



Knockout.JS

JAVASCRIPT

tutorialspoint
SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

KnockoutJS is basically a library written in JavaScript, based on MVVM pattern that helps developers in building rich and responsive websites. KnockoutJS library provides an easy and clean way to handle complex data-driven interfaces. It is independent of any other framework.

This tutorial covers most of the topics required for a basic understanding of KnockoutJS and explains its various functionalities.

Audience

This tutorial is designed for software programmers who want to learn the basics of KnockoutJS and its programming concepts in a simple and easy way. This tutorial will give you enough understanding on the components of KnockoutJS with suitable examples.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of HTML, CSS, JavaScript, Document Object Model (DOM), and any text editor. As we are going to develop web-based application using KnockoutJS, it will be good if you have an understanding on how the Internet and web-based applications work.

Copyright & Disclaimer

© Copyright 2017 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial.....	i
Audience	i
Prerequisites	i
Copyright & Disclaimer.....	i
Table of Contents	ii
1. KNOCKOUTJS – OVERVIEW	1
2. KNOCKOUTJS – ENVIRONMENT SETUP	2
3. KNOCKOUTJS – APPLICATION	6
Architecture	6
4. KNOCKOUTJS – MVVM FRAMEWORK.....	8
Architecture	8
5. KNOCKOUTJS – OBSERVABLES.....	10
Reading and Writing Observables	12
Observable Arrays.....	12
ObservableArray Functions	13
push() Method	15
pop() Method.....	16
unshift() Method	18
shift() Method	20
reverse() Method	21
sort() Method.....	23
splice() Method	24
indexOf() Method	26

slice() Method	28
removeAll() Method	29
remove() Method	31
remove(function(item) { condition }) Method	32
remove([set of values]) Method	34
6. KNOCKOUTJS – COMPUTED OBSERVABLES	36
Pure Computed Observables	39
Writable Computed Observables	40
7. KNOCKOUTJS – DECLARATIVE BINDINGS	42
Binding Syntax	42
Binding Context	43
Visible Binding	45
Text Binding	46
Observations	49
HTML Binding	51
CSS Binding	52
Style Binding	54
Observations	56
Attr Binding	56
Foreach Binding	59
Observations	62
If Binding	63
Observations	65
Ifnot Binding	66
with Binding	68
Observations	70

component Binding	70
Component Processing Workflow	71
Observations	75
Click Binding	77
Observations	79
Event Binding	85
Observations	87
Submit Binding	93
Enable Binding	95
Disable Binding	98
Value Binding	100
Observations	102
textInput Binding	105
Observations	106
hasFocus Binding	107
checked Binding	109
options Binding	113
Observations	115
selectedOptions Binding	117
uniqueName Binding	119
8. KNOCKOUTJS – DEPENDENCY TRACKING	122
Controlling Dependencies Using Peek	124
Observations	126
9. KNOCKOUTJS – TEMPLATING	127
Observations	128

10. KNOCKOUTJS – COMPONENTS	138
Component Registration	138
Component Binding	143
Custom Element.....	143
Component Loaders	147

1. KnockoutJS – Overview

KnockoutJS is basically a library written in JavaScript, based on MVVM pattern that helps developers build rich and responsive websites. The model separates the application's Model (stored data), View (UI) and View Model (JavaScript Representation of model).

KnockoutJS was developed and is maintained as an open source project by Steve Sanderson, a Microsoft employee on July 5, 2010. KO is an abbreviation used for KnockoutJS. KO supports all mainstream browsers - IE 6+, Firefox 3.5+, Chrome, Opera, Safari (desktop/mobile).

Features of KnockoutJS

Here is a list of some of the most prominent features of KnockoutJS:

- **Declarative Binding:** HTML DOM elements are connected to the model through data-bind attribute using a very simple syntax. It is made easy to achieve responsiveness using this feature.
- **Automatic UI Refresh:** Any changes made to view the model data are reflected in the UI automatically and vice-versa. No need of writing extra code.
- **Dependency Tracking:** Relationship between KO attributes and KO library functions/components is transparent. Automatically tracks data changes in KO attribute and updates respective affected areas.
- **Templating:** Templates are a simple and convenient way to build complex UI structures - with the possibility of repeating or nesting blocks - as a function of view model data.
- **Extensible:** Extends custom behavior very easily.

Why Use KnockoutJS?

- KnockoutJS library provides an easy and clean way to handle complex data-driven interfaces. One can create self-updating UIs for Javascript objects.
- It is pure JavaScript Library and works with any web framework. It's not a replacement of JQuery but can work as a supplement providing smart features.
- KnockoutJS library file is very small and lightweight.
- KnockoutJS is independent of any other framework. It is compatible with other client or server side technologies.
- Most important of all KnockoutJS is open source and hence free for use.
- KnockoutJS is fully documented. The official site has full documentation including API docs, live examples, and interactive tutorials.

2. KnockoutJS – Environment Setup

It is very easy to use KnockoutJS. Simply refer the JavaScript file using `<script>` tag in HTML pages.

Knockout.js can be accessed in the following ways:

- You can download production build of Knockout.js from its [official website](#):

A page as in the following image will be displayed. Click the download link and you will get the latest knockout.js file.



Now refer the file as shown in the following code.

```
<script type='text/javascript' src='knockout-3.3.0.js'></script>
```

Update the src attribute to match the location where the downloaded files are kept.

- You can refer to the KnockoutJS library from CDNs:
 - You can refer KnockoutJS library from [Microsoft Ajax CDN](#) in your code as follows:

```
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js" type="text/javascript"></script>
```

- Alternatively you can refer to a minified version of KnockoutJS library from [CDNJS](#) as follows:

```
<script  
src="https://cdnjs.cloudflare.com/ajax/libs/knockout/3.3.0/knockout-  
min.js" type="text/javascript"></script>
```

Note: In all the chapters for this tutorial, we have referred to CDN version of the KnockoutJS library.

Example

KnockoutJS is based on Model-View-ViewModel (MVVM) pattern. We will study this pattern in depth in chapter [KnockoutJS - MVVM Framework](#). First let's take a look at a simple example of KnockoutJS.


```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Simple Example</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>

<body>
  <!-- This is called "view" of HTML markup that defines the appearance of UI -->

  <p>First String: <input data-bind="value: firstString" /></p>
  <p>Second String: <input data-bind="value: secondString" /></p>

  <p>First String: <strong data-bind="text: firstString">Hi</strong></p>
  <p>Second String: <strong data-bind="text: secondString">There</strong></p>
  <p>Derived String: <strong data-bind="text: thirdString"></strong></p>

  <script>
    <!-- This is called "viewmodel". This javascript section defines the data
and behavior of UI -->

    function AppViewModel() {
      this.firstString = ko.observable("Enter First String");
      this.secondString = ko.observable("Enter Second String");

      this.thirdString = ko.computed(function() {
        return this.firstString() + " " + this.secondString();
      }, this);
    }

    // Activates knockout.js
    ko.applyBindings(new AppViewModel());
  </script>
</body>
</html>
```

The following line refers to KnockoutJS library.

```
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"> </script>
```

We have two input boxes: **First String** and **Second String**. These 2 variables are initialized with values Enter First String and Enter Second String respectively in ViewModel.

```
<p>First String: < input data-bind="value: firstString" /> </p>
```

This is how we are binding values from ViewModel to HTML elements using '**data-bind**' attribute in the body section.

