko if: featured-->
  <option value="1">featured option</option>
<!-- /ko -->
</select>
```

Foreach

Similar to repeaters used in other languages. This binding will allow you to replicate a block of html for each item in an array.

```
<div data-bind="foreach:contacts">
    <div class="contact">
        <h2 data-bind="text:name">
        <p data-bind="text:info">
    </div>
</div>
```

```
<script type="text/javascript">
    var contactViewModel = function (data) {
        this.name = ko.observable(data.name);
        this.info = ko.observable(data.info);
    };

    ko.applyBindings({
        contacts: [
            new contactViewModel({name:'Erik', info:'Erik@gmail.com'}),
            new contactViewModel({name:'Audrey', info:'Audrey@gmail.com'})
        ]
    });
</script>
```

Notice that when we are looping through our context becomes the item within the array, in this case an instance of the `contactViewModel`. Within a `foreach` we also have access to

- `$parent` - the view model that created this binding
- `$root` - the root view model (could also be parent)
- `$data` - the data at this index of the array
- `$index` - the (observable) zero-based index of the rendered item

With

The `with` binding binds the HTML inside the bound node to a separate context:

```
<div data-bind="with: subViewModel">
  <p data-bind="text: title"></p>
</div>
```

The `with` binding may also be used without a container element where a container element may not be appropriate.

```
<!-- ko with: subViewModel -->
  <p data-bind="text: title"></p>
<!-- /ko -->
```

```
var vm = {
  subViewModel: ko.observable()
};

// Doesn't throw an error on the `text: title`; the `<p>` element
// isn't bound to any context (and even removed from the DOM)
ko.applyBindings(vm);

// Includes the `<p>` element and binds it to our new object
vm.subViewModel({ title: "SubViewModel" });
```

The `with` binding has many similarities to the `template` or `foreach` bindings.

Visible

The `visible` binding will hide an element by applying `style="display: none;"` to it when the binding evaluate as falsey.

```
<input type="text" data-bind="textInput: name"> <span class="error" data-bind="visible:
isInvalid">Required!</span>

ko.applyBindings(new ViewModel());

function ViewModel(){
    var vm = this;
  vm.name = ko.observable("test");
  vm.isInvalid = ko.computed(function() {
      return vm.name().length == 0;
  });
}
```

jsFiddle

Read Bindings online: https://riptutorial.com/knockout-js/topic/2249/bindings

Chapter 4: Bindings - Form fields

Examples

Click

The `click` binding can be used with any visible DOM element to add an event handler, that will invoke a JavaScript function, when element is clicked.

```
<button data-bind="click: onClick">Click me</button>
```

```
ko.applyBindings({
  onClick: function(data, event) {
    // data: the context of the element that triggered the event
    console.log(data);

    // event: the click event
    console.log(event);
  }
});
```

Options

Use this binding to build options for a select item

```
<select data-bind="options: gasGiants"></select>

<script type="text/javascript">
    var viewModel = {
        gasGiants: ko.observableArray(['Jupiter', 'Saturn', 'Neptune', 'Uranus'])
    };
</script>
```

You can also use properties inside the array for displaying in the list and for saving in the viewModel: `optionsText` enables a custom display text

`optionsValue` sets the value property of the corresponding `<option>`

`value` stores the value of the selected option into an observable of the viewModel

```
<select data-bind="options: gasGiants, optionsText:'name', optionsValue:'id',
value:selectedPlanetId"></select>

<script type="text/javascript">
    var viewModel = {
        selectedPlanetId: ko.observable(),
        gasGiants: ko.observableArray([{name:'Jupiter', id:'0'},
                                       {name:'Saturn', id:'1'},
                                       {name:'Neptune', id:'2'},
                                       {name:'Uranus', id:'3'}])
    };
```

```
</script>
```

To store the results of a multi-select list, the options binding can be combined with the `selectedOptions` binding.

```
<select data-bind="options: gasGiants, selectedOptions: chosenGasGiants"
multiple="true"></select>

<script type="text/javascript">
    var viewModel = {
        gasGiants: ko.observableArray(['Jupiter', 'Saturn', 'Neptune', 'Uranus'])
        chosenGasGiants: ko.observableArray(['Jupiter','Saturn']) // Initial selection
    }; </script>
```

disabled / enabled

The disabled binding adds a `disabled` attribute to a html element causing it to no longer be editable or clickable. This is useful mainly for `<input>`, `<select>`, `<textarea>`, `<a>` and `<button>` elements

```
<input data-bind="disabled: disableInput"/>

<script type="text/javascript">
var viewModel = {
    disableInput: ko.observable(true);
};
</script>
```

The inverse of the `disabled` binding is `enabled`

The visibility can also be calculated using JavaScript functions. Any observables used in this functions have to called with parentheses

```
<script type="text/javascript">
var viewModel = {
    disableInput: ko.observable(true);
};
</script>
```

or

```
<input data-bind="disabled: allValues().length>4"/>

<script type="text/javascript">
var viewModel = {
    allValues: ko.observableArray([1,2,3,4,5]);
};
</script>
```

submit

Event handler to be invoked when a DOM element is submitted.

```
<form data-bind="submit: doSomething">
    <!-- form content here -->
    <button type="submit"></button>
</form>

<script type="text/javascript">
    var vm = {
        doSomething: function(data){
            //do something here
        };
    }
</script>
```

Knockout will prevent the browser's default submit action for that form. If you want your form to be submitted like a normal HTML form, you just return `true` in the submit handler.

Value

Use the value binding to obtain the value of an element. The value binding can be applied to any form control, however there are other bindings that may be better suited for checkboxes, radio buttons, and text inputs.

The following example illustrates how to apply the binding element to several form input fields, and how to populate default values:

ViewModel definition:

```
var MyViewModel = function(){
    var self = this;
  //Initialize valueOne
  self.valueOne = ko.observable();
  //Initialize valueTwo with a default value of "Value two"
  self.valueTwo = ko.observable("Value two");
  //Initialize the color dropdown, and by default, select the "blue" option
  self.color = ko.observable("blue");

  self.valueOne.subscribe(function(newValue){
      console.log("valueOne: " + newValue);
  });

  self.valueTwo.subscribe(function(newValue){
      console.log("valueTwo: " + newValue);
  });

  self.color.subscribe(function(newValue){
      console.log("color: " + newValue);
  });
}
```

Associated markup:

```
<input type="text" data-bind="value: valueOne" />
<input type="text" data-bind="value: valueTwo" />

<select data-bind="value: color">
```

```
  <option value="red">Red</option>
  <option value="green">Green</option>
  <option value="blue">Blue</option>
</select>
```

In the above example, when a value changes, the new value will be logged to the console. The initial values will not trigger a change event.

By default, the value binding defines a change as a change to the elements value, and focus being transferred to another element. This can be altered using the valueUpdate option:

```
<input type="text" data-bind="value: valueOne, valueUpdate: 'keyup'" />
```

The above example will change the value update to trigger on key up. Available options are input, keyup, keypress, and afterkeydown.

Read Bindings - Form fields online: https://riptutorial.com/knockout-js/topic/7101/bindings---form-fields

Chapter 5: Bindings - Text and appearance

Examples

Text

The `text` binding can be used with any element to update it's innerText.

```
<p>
  <span data-bind="text: greeting"></span>,
  <span data-bind="text: subject"></span>.
</p>
```

```
ko.applyBindings({
  greeting: ko.observable("Hello"),
  subject: ko.observable("world")
});
```

The `text` binding may also be used with virtual elements.

```
<p>
  <!--ko text: greeting--><!--/ko-->,
  <!--ko text: subject--><!--/ko-->.
</p>
```

CSS

This binding will apply the supplied CSS class to the element. Static classes are applied when the given conditions are loosely-evaluated to true. Dynamic classes use the value of an observable or computed.

page.html

```
<p data-bind="css: { danger: isInDanger }">Checks external expression</p>
<p data-bind="css: { danger: dangerLevel() > 10 }">Expression can be inline</p>
<p data-bind="css: { danger: isInDanger, glow: shouldGlow }">Multiple classes</p>
<p data-bind="css: dynamicObservable">Dynamic CSS class from observable</p>
<p data-bind="css: dynamicComputed">Dynamic CSS class from computed</p>
```

page.js