Here, 'firstString' refers to ViewModel variable.

```
this.firstString = ko.observable("Enter First String");
```

ko.observable is a concept which keeps an eye on the value changes so that it can update the underlying ViewModel data.

To understand this better, let's update the first input box to "Hello" and the second input box to "TutorialsPoint". You will see the values are updated simultaneously. We will study more about this concept in [KnockoutJS - Observables](#) chapter.

```
this.thirdString = ko.computed(function() {
    return this.firstString() + " " + this.secondString();
}, this);
```

Next, we have computed function in viewmodel. This function derives the third string based on 2 strings mentioned earlier. Thus, any updates made to these strings automatically get reflected in this derived string. There is no need of writing an extra code to accomplish this. This is just a simple example. We will study about this concept in [KnockoutJS - Computed Observables](#) chapter.

Output

Save the above code as **my_first_knockoutjs_program.html**. Open this file in your browser and you will see an output as the following.

First String:

Second String:

First String: **Enter First String**

Second String: **Enter Second String**

Derived String: **Enter First String Enter Second String**

Modify strings to "Hello" and "TutorialsPoint" and the output changes as follows.

First String:

Second String:

First String: **Hello**

Second String: **TutorialsPoint**

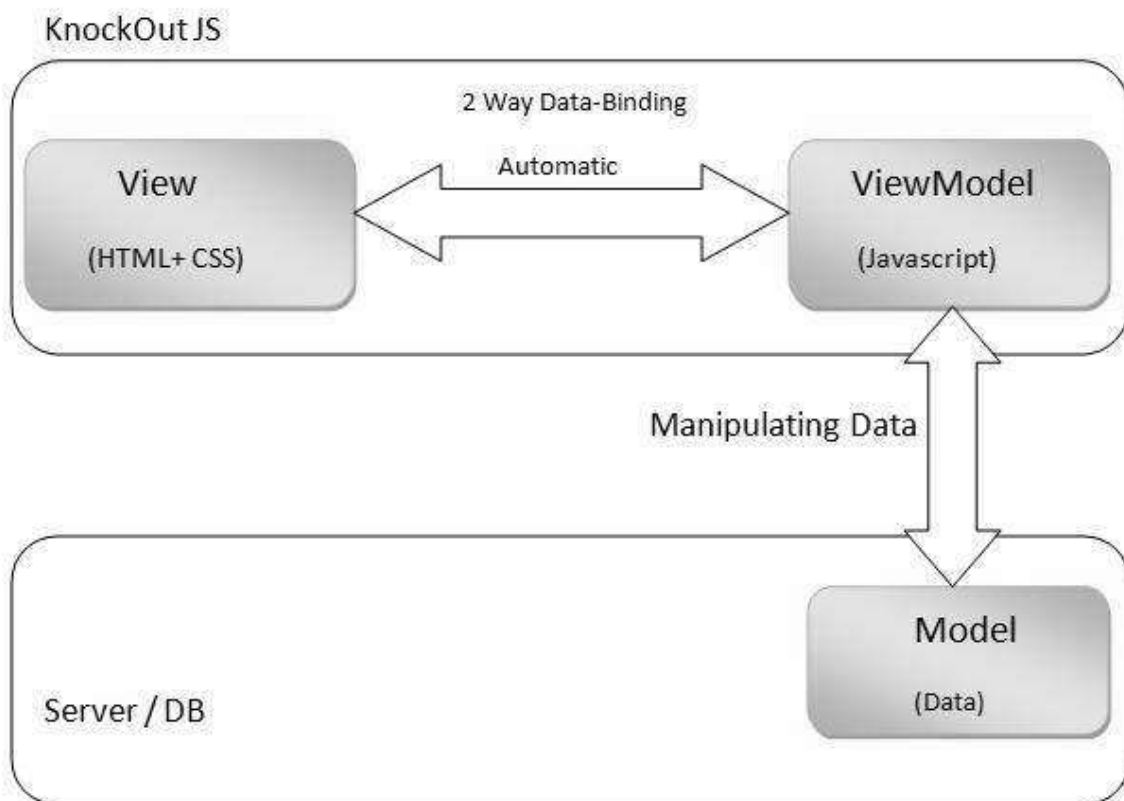
Derived String: **Hello TutorialsPoint**

3. KnockoutJS – Application

KnockoutJS is widely used for Single Page Applications - A website created with the ability to retrieve all necessary data dynamically with a single page load reducing server round trips.

KnockoutJS is a client-side framework. This is a JavaScript library which makes it very easy to bind HTML to domain data. It implements a pattern called Model-View-ViewModel (MVVM). Observables is the magic ingredient of KnockoutJS. All data remains in sync because of Observable attribute.

Architecture



View

View is nothing but user interface created using HTML elements and CSS styling.

You can bind HTML DOM elements to data model using KnockoutJS. It provides 2-way data binding between View and ViewModel using 'data-bind' concept, which means any updates done in the UI are reflected in the data model and any changes done in the data model are reflected in the UI. One can create self-updating UI with the help of knockoutJS.

ViewModel

ViewModel is a JavaScript object, which contains necessary properties and functions to represent data. View and ViewModel are connected together with declarative data-bind concept used in HTML. This makes it easy to change HTML without changing ViewModel. KnockoutJS takes care of automatic data refresh between them through the use of Observables.

Synchronization of data is achieved through binding DOM elements to Data Model, first using data-bind and then refreshing these 2 components through the use of Observables. Dependency tracking is done automatically due to this synchronization of data. No extra coding is required to achieve it. KnockoutJS allows to create direct connection between the display and underlying data.

You can create your own bindings called as custom bindings for application specific behaviors. This way Knockout gives direct control of how you want to transform your data into HTML.

Model

Model is the domain data on the server and it gets manipulated as and when the request is sent/received from ViewModel.

The data could be stored in database, cookie, or other form of persistent storage. KnockoutJS does not worry about how it is stored. It is up to the programmer to communicate between the stored data and KnockoutJS.

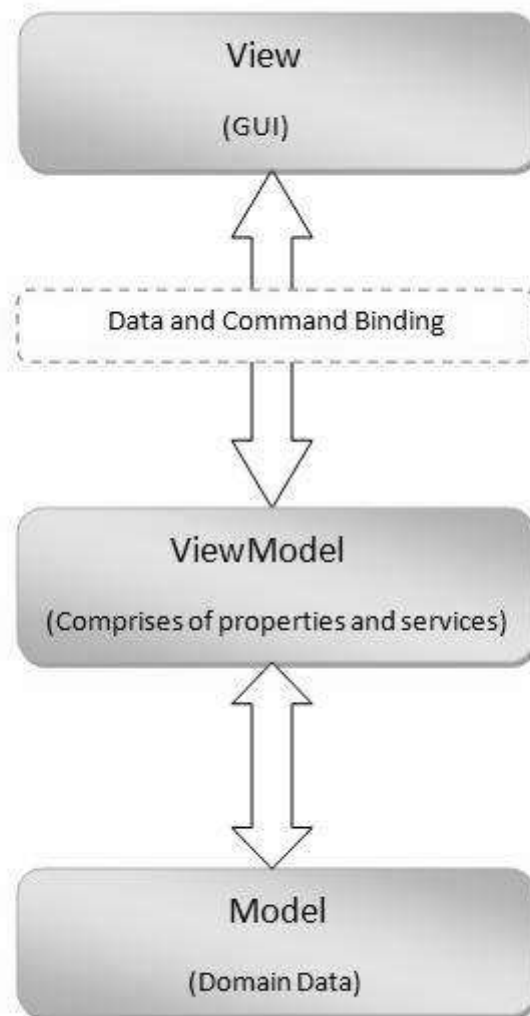
Most of the times, data is saved and loaded via an Ajax call.