```
ko.applyBindings({
  isInDanger: ko.observable(true),
  dangerLevel: ko.observable(5),
  isHot: ko.observable(true),
  shouldGlow: ko.observable(true),
  dynamicObservable: ko.observable('highlighted'),
  dynamicComputed: ko.computed(function() {
        var customClass = "";
        if(dangerLevel() >= 15 ) {
```

```
            customClass += " danger";
        }
        if(dangerLevel() >= 10) {
            customClass += " glow";
        }
        if(dangerLevel() >= 5) {
            customClass += " highlighted";
        }
        return customClass;
    });
});
```

page.css

```
.danger { background: red; }
.glow { box-shadow: 5px 5px 5px gold; }
.highlighted { color: purple; }
```

See also: official documentation.

Visible

Can be used to show/hide DOM elements. Similar to using `if`, except that `visible` will still build the element and set `display:none`.

```
<div data-bind="visible: shouldShowMessage">
    You will see this message only when "shouldShowMessage" holds a true value.
</div>

<script type="text/javascript">
    var viewModel = {
        shouldShowMessage: ko.observable(true);
    };
</script>
```

Attr

Use the `attr` binding to apply any additional attributes to your element. Most commonly used for setting an href, src, or any data-attributes.

```
<img data-bind="attr: { src: url, title: title }"/>
```

```
var viewModel = {
    url: ko.observable("images/example.png"),
    title: "example title"
};
```

HTML

This binding updates the innerHTML of the element using `jQuery.html()`, if jQuery has been referenced, otherwise, KO's own parsing logic. It can be useful if retrieving HTML from an API, RSS feed, etc. Be mindful of using this tag with user input HTML.

page.html

```
<p>
  <span data-bind="html: demoLink"></span>
</p>
```

page.js

```
ko.applyBindings({
  demoLink: ko.observable("<a href='#'>Make a link</a>")
});
```

Read Bindings - Text and appearance online: https://riptutorial.com/knockout-js/topic/7103/bindings---text-and-appearance

Chapter 6: Components introduction

Remarks

Components allow reusable controls/widgets represented by their own view (template) and viewmodel. They were added in Knockout 3.2. Inspired by WebComponents, Knockout allows Components to be defined as Custom Elements, allowing the use of more self-explanatory markup.

Examples

Progress bar (Boostrap)

Component definition

```
ko.components.register('progress-bar', {
    viewModel: function(params) {
            var that = this;

        // progress is a numeric value between 0 and 100
        that.progress = params.progress;


        that.progressPercentual = ko.computed(function(){
            return '' + ko.utils.unwrapObservable(that.progress) + '%';
        })
    },
    template:
        '<div class="progress"> <div data-bind="attr:{\'aria-valuenow\':progress},
style:{width:progressPercentual}, text:progressPercentual" class="progress-bar"
role="progressbar" aria-valuenow="0" aria-valuemin="0" aria-valuemax="100" style="min-width:
2em;"></div> </div>'
});
```

Html usage

```
<progress-bar params="progress:5"></progress-bar>
```

Read Components introduction online: https://riptutorial.com/knockout-js/topic/6207/components-introduction

Chapter 7: Custom Bindings

Examples

Binding Registration

Cusom bindings should be registered by extending the current knockout bindingHandlers object. This is done by adding a new property to the object.

```
ko.bindingHandlers.newBinding = {
    init: function(element, valueAccessor, allBindings, viewModel, bindingContext) {
    },
    update: function(element, valueAccessor, allBindings, viewModel, bindingContext) {
    }
};
```

Custom fade in/fade out visibility binding

This example implements a custom binding that toggles visibility (similar to the existing visible binding), but will utilize jQuery's fading API to animate the transition from visible to invisible.

Custom binding definition:

```
//Add a custom binding called "fadeVisible" by adding it as a property of ko.bindingHandlers
ko.bindingHandlers.fadeVisible = {
  //On initialization, check to see if bound element should be hidden by default
  'init': function(element, valueAccessor, allBindings, viewModel, bindingContext){
    var show = ko.utils.unwrapObservable(valueAccessor());
    if(!show){
      element.style.display = 'none';
    }
  },
  //On update, see if fade in/fade out should be triggered. Factor in current visibility
  'update': function(element, valueAccessor, allBindings, viewModel, bindingContext) {
    var show = ko.utils.unwrapObservable(valueAccessor());
    var isVisible = !(element.style.display == "none");

    if (show && !isVisible){
      $(element).fadeIn(750);
    }else if(!show && isVisible){
      $(element).fadeOut(750);
    }
  }
}
```

Sample markup with the fadeVisible binding:

```
<div data-bind="fadeVisible: showHidden()">
  Field 1: <input type="text" name="value1" />
  <br />
  Field 2: <input type="text" name="value2" />
</div>
```

```
<input data-bind="checked: showHidden" type="checkbox"/> Show hidden
```

Sample view model:

```
var ViewModel = function(){
    var self = this;
    self.showHidden = ko.observable(false);
}


ko.applyBindings(new ViewModel());
```

Custom text replace binding

This example is a custom binding that replaces text whenever an input value is updated. In this case, spaces will be replaced with "+". It is intended to be used alongside the existing value binding, and shows binding with an object literal.

Sample markup with the replaceText binding:

```
<input type="text" data-bind="value: myField, replaceText: {value: myField, find:' ',
replace:'+'}" />
```

Custom binding definition:

```
ko.bindingHandlers.replaceText = {

   //On update, grab the current value and replace text
  'update': function(element, valueAccessor, allBindings, viewModel, bindingContext) {

      //Get the current value of the input
      var val = ko.utils.unwrapObservable(valueAccessor().value());

      //Replace text using passed in values obtained from valueAccessor()
      //Note - Consider using something like string.js to do the find and replace
      var replacedValue = val.split(valueAccessor().find).join(valueAccessor().replace);

      //Set new value
      valueAccessor().value(replacedValue);
  }
}
```

Sample view model:

```
var ViewModel = function(){
  var self = this;
  self.myField = ko.observable("this is a simple test");
}


ko.applyBindings(new ViewModel());
```

Replace with regular expression custom binding

Custom binding definition

```
function regExReplace(element, valueAccessor, allBindingsAccessor, viewModel, bindingContext)
{

  var observable = valueAccessor();
  var textToReplace = allBindingsAccessor().textToReplace || '';
  var pattern = allBindingsAccessor().pattern || '';
  var flags = allBindingsAccessor().flags;
  var text = ko.utils.unwrapObservable(valueAccessor());
  if (!text) return;
  var textReplaced = text.replace(new RegExp(pattern, flags), textToReplace);

  observable(textReplaced);
}

ko.bindingHandlers.regExReplace = {
  init: regExReplace,
  update: regExReplace
}
```

Usage

ViewModel

```
ko.applyBindings({
  name: ko.observable(),
  num: ko.observable()
});
```

View

```
<input type="text" data-bind="textInput : name, regExReplace:name, pattern:'(^[^a-zA-
Z]*)|(\\W)',flags:'g'" placeholder="Enter a valid name" />
<span data-bind="text : name"></span>
<br/>
<input class=" form-control " type="text " data-bind="textInput : num, regExReplace:num,
pattern: '[^0-9]',flags: 'g' " placeholder="Enter a number " />
<span data-bind="text : num"></span>
```

Read Custom Bindings online: https://riptutorial.com/knockout-js/topic/6332/custom-bindings

Chapter 8: Debugging a knockout.js application

Examples

Checking the binding context of a DOM element

Many bugs in knockout data binds are caused by undefined properties in a viewmodel. Knockout has two handy methods to retrieve the binding context of an HTML element:

```
// Returns the binding context to which an HTMLElement is bound
ko.contextFor(element);