4. KnockoutJS – MVVM Framework

Model-View-ViewModel (MVVM) is an architectural design pattern for developing software applications. MVVM was developed by Microsoft Architect John Gossman in 2005. This pattern is derived from Model-View-Controller (MVC) pattern. The advantage of MVVM is that it separates the application layer's graphical user interface from business logic. MVVM is responsible for handling data from the underlying model in such a way that it is represented and managed very easily. ViewModel in MVVM represents an abstract version of View's state and actions.

The view classes do not know that Model and ViewModel classes exist, also Model and ViewModel does not know that View exists. Model is also unaware that ViewModel and View exist.

Architecture



View

View is a Graphical User Interface created using markup language to represent data. View binds to properties of a ViewModel through data-bind concept, which indirectly connects to the model data. View need not be changed for any alteration done in ViewModel. Changes made to data in ViewModel is automatically propagated in View due to binding.

Model

Model is domain data or business object, which holds real-time data. Model does not carry behaviors. Behavior is mostly implemented in business logic.

ViewModel

ViewModel is the center place, where data from Model and View's display logic are bundled together. ViewModel holds the dynamic state of data. There is an implicit binder in between View and ViewModel to communicate with each other. This binding is inclusive of declarative data and command binding. Synchronization of View and ViewModel is achieved through this binding. Any change made in View is reflected in ViewModel, and similarly any change in ViewModel gets automatically reflected in View. Existence of this 2-way binding mechanism is a key aspect of this MVVM pattern.

5. KnockoutJS – Observables

KnockoutJS is build upon the following 3 important concepts.

- Observables and dependency tracking between them - DOM elements are connected to ViewModel via 'data-bind'. They exchange information through Observables. This automatically takes care of dependency tracking.
- Declarative Bindings between UI and ViewModel - DOM elements are connected to ViewModel via 'data-bind' concept.
- Templating to create re-usable components - Templating provides a robust way to create complex web applications.

We will study Observables in this chapter.

As the name specifies, when you declare a ViewModel data/property as Observable, any data modification each time automatically gets reflected at all places the data is used. This also includes refreshing the related dependencies. KO takes care of these things and there is no need to write extra code to achieve this.

Using Observable, it becomes very easy to make UI and ViewModel communicate dynamically.

Syntax

You just need to declare ViewModel property with function **ko.observable()** to make it Observable.

```
this.property = ko.observable('value');
```

Example

Let's take a look at the following example which demonstrates the use of Observable.

```
<!DOCTYPE html>
<head>
<title>KnockoutJS Observable Example</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <!-- This is called "view" of HTML markup that defines the appearance of UI -->

  <p>Enter your name: <input data-bind="value: yourName" /></p>
  <p>Hi <strong data-bind="text: yourName"></strong> Good Morning!!!</p>
```

```

<script>
  <!-- This is called "viewModel". This javascript section defines the data
  and behavior of UI -->

  function AppViewModel() {
    this.yourName = ko.observable("");
  }

  // Activates knockout.js
  ko.applyBindings(new AppViewModel());
</script>
</body>
</html>

```

The following line is for the input box. As can be seen, we have used data-bind attribute to bind yourName value to ViewModel.

```
<p>Enter your name: <input data-bind="value: yourName" /> <p>
```

The following line just prints the value of yourName. Note, that here data-bind type is the text as we are simply reading the value.

```
<p>Hi <strong data-bind="text: yourName"></strong> Good Morning!!!</p>
```

In the following line, ko.observable keeps an eye on yourName variable for any modification in data. Once there is a modification, the corresponding places also get updated with the modified value. When you run the following code, an input box will appear. As and when you update that input box, the new value will get reflected or refreshed in places wherever it is used.

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

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **first_observable_pgm.htm** file.
- Open this HTML file in a browser.
- Enter the name as Scott and observe that the name is reflected in the output.

Enter your name:

Hi Good Morning!!!

Data modification can take place either from the UI or from ViewModel. Irrespective of from where the data is changed, the UI and ViewModel keeps synchronization among them. This makes it a two-way-binding mechanism. In the above example, when you change your name in the input box, ViewModel gets a new value. When you change yourName property from inside ViewModel, then the UI receives a new value.

Reading and Writing Observables

Following table lists the read and write operations which can be performed on Observables.

Sr. No.	Read/Write Operation & Syntax
1	Read To read value just call Observable property without parameters like: <code>AppViewModel.yourName();</code>
2	Write To write/update value in Observable property, just pass the desired value in parameter like: <code>AppViewModel.yourName('Bob');</code>
3	Write multiple Multiple ViewModel properties can be updated in a single row with the help of chaining-syntax like: <code>AppViewModel.yourName('Bob').yourAge(45);</code>

Observable Arrays

Observable declaration takes care of data modifications of a single object. ObservableArray works with the collection of objects. This is a very useful feature when you are dealing with complex applications containing multiple type of values and changing their status frequently based on the user actions.

Syntax

```
this.arrayName = ko.observableArray();    // It's an empty array
```

Observable array only tracks which objects in it are added or removed. It does not notify if the individual object's properties are modified.

Initialize It for the First Time

You can initialize your array and at the same time you can declare it as Observable by passing the initial values to the constructor as follows.

```
this.arrayName = ko.observableArray(['scott','jack']);
```

Reading from Observable Array

You can access Observable array elements as follows.

```
alert('The second element is ' + arrayName()[1]);
```

ObservableArray Functions

KnockoutJS has its own set of Observable array functions. They are convenient because:

- These functions work on all browsers.
- These functions will take care of dependency tracking automatically.
- Syntax is easy to use. For example, to insert an element into an array, you just need to use `arrayName.push('value')` instead of `arrayName().push('value')`.

Following is the list of various Observable Array methods.

Sr. No.	Methods & Description
1	<u>push('value')</u> Inserts a new item at the end of array.
2	<u>pop()</u> Removes the last item from the array and returns it.
3	<u>unshift('value')</u> Inserts a new value at the beginning of the array.
4	<u>shift()</u> Removes the first item from the array and returns it.
5	<u>reverse()</u> Reverses the order of the array.

6	<u>sort()</u> Sorts array items in an ascending order.
7	<u>splice(start-index,end-index)</u> Accepts 2 parameters - start-index and end-index - removes items starting from start to end index and returns them as an array.
8	<u>indexOf('value')</u> This function returns the index of the first occurrence of parameter provided.
9	<u>slice(start-index,end-index)</u> This method slices out a piece of an array. Returns the items from start-index up to end-index.
10	<u>removeAll()</u> Removes all items and returns them as an array.
11	<u>remove('value')</u> Removes items that match the parameter and returns as an array.
12	<u>remove(function(item) { condition })</u> Removes items which are satisfying the condition and returns them as an array.
13	<u>remove([set of values])</u> Removes items that match with a given set of values.
14	<u>destroyAll()</u> Marks all items in an array with property _destroy with value true.
15	<u>destroy('value')</u> Searches for an item equal to the parameter and mark it with a special property _destroy with value true.
16	<u>destroy(function(item) { condition })</u> Finds all items which are satisfying the condition, marks them with property _destroy with true value.
17	<u>destroy([set of values])</u> Finds the items that match with a given set of values, marks them as _destroy with true value.

Note: Destroy and DestroyAll Functions from ObservableArrays are mostly for 'Ruby on Rails' developers only.

When you use destroy method, the corresponding items are not really deleted from array at that moment but are made hidden by marking them with property `_destroy` with `true` value so that they can't be read by UI. Items marked as `_destroy` equal to `true` are deleted later while dealing with JSON object graph.

push() Method

Description

The KnockoutJS Observable **push('value')** method inserts a new item at the end of an array.

Syntax

```
arrayName.push('value')
```

Parameters

Accepts only one parameter, that is the value to be inserted.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS Observable Array push() Method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>Example to demonstrate push() method.</p>
  <p>Enter name: <input data-bind='value: empName' /></p>
  <p><button data-bind="click: addEmp">Add Emp </button></p>
  <p>Array of employees: <span data-bind="text: empArray()" ></span></p>

  <script>
function EmployeeModel(){
  this.empName = ko.observable("");
  this.chosenItem = ko.observableArray("");
  this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);
//Initial Values

  this.addEmp = function() {
    if (this.empName() != "") {
      this.empArray.push(this.empName()); //insert accepted value in array
      this.empName("");
    }
  }
}
```

```

        }.bind(this);
    }/
    var emp = new EmployeeModel();
    ko.applyBindings(emp);
</script>

</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-push.htm** file.
- Open this HTML file in a browser.
- Type 'Tom' as an input and click the Add Emp button.

Example to demonstrate push() method.

Enter name:

Add Emp

Array of employees: Scott,James,Jordan,Lee,RoseMary,Kathie

pop() Method

Description

The KnockoutJS Observable **pop()** method removes the last item from an array and returns it.

Syntax

```
arrayName.pop()
```

Parameters

Does not accept any parameters.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray pop method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>Example to demonstrate pop()method.</p>
  <button data-bind="click: popEmp">Remove Emp</button>
  <p>Array of employees: <span data-bind="text: empArray()" ></span></p>

  <script>
  function EmployeeModel(){
    this.empName = ko.observable("");
    this.chosenItem = ko.observableArray("");
    this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);

    this.popEmp = function() {
      this.empArray.pop();
    }
  }
  var em = new EmployeeModel();
  ko.applyBindings(em);
  </script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-pop.htm** file.
- Open this HTML file in a browser.
- Click the Remove Emp button and observe that the last element is removed.

Example to demonstrate pop() method.

Array of employees: Scott,James,Jordan,Lee,RoseMary,Kathie

unshift() Method

Description

The KnockoutJS Observable **unshift('value')** method inserts a new item at the beginning of the array.

Syntax

```
arrayName.unshift('value')
```

Parameters

Accepts one parameter, that is the value to be inserted.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray unshift method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>Example to demonstrate unshift() method.</p>
  <p>Enter name: <input data-bind='value: empName' /></p>
  <button data-bind="click: unshiftEmp">Add Emp in Beginning</button><br><br>
  <p>Array of employees: <span data-bind="text: empArray()" ></span></p>

  <script>
```

```

function EmployeeModel(){
    this.empName = ko.observable("");
    this.chosenItem = ko.observableArray("");
    this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);

    this.unshiftEmp = function() {
        if (this.empName() != "") {
            this.empArray.unshift(this.empName());    // insert at the beginning
            this.empName("");
        }
    }.bind(this);
}
var em = new EmployeeModel();
ko.applyBindings(em);
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-unshift.htm** file.
- Open this HTML file in a browser.
- Enter name as Tom and click the Add Emp in Beginning button.

Example to demonstrate `unshift()` method.

Enter name:

Add Emp in Beginning

Array of employees: Scott,James,Jordan,Lee,RoseMary,Kathie

shift() Method

Description

The KnockoutJS Observable **shift()** method removes the first item from the array and returns it.

Syntax

```
arrayName.shift()
```

Parameters

Does not accept any parameter.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray shift method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>Example to demonstrate shift() method.</p>
  <button data-bind="click: shiftEmp">Remove First Emp</button>
  <p>Array of employees: <span data-bind="text: empArray()" ></span></p>

  <script>
function EmployeeModel(){
  this.empName = ko.observable("");
  this.chosenItem = ko.observableArray("");
  this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);

  this.shiftEmp = function() {
    this.empArray.shift();    //remove first item
  }
}

var em = new EmployeeModel();
```

```

    ko.applyBindings(em);
  </script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-shift.htm** file.
- Open this HTML file in a browser.
- Click the Remove First Emp button and observe that the first element is removed.

Example to demonstrate shift() method.

Remove First Emp

Array of employees: Scott,James,Jordan,Lee,RoseMary,Kathie

reverse() Method

Description

The KnockoutJS Observable **reverse()** method reverses the order of the array.

Syntax

```
arrayName.reverse()
```

Parameters

Does not accept any parameter.

Example

```

<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray reverse method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>

```

```

</head>
<body>
  <p>Example to demonstrate reverse() method.</p>
  <button data-bind="click: revEmp">Reverse Array</button>
  <p>Array of employees: <span data-bind="text: empArray()" ></span></p>

  <script>
    function EmployeeModel(){
      this.empName = ko.observable("");
      this.chosenItem = ko.observableArray("");
      this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);

      this.revEmp = function() {
        this.empArray.reverse(); // reverse order
      }
    }
    var em = new EmployeeModel();
    ko.applyBindings(em);
  </script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-reverse.htm** file.
- Open this HTML file in a browser.
- Click the Reverse Array button and observe that the array order is changed.

Example to demonstrate reverse() method.

Reverse Array

Array of employees: Scott,James,Jordan,Lee,RoseMary,Kathie

sort() Method

Description

The KnockoutJS Observable **sort()** method sorts all items in the array.

By default, items are sorted in an ascending order. For sorting an array in a descending order, use reverse() method on sorted array.

Syntax

```
arrayName.sort()
```

Parameters

Does not accept any parameter.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray sort method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>Example to demonstrate sort() method.</p>
  <button data-bind="click: sortEmp">Sort Array</button>
  <p>Array of employees: <span data-bind="text: empArray()" ></span></p>

  <script>
```



```

function EmployeeModel(){
    this.empName = ko.observable("");
    this.chosenItem = ko.observableArray("");
    this.empArray =
    ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);

    this.sortEmp = function() {
        this.empArray.sort(); //sort array
    }
}
var em = new EmployeeModel();
ko.applyBindings(em);
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-sort.htm** file.
- Open this HTML file in a browser.
- Click the Sort Array button and see that the array is sorted.

Example to demonstrate sort() method.

Sort Array

Array of employees: Scott,James,Jordan,Lee,RoseMary,Kathie

splice() Method

Description

The KnockoutJS Observable **splice()** method takes 2 parameters specifying the start-index and the end-index. It removes items starting from start to end index and returns them as an array.

Syntax