// Returns the viewmodel to which an HTMLElement is bound
// similar to: ko.contextFor(element).$data
ko.dataFor(element);
```

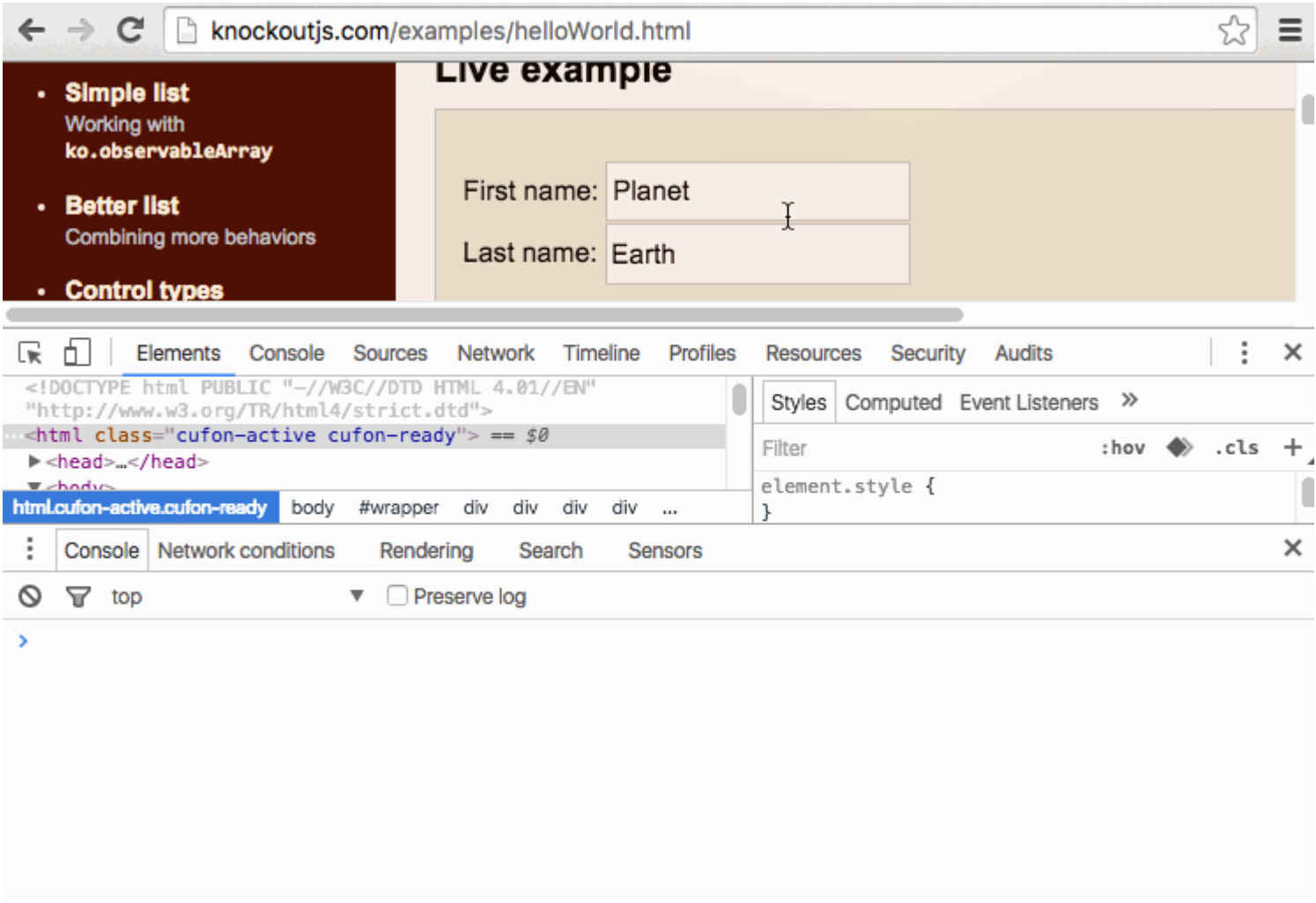
To quickly find out the binding context of a UI element, here's a handy trick:

Most modern browsers store the currently selected DOM element in a global variable: `$0` (more about this mechanism)

- Right click an element in your UI and choose *"inspect"* or *"inspect element"* in the context menu.
- type `ko.dataFor($0)` in the developer console and press enter

Browser plugins also exist which may assist with finding the object context.

An example (try it on Knockout hello world example):



Printing a binding context from markup

Sometimes it is useful to print a current binding directly from markup. A neat trick which allows that is to use an additional DOM element with a non-existing binding (KO < 3.0), custom binding or a binding that is not relevant such as `uniqueName`.

Consider this example:

```
<tbody data-bind="foreach: people">
    <tr>
         <td data-bind="text: firstName"></td>
         <td data-bind="text: lastName"></td>
    </tr>
</tbody>
```

If one would like to to find out the binding context of each element in the people array, one can write:

```
<tbody data-bind="foreach: people">
    <span data-bind="uniqueName: console.log($data)"></span>
    <tr>
        <td data-bind="text: firstName"></td>
        <td data-bind="text: lastName"></td>
    </tr>
```

```
</tbody>
```

Read Debugging a knockout.js application online: https://riptutorial.com/knockout-js/topic/5066/debugging-a-knockout-js-application

Chapter 9: Equivalents of AngularJS bindings

Remarks

Not everything in AngularJS has a KnockoutJS equivalent (for example `ngCloack`, or `ngSrc`). There are two main solutions typically available:

1. Use the generic `attr` or `event` binding instead.
2. Similar to custom directives in AngularJS, you can write your own custom binding handler if you need something that isn't included in the base library.

If you prefer the AngularJS binding syntax you can consider using Knockout.Punches which enables handlebar-style binding.

Examples

ngShow

AngularJS code for dynamically showing/hiding an element:

```
<p ng-show="SomeScopeProperty">This is conditionally shown.</p>
```

KnockoutJS equivalent:

```
<p data-bind="visible: SomeScopeObservable">This is conditionally shown.</p>
```

ngBind (curly markup)

AngularJS code for rendering plain text:

```
<p>{{ ScopePropertyX }} and {{ ScopePropertyY }}</p>
```

KnockoutJS equivalent:

```
<p>
  <!-- ko text: ScopeObservableX --><!-- /ko -->
  and
  <!-- ko text: ScopeObservableY --><!-- /ko -->
</p>
```

or:

```
<p>
  <span data-bind="text: ScopeObservableX"></span>
  and
  <span data-bind="text: ScopeObservableY"></span>
```

```
</p>
```

ngModel on input[type=text]

AngularJS code for two-way binding on a text `input`:

```
<input ng-model="ScopePropertyX" type="text" />
```

KnockoutJS equivalent:

```
<input data-bind="textInput: ScopeObservableX" type="text" />
```

ngHide

There is no *direct* equivalent binding in KnockoutJS. However, since hiding is just the opposite of showing, we can just invert the example for Knockout's `ngShow` equivalent.

```
<p ng-hide="SomeScopeProperty">This is conditionally shown.</p>
```

KnockoutJS equivalent:

```
<p data-bind="visible: !SomeScopeObservable()">This is conditionally hidden.</p>
```

The above KnockoutJS example assumes `SomeScopeObservable` is an observable, and because we use it in an expression (because of the `!` operator in front of it) we cannot omit the `()` at the end.

ngClass

AngularJS code for dynamic classes:

```
<p ng-class="{ highlighted: scopeVariableX, 'has-error': scopeVariableY }">Text.</p>
```

KnockoutJS equivalent:

```
<p data-bind="css: { highlighted: scopeObservableX, 'has-error': scopeObservableY }">Text.</p>
```

Read Equivalents of AngularJS bindings online: https://riptutorial.com/knockout-js/topic/2408/equivalents-of-angularjs-bindings

Chapter 10: Href binding

Remarks

There is no `href` binding in the core KnockoutJS library, which is the reason all examples showcase *other* features of the library to get the same effect.

See also this Stack Overflow question on the same topic.

Examples

Using attr binding

```
<a data-bind="attr: { href: myUrl }">link with dynamic href</a>
```

```
ko.applyBindings({
  myUrl: ko.observable("http://www.stackoverflow.com")
});
```

Since there is no native `href` binding in KnockoutJS, you need to use a different feature to get dynamic links. The above example showcases the built-in `attr` binding to get a dynamic link.

Custom binding handler

`href` binding is not native to KnockoutJS, so to get dynamic links use a custom binding handler:

```
<a data-bind="href: myUrl">link with dynamic href</a>
```

```
ko.bindingHandlers['href'] = {
  update: function(element, valueAccessor) {
    element.href = ko.utils.unwrapObservable(valueAccessor());
  }
};
```

Read Href binding online: https://riptutorial.com/knockout-js/topic/6582/href-binding

Chapter 11: Observables

Examples

Creating an observable

JS