```
arrayName.splice(start-index,end-index)
```

Parameters

Accepts 2 parameters, start-index is start index and end-index is end index.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray splice method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>Example to demonstrate splice() method.</p>
  <button data-bind="click: spliceEmp">Splice Emp</button>
  <p>Array of employees: <span data-bind="text: empArray()" ></span></p>

  <script>
function EmployeeModel(){
  this.empName = ko.observable("");
  this.chosenItem = ko.observableArray("");
  this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);

  this.spliceEmp = function() {
    alert("Splice is removing items from index 1 to 3(If exists).");
    this.empArray.splice(1,3); // remove 2nd,3rd and 4th item, as array
index starts with 0.
  }
}
var em = new EmployeeModel();
ko.applyBindings(em);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-splice.htm** file.
- Open this HTML file in a browser.
- Click the Splice Emp button and observe that items starting from index 1 to 3 are removed.

Example to demonstrate splice() method.

Splice Emp

Array of employees: Scott,James,Jordan, Lee,RoseMary,Kathie

indexOf() Method

Description

The KnockoutJS Observable **indexOf('value')** method returns the index of the first occurrence of the parameter provided. This function will return -1, if no matching element is found.

Syntax

```
arrayName.indexOf('value')
```

Parameters

Accepts 1 parameter, whose index will be returned.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray indexOf method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
```

```

<body>
  <p>Example to demonstrate indexOf() method.</p>
  <p>Index of Employee 'Jordan':<span data-bind="text:
empArray().indexOf('Jordan')"></span ></p>
  <p>Array of employees: <span data-bind="text: empArray()" ></span></p>

  <script>
  function EmployeeModel(){
  this.empName = ko.observable("");
  this.chosenItem = ko.observableArray("");
  this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);
  }
  var em = new EmployeeModel();
  ko.applyBindings(em);
  </script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-indexof.htm** file.
- Open this HTML file in a browser.

Example to demonstrate indexOf() method.

Index of Employee 'Jordan':2

Array of employees: Scott,James,Jordan, Lee, RoseMary, Kathie

slice() Method

Description

The KnockoutJS Observable **slice()** method slices out a piece of an array. Slice here is the same as native JavaScript slice function. Returns the elements from start-index up to end-index.

Syntax

```
arrayName.slice(start-index,end-index)
```

Parameters

Accepts 2 parameters, start-index and end-index.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray slice method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
    <p>Example to demonstrate slice() method.</p>
    <p>slice(1,3) to show items starting from index 1 up to 3:<span data-
bind="text: empArray().slice(1,3)"></span ></p>
    <p>Array of employees: <span data-bind="text: empArray()" ></span></p>

    <script>
    function EmployeeModel(){
        this.empName = ko.observable("");
        this.chosenItem = ko.observableArray("");
        this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);
    }
    var em = new EmployeeModel();
    ko.applyBindings(em);
    </script>
</body>
```

```
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-slice.htm** file.
- Open this HTML file in a browser.

Example to demonstrate slice() method.

slice(1,3) to show items starting from index 1 up to 3:James,Jordan

Array of employees: Scott,James,Jordan,Lee,RoseMary,Kathie

removeAll() Method

Description

The KnockoutJS Observable **removeAll()** method removes all items and returns them as an array.

Syntax

```
arrayName.removeAll()
```

Parameters

Does not accept any parameter.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray removeAll method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
```

```

<p>Example to demonstrate removeAll() method.</p>
<button data-bind="click: removeAllEmp">Remove All Emps</button>
<p>Array of employees: <span data-bind="text: empArray()" ></span></p>

<script>
function EmployeeModel(){
this.empName = ko.observable("");
this.chosenItem = ko.observableArray("");
this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);

    this.removeAllEmp = function() {
        this.empArray.removeAll();
    }
}
var em = new EmployeeModel();
ko.applyBindings(em);
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-removeall.htm** file.
- Open this HTML file in a browser.
- Click the Remove All Emps Button.

Example to demonstrate removeAll() method.

Remove All Emps

Array of employees: Scott,James,Jordan, Lee, RoseMary, Kathie

remove() Method

Description

The KnockoutJS Observable **remove('value')** method removes the items matching with the 'value' and returns as an array.

Syntax

```
arrayName.remove('value')
```

Parameters

Accepts one parameter as a value to be removed.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray remove method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>Example to demonstrate remove() method.</p>
  <button data-bind="click: removeEmp">Remove selected Emp</button>
  <p>Array of employees:</p>
  <select multiple="true" size="8" data-bind="options: empArray ,
selectedOptions: chosenItem"> </select>

  <script>
function EmployeeModel(){
this.empName = ko.observable("");
this.chosenItem = ko.observableArray("");
this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);

this.removeEmp = function(chosenItem) {
if (this.chosenItem().length == 0) {
  alert("Please select item/s to be removed.");
}
}
```

```

        while(this.chosenItem().length > 0) {
            this.empArray.remove(this.chosenItem()[0]);
        }
    }
}
var em = new EmployeeModel();
ko.applyBindings(em);
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-remove.htm** file.
- Open this HTML file in a browser.
- Select the item to be removed from the list and click the Remove selected Emp Button.

Example to demonstrate remove() method.

Remove selected Emp

Array of employees:

Scott
James
Jordan
Lee
RoseMary
Kathie

remove(function(item) { condition }) Method

Description

The KnockoutJS Observable **remove(function(item) { return condition })** method removes the items which are satisfying a condition and returns them as an array.

Syntax

```
arrayName.remove(function(item) { return condition })
```

Parameters

This method accepts a parameter in the form of a function having a condition. Items from an array satisfying the mentioned condition are removed.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray function based remove method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>Example to demonstrate function based remove() method.</p>
  <button data-bind="click: removeoncondEmp">Remove on condition</button>
  <p>Array of employees: <span data-bind="text: empArray()" ></span></p>

  <script>
    function EmployeeModel(){
      this.empName = ko.observable("");
      this.chosenItem = ko.observableArray("");
      this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);
      this.removeoncondEmp = function() {
        alert("This function is removing items whos value is 'James'.");
        this.empArray.remove(function (empName) { return empName ===
'James' }); //remove where empName is James
      }
    }
    var em = new EmployeeModel();
    ko.applyBindings(em);
  </script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-remove-fun.htm** file.
- Open this HTML file in a browser.
- Click the Remove on condition Button.

Example to demonstrate function based remove() method.

Remove on condition

Array of employees: Scott,James,Jordan,Lee,RoseMary,Kathie

remove([set of values]) Method

Description

The KnockoutJS Observable **remove([set of values])** method removes the items that match a given set of values.

Syntax

```
arrayName.remove([set of values])
```

Parameters

This method accepts a parameter in the form of a values set.

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS ObservableArray remove multiple method</title>
<script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>Example to demonstrate remove([set of values]) method.</p>
  <button data-bind="click: removemultiEmp">Remove multiple Emps</button>
  <p>Array of employees: <span data-bind="text: empArray()" ></span></p>
```

```

<script>
function EmployeeModel(){
this.empName = ko.observable("");
this.chosenItem = ko.observableArray("");
this.empArray =
ko.observableArray(['Scott','James','Jordan','Lee','RoseMary','Kathie']);

    this.removemultiEmp = function() {
        alert("This function is removing multiple matching items
e.g. ['RoseMary','Lee'].");
        this.empArray.removeAll(['RoseMary','Lee']);
    }
}
var em = new EmployeeModel();
ko.applyBindings(em);
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **array-remove-set.htm** file.
- Open this HTML file in a browser.
- Click the Remove multiple Emps Button.

Example to demonstrate remove([set of values]) method.

Remove multiple Emps

Array of employees: Scott,James,Jordan, Lee,RoseMary,Kathie

6. KnockoutJS – Computed Observables

Computed Observable is a function which is dependent on one or more Observables and automatically updates whenever its underlying Observables (dependencies) change.

Computed Observables can be chained.

Syntax

```
this.varName = ko.computed(function(){  
    ...  
    ... // function code  
    ...  
},this);
```

Example

Let us look at the following example which demonstrates the use of Computed Observables.

```
<!DOCTYPE html>  
<head >  
    <title>KnockoutJS Computed Observables</title>  
    <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"></script>  
</head>  
<body>  
  
    <p>Enter first number: <input data-bind="value: a" /></p>  
    <p>Enter second number: <input data-bind="value: b"/></p>  
    <p>Average := <span data-bind="text: totalAvg"></span></p>  
  
<script>  
    function MyViewModel() {  
        this.a = ko.observable(10);  
        this.b = ko.observable(40);  
  
        this.totalAvg = ko.computed(function(){  
            if(typeof(this.a()) !== "number" || typeof(this.b()) !== "number"){
```

```

        this.a(Number(this.a())); //convert string to Number
        this.b(Number(this.b())); //convert string to Number
    }
    total = (this.a() + this.b())/2 ;
    return total;
    },this);
}
ko.applyBindings(new MyViewModel());

</script>
</body>
</html>

```

In the following lines, first two are for accepting input values. Third line prints the average of these two numbers.

```

<p>Enter first number: <input data-bind="value: a" /></p>
<p>Enter second number: <input data-bind="value: b"/></p>
<p>Average := <span data-bind="text: totalAvg"></span></p>

```

In the following lines, type of Observables **a** and **b** is number when they are initialized for the first time inside ViewModel. However, in KO every input accepted from UI is by default in the String format. So they need to be converted to Number so as to perform arithmetic operation on them.

```

this.totalAvg = ko.computed(function(){
    if(typeof(this.a()) !== "number" || typeof(this.b()) !== "number"){
        this.a(Number(this.a())); //convert string to Number
        this.b(Number(this.b())); //convert string to Number
    }
    total = (this.a() + this.b())/2 ;
    return total;
},this);

```

In the following line, the calculated average is displayed in the UI. Note that data-bind type of totalAvg is just text.

```

<p>Average := <span data-bind="text: totalAvg"></span></p>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **computed-observable.htm** file.
- Open this HTML file in a browser.
- Enter any 2 numbers in the text boxes and observe that the average is calculated.

Enter first number:

Enter second number:

Average := 25

Managing 'This'

Note that in the above example, the second parameter is provided as **this** to Computed function. It is not possible to refer to Observables **a()** and **b()** without providing **this**.

To overcome this, **self** variable is used which holds the reference of **this**. Doing so, there is no need to track **this** throughout the code. Instead, **self** can be used.

Following ViewModel code is rewritten for the above example using self.

```
function MyViewModel(){
    self = this;
    self.a = ko.observable(10);
    self.b = ko.observable(40);

    this.totalAvg = ko.computed(function(){
        if(typeof(self.a()) !== "number" || typeof(self.b()) !== "number"){
            self.a(Number(self.a())); //convert string to Number
            self.b(Number(self.b())); //convert string to Number
        }
        total = (self.a() + self.b())/2 ;
        return total;
    });
}
```


Pure Computed Observables

A Computed Observable should be declared as **Pure** Computed Observable if that Observable is simply calculating and returning the value and not directly modifying the other objects or state. Pure Computed Observables helps Knockout to manage re-evaluation and memory usage efficiently.

Notifying Subscribers Explicitly

When a Computed Observable is returning primitive data type value (String, Boolean, Null, and Number) then its subscribers are notified if and only if the actual value change takes place. It means if an Observable has received the value same as the previous value, then its subscribers are not notified.

You can make Computed Observables always explicitly notify the observers, even though the new value is the same as the old by using the **notify** syntax as follows.

```
myViewModel.property = ko.pureComputed(function() {
    return ...; // code logic goes here
}).extend({ notify: 'always' });
```

Limiting Change Notifications

Too many expensive updates can result in performance issues. You can limit the number of notifications to be received from Observable using **rateLimit** attribute as follows.

```
// make sure there are updates no more than once per 100-millisecond period
myViewModel.property.extend({ rateLimit: 100 });
```

Finding Out If a Property is Computed Observable

In certain situations, it might be necessary to find out if a property is a Computed Observable. Following functions can be used to identify the types of Observables.

Sr. No.	Function
1	ko.isComputed Returns true if the property is Computed Observable.
2	ko.isObservable Returns true if the property is Observable, Observable array, or Computed Observable.
3	ko.isWritableObservable Returns true if Observable, Observable array, or Writable Computed Observable. (This is also called as ko.isWriteableObservable)

Writable Computed Observables

Computed Observable is derived from one or multiple other Observables, so it is read only. However, it is possible that one can make Computed Observable writable. For this you need to provide callback function that works on written values.

These writable Computed Observables work just like regular Observables. In addition, they require custom logic to be built for interfering read and write actions.

One can assign values to many Observables or Computed Observable properties using the chaining syntax as follows.

```
myViewModel.fullName('Tom Smith').age(45)
```

Example

Following example demonstrates the use of Writable Computable Observable.

```
<!DOCTYPE html>
<head >
  <title>KnockoutJS Writable Computed Observable</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"></script>
</head>
<body>
  <p>Enter your birth Date: <input type="date" data-bind="value: rawDate" > </p>
  <p> <span data-bind="text: yourAge"></span></p>

  <script>
    function MyViewModel(){
      this.yourAge = ko.observable();
      today = new Date();
      rawDate = ko.observable();

      this.rawDate = ko.pureComputed({
        read: function(){
          return this.yourAge;
        },
        write: function(value){
          var b = Date.parse(value);    // convert birth date into milliseconds
          var t = Date.parse(today);    // convert today's date into milliseconds
          diff = t - b;                 // take difference
        }
      });
    }
  </script>
</body>
```

```

        var y = Math.floor(diff/31449600000);
        // difference is converted into years. 31449600000 milliseconds form a year.

        var m = Math.floor((diff % 31449600000)/604800000/4.3);
        //calculating months. 604800000 milliseconds form a week.

        this.yourAge("You are " + y + " year(s) " + m +" months old.");
    },
    owner: this
  });
}
ko.applyBindings(new MyViewModel());

</script>
</body>
</html>

```

In the above code, **rawDate** is pureComputed property accepted from UI. **yourAge** Observable is derived from **rawDate**.

Dates in JavaScript are manipulated in milliseconds. Hence, both the dates (today date and birth date) are converted into milliseconds and then the difference between them is converted back in years and months.

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **writable_computed_observable.htm** file.
- Open this HTML file in a browser.
- Enter any birth date and observe that the age is calculated.

Enter your birth Date:

7. KnockoutJS – Declarative Bindings

Declarative binding in KnockoutJS provides a powerful way to connect data to UI.