```
// data model
var person = {
    name: ko.observable('Jack'),
    age: ko.observable(29)
};

ko.applyBindings(person);
```

HTML

```
<div>
    <p>Name: <input data-bind='value: name' /></p>
    <p>Age: <input data-bind='value: age' /></p>
    <h2>Hello, <span data-bind='text: name'> </span>!</h2>
</div>
```

Explicit Subscription to Observables

```
var person = {
  name: ko.observable('John')
};

console.log(person.name());

console.log('Update name');

person.name.subscribe(function(newValue) {
  console.log("Updated value is " + newValue);
});

person.name('Jane');
```

Read Observables online: https://riptutorial.com/knockout-js/topic/6363/observables

Chapter 12: Working with knockout foreach binding with JSON

Examples

Working with nested looping

Here is the JSON Structure we are going to use.

```
{
  "employees": [
    {
      "firstName": "John",
      "lastName": "Doe",
      "skills": [
        {
          "name": "javascript",
          "rating": 5
        }
      ]
    },
    {
      "firstName": "Anna",
      "lastName": "Smith",
      "skills": [
        {
          "name": "css",
          "rating": 5
        },
        {
          "name": "javascript",
          "rating": 5
        }
      ]
    },
    {
      "firstName": "Peter",
      "lastName": "Jones",
      "skills": [
        {
          "name": "html",
          "rating": 5
        },
        {
          "name": "javascript",
          "rating": 3
        }
      ]
    }
  ]
};
```

This json structure can be assigned to a variable or it can be a response of any api.

As we can see in this JSON there is an outer node employees which holds information about them, and there is an internal node which tells about each employee skills.

so here we are going to create a nested for each using knockout foreach. Here is the html

```
<ul id="employee" data-bind="foreach: employee">
   <li data-bind="text:firstName + ' ' + lastName">
   </li>
   <ul data-bind="foreach : skills">
        <li data-bind="text: name">
        </li>
      <ul>
        <li>
          Rating : <!-- ko text: rating --><!-- /ko -->
        </li>
      </ul>
   </ul>
</ul>
```

Here in the above html there two list that hold foreach loop. outer loop will hold the outer node of the json structure that is employees. Inner loop holds the skills of each employee. Inside each loop we can access the properties of corresponding node. For an example we can access name and rating inside skill loop not from outside.

Below is the javascript code.

```
var employeeViewModel = function(){
  var self = this;
  self.employee = ko.observableArray(employees);//here we can assign json
}
var viewModel = new employeeViewModel();
ko.applyBindings(viewModel);
```

Form the javascript point of view there is not much code. we can directly assign our json to an observablearray which will be used by Html.

Read Working with knockout foreach binding with JSON online: https://riptutorial.com/knockout-js/topic/6961/working-with-knockout-foreach-binding-with-json

Credits

S. No	Chapters	Contributors
1	Getting started with knockout.js	ASindleMouat, aswallows, Community, Jeroen, Michael Best, Ray, Ross Vernal, Tyrsius, user3297291
2	AJAX requests and binding	Kritner
3	Bindings	CreationEdge, dotnetom, Homer, Jeroen, Knekki, Kritner, Matthias Lloyd, natus, Nick DeFazio, Olga, stackoverfloweth, Steffen Nieuwenhoven, tmg, user3297291
4	Bindings - Form fields	Knekki, Matthias Lloyd, Nick DeFazio, stackoverfloweth, Steffen Nieuwenhoven, tmg, user3297291
5	Bindings - Text and appearance	CreationEdge, Jeroen, Kritner, Nick DeFazio, stackoverfloweth, tmg
6	Components introduction	4444, AldoRomo88, ASindleMouat
7	Custom Bindings	AldoRomo88, natus, Nick DeFazio
8	Debugging a knockout.js application	Adam Wolski, AldoRomo88, natus, user3297291
9	Equivalents of AngularJS bindings	Jeroen, Quango
10	Href binding	4444, Jeroen
11	Observables	Arun S, Norbert
12	Working with knockout foreach binding with JSON	suyesh