It is important to understand the relationship between bindings and Observables. Technically, these two are different. You can use normal JavaScript object as ViewModel and KnockoutJS can process View's binding correctly.

Without Observable, the property from the UI will be processed only for the first time. In this case, it cannot update automatically based on the underlying data update. To achieve this, bindings must be referred to Observable properties.

Binding Syntax

The binding consists of 2 items, the binding **name** and **value**. Following is a simple example:

```
Today is : <span data-bind="text: whatDay"></span>
```

Here, text is the binding name and whatDay is the binding value. You can have multiple bindings separated by comma, as shown in the following syntax.

```
Your name: <input data-bind="value: yourName, valueUpdate: 'afterkeydown'" />
```

Here, value is updated after each key is pressed.

Binding Values

The binding value can be a **single value**, **literal**, a **variable** or can be a **JavaScript expression**. If the binding refers to some invalid expression or reference, then KO will produce an error and stop processing the binding.

Following are few examples of bindings.

```
<!-- simple text binding -->
  <p>Enter employee name: <input data-bind='value: empName' /></p>

<!-- click binding, call a specific function -->
  <button data-bind="click: sortEmpArray">Sort Array</button>

<!-- options binding -->
  <select multiple="true" size="8" data-bind="options: empArray ,
  selectedOptions: chosenItem"> </select>
```

Note the following points:

- Whitespaces do not make any difference.
- Starting from KO 3.0, you can skip the binding value which will give binding an undefined value.

Binding Context

The data that is being used in current bindings can be referenced by an object. This object is called **binding context**.

Context hierarchy is created and managed by KnockoutJS automatically. Following table lists the different types of binding contexts provided by KO.

Sr. No.	Binding Context Types & Description
1	\$root This always refers to top level ViewModel. This makes it possible to access top level methods for manipulating ViewModel. This is usually the object, which is passed to ko.applyBindings.
2	\$data This property is lot like this keyword in Javascript object. \$data property in a binding context refers to ViewModel object for the current context.
3	\$index This property contains index of a current item of an array inside a foreach loop. The value of \$index will change automatically as and when the underlying Observable array is updated. Obviously, this context is available only for foreach bindings.
4	\$parent This property refers to parent ViewModel object. This is useful when you want to access outer ViewModel properties from inside of a nested loop.
5	\$parentContext The context object which is bound at the parent level is called \$parentContext . This is different from \$parent. \$parent refers to data. Whereas, <i>\$parentContext</i> refers to binding context. E.g. you might need to access the index of outer foreach item from an inner context.

6	\$rawdata This context holds raw ViewModel value in the current situation. This resembles \$data but the difference is, if ViewModel is wrapped in Observable, then \$data becomes just unwrapped. ViewModel and \$rawdata becomes actual Observable data.
7	\$component This context is used to refer to ViewModel of that component, when you are inside a particular component. E.g. you might want to access some property from ViewModel instead of current data in the template section of component.
8	\$componentTemplateNodes This represents an array of DOM nodes passed to that particular component when you are within a specific component template.

Following terms are also available in binding but are not actually binding context.

- **\$context**: This is nothing but existing binding context object.
- **\$element**: This object refers to an element in DOM in the current binding.

Working with Text and Appearances

Following is a list of binding types provided by KO to deal with text and visual appearances.

Sr. No.	Binding Type & Usage
1	<u>visible: <binding-condition></u> To show or hide HTML DOM element depending on certain conditions.
2	<u>text: <binding-value></u> To set the content of an HTML DOM element.
3	<u>html: <binding-value></u> To set the HTML markup contents of a DOM element.
4	<u>css: <binding-object></u> To apply CSS classes to an element.
5	<u>style: <binding-object></u> To define the inline style attribute of an element.

6	<u>attr: <binding-object></u> To add attributes to an element dynamically.
---	--

Visible Binding

As the name specifies, this binding causes the associated DOM element to be visible or hidden based on the value passed in the binding.

Syntax

```
visible: <binding-condition>
```

Parameters

- When the parameter transforms into false-like value (such as boolean false, or 0, or null, or undefined), then the binding sets `display:none` for `yourElement.style.display` making it hidden. This is given more priority over any existing styles in CSS.
- When the parameter transforms into true-like value (such as boolean true, or non-null object or array), the binding removes `yourElement.style.display` value, making it visible.
- If this is an observable property, then the binding will update visibility every time the property changes. Else, it will just set the visibility once.
- The parameter value can also be JavaScript function or arbitrary JavaScript expression that returns a Boolean. DOM element is made visible or hidden based on the output.

Example

Let us take a look at the following example.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Visible Binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <div data-bind="visible: showMe">
    You are seeing this line because showMe function is set to true.
```

```
</div>
<script>
    function AppViewModel() {
        this.showMe = ko.observable(false); // hidden initially
    }
    var vm = new AppViewModel();
    ko.applyBindings(vm); // shown now
    vm.showMe(true);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **visible-bind.htm** file.
- Open this HTML file in a browser.

You are seeing this line because showMe function is set to true.

Text Binding

Text binding causes the associated DOM element to display the text value of the parameter. This is used in text-level DOM elements such as **** or ****. The text binding accepts any data type and parses it into String before rendering it.

Syntax

```
text: <binding-value>
```


Parameters

- KO sets the element's content to a text node with your parameter value. Any previous content will be overwritten.
- If the parameter is an observable, then the element value is updated whenever the underlying property changes, else it is assigned only for the first time.
- If anything other than a Number or a String is passed, then KO parses it into a String format equivalent to `yourParameter.toString()`.
- The parameter value can also be JavaScript function or arbitrary JavaScript expression that returns a text value.

Example

Let us take a look at the following example which demonstrates the use of text binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS text binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p data-bind="text: hiThere"></p>

  <script>
    function AppViewModel() {
      this.hiThere = ko.observable("Hi Tutorialspoint !!!");
    }
    var vm = new AppViewModel();
    ko.applyBindings(vm);
  </script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **text-bind.htm** file.
- Open this HTML file in a browser.

Hi Tutorialspoint !!!

Example

Let us take a look at another example in which the text is derived using Computed Observable.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS text binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>Your full Name is <span data-bind="text: fullName"></span></p>

  <script>
    function AppViewModel() {
      this.firstName= ko.observable("John");
      this.lastName= ko.observable("Smith");

      this.fullName = ko.computed( function(){
        return this.firstName()+" "+this.lastName();
      },this);
    }
  </script>
</body>
```

```
    }  
    var vm = new AppViewModel();  
    ko.applyBindings(vm);  
  </script>  
</body>  
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **text-bind-fun.htm** file.
- Open this HTML file in a browser.

Your full Name is John Smith

Observations

HTML encoding

The text binding escapes HTML entities, meaning that it is possible to set any String value without getting it injected. For example -

```
viewModel.message("<strong>Hi Tutorialspoint !!!</strong>");
```

The above code will just print `Hi Tutorialspoint !!!` on the screen. It will not make the text bold.

Using text without container element

Sometimes it is not possible to use HTML element to set the text inside another element. In such cases, **container-less syntax** can be used which consists of comment tags shown as follows:

The **<!--ko-->** and **<!--/ko-->** comment works as start and end markers making it a virtual syntax and binds the data as if it is a real container.

Let us take a look at the following example.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS container less text binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p data-bind="text: hiThere"></p>
  <select data-bind="foreach: items">
    <option> <!--ko text: $data --><!--/ko--></option>
  </select>

  <script>
    function AppViewModel() {
      this.hiThere = ko.observable("Days of week !!!");
      this.items =
([ 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday' ]);
    }
    var vm = new AppViewModel();
    ko.applyBindings(vm);
  </script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **text-bind-containerless.htm** file.
- Open this HTML file in a browser.
- Note, that \$data binding context is used here to display the current item from the array.

Days of week !!!

monday ▼

HTML Binding

HTML binding causes the associated DOM element to display the HTML specified by the parameter. This is very useful if you want to generate HTML markup dynamically.

Syntax

```
html: <binding-value>
```

Parameters

- KnockoutJS sets DOM element's content to the parameter value provided. This functionality is also available in JQuery. If JQuery is not available, then KO is used to achieve this.
- If the parameter is **observable**, then elements value is updated as and when the underlying **observable** is changed. Element is processed only once, if **no observable** is used.

Example

Let us take a look at the following example which demonstrates the use of html binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Html binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p><span data-bind="html: welcomeMessgae "></span></p>
  <script>
```

```
function AppViewModel() {  
    this.welcomeMessgae = ko.observable();  
    this.welcomeMessgae ("<strong>Welcome to TutorialsPoint !!! For free  
online tutorials and courses click <a  
href='http://tutorialspoint.com/'>here</a>.</strong>");  
}  
ko.applyBindings(new AppViewModel());  
</script>  
</body>  
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **html-bind.htm** file.
- Open this HTML file in a browser.

Welcome to TutorialsPoint !!! For free online tutorials and courses click [here](#).

CSS Binding

This binding allows you to define CSS classes for the HTML DOM elements based on certain condition. This is useful in case you need to highlight some data depending on a situation.

Syntax

```
css: <binding-object>
```

Parameters

- In case of static CSS binding, the parameter can be in the form of JavaScript Object, consisting of property and its value.

- **Property** here refers to CSS class to be applied and **value** can be Boolean true, or false, or JavaScript expression or a function.
 - **Classes** can be also applied dynamically using Computed Observable functions.
- Multiple CSS classes can also be applied at once. Following is an example of how 2 classes are added to binding.

```
<div data-bind="css: { outOfStock : productStock() === 0,
discountAvailable: discount }">
```

- Class names can also be specified in single quotes such as 'discount Available'.
- 0 and null are treated as false value. Numbers and other objects are treated as true value.
- If the ViewModel property is an observable, then CSS classes are decided depending on the new updated parameter value. If it is not an observable value, then classes are determined only once for the first time.

Example

Let us take a look at the following example which demonstrates the use of CSS binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS CSS binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
  <style>
    .outOfStock {
      color: red;
      font-weight: bold;
    }
  </style>
</head>
<body>
  <div data-bind="css: { outOfStock : productStock() === 0 }">
    Product Details.
  </div>

  <script>
    function AppViewModel() {
```

```
        this.productStock = ko.observable(0);
    }
    var vm = new AppViewModel();
    ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **css-bind.htm** file.
- Open this HTML file in a browser.
- Product Information is shown in a normal way when the productStock property is above 0. Product Information becomes red and bold. once productStock becomes 0.

Product Details.

Style Binding

Style Binding allows you to apply inline styling to HTML DOM element by manipulating the element's style attribute instead of applying CSS classes. This binding requires key-value pair due to inline styling.

Syntax

```
style: <binding-object>
```


Parameters

- JavaScript object should be passed as a parameter in which the property name refers to style attribute and values refer to the desired values to be applied on the elements.
- Multiple styles can also be applied at once. Suppose you have a discount property in your ViewModel and you want to add that, then the code will look like the following -

```
<div data-bind="style: { color: productStock() < 0 ? 'red' : 'blue',
fontWeight: discount() ? 'bold' : 'normal' }">
```

If the productStock is not available, then the font becomes red. Else, it becomes blue. If the discount is set to true, then Product Details will become bold font. Else, it will remain in normal font.

- If the ViewModel property is observable, then styles are applied depending on the new updated parameter value. If it is not an observable value, then style is applied only once for the first time.

Example

Let us take a look at the following example which demonstrates the use of style Binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS style binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <div data-bind="style: { color: productStock() < 0 ? 'red' : 'black' }">
    Product Details.
  </div>

  <script type="text/javascript">
    function AppViewModel() {
      this.productStock= ko.observable(30000) // initially black as positive value
      this.productStock(-10); // now changes DIV's contents to red
    };
    var vm = new AppViewModel();
    ko.applyBindings(vm);
```

```

    </script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **style-bind.htm** file.
- Open this HTML file in a browser.
- If productStock goes below 0, then Product Details becomes red. Else, if stock is available it becomes black.

Product Details.

Observations

Illegal JavaScript variable names

CSS style name **font-weight** is not allowed in JavaScript. Instead, write it like **fontWeight** (Hyphen in variable names is not allowed in JavaScript).

[Click here](#) for all JavaScript style attributes, which are also available at KO's official site.

Attr Binding

This binding allows you to dynamically assign HTML elements **attribute** using ViewModel property. For example, you can set **src** attribute for an image, **title** attribute for HTML page, or a **href** for a link in the tag based on values in ViewModel.

Syntax

```
attr: <binding-object>
```

Parameter

- JavaScript object should be passed as a parameter in which the property name refers to attribute name and values refer to the desired values to be passed to DOM element.
- Multiple attributes can also be assigned at once. Suppose you want to assign a title and href a value, then binding will look like the following -

```
<a data-bind="attr: { href: courseUrl, title: courseTitle}">
```

courseUrl and **courseTitle** variables will have the desired values in ViewModel.

- If the ViewModel property is an observable value, then attribute is assigned depending on the new updated parameter value. If it is not an observable value, then attribute is assigned only once for the first time.

Example

Let us take a look at the following example which demonstrates the use of Attr binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS attribute binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <a data-bind="attr: { href: courseUrl}">
    Click here for free online tutorials and courses.
  </a>

  <script type="text/javascript">
    function AppViewModel(){
      this.courseUrl= ("http://tutorialspoint.com/");
    };
    var vm = new AppViewModel();
    ko.applyBindings(vm);
  </script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **attr-bind.htm** file.
- Open this HTML file in a browser.
- The courseUrl will assign the value to href attribute in HTML DOM element.

[Click here for free online tutorials and courses.](#)

Working with Control Flow Bindings

Following is a list of Control Flow Binding types provided by KO.

Sr. No.	Binding Type & Usage
1	<u>foreach: <binding-array></u> In this binding, each array item is referenced in HTML markup in a loop.
2	<u>if: <binding-condition></u> If the condition is true, then the given HTML markup will be processed. Else, it will be removed from DOM.
3	<u>ifnot: <binding-condition></u> Negation of If. If the condition is true, then the given HTML markup will be processed. Else, it will be removed from DOM.
4	<u>with: <binding-object></u> This binding is used to bind the child elements of an object in the specified object's context.
5	<u>component: <component-name> OR component: <component-object></u> This binding is used to insert a component into DOM elements and pass the parameters optionally.

Foreach Binding

In this binding, each array item is referenced in HTML markup in a loop. This is very useful while populating a list or table data. **foreach** can be nested along with other control flow bindings.

Syntax

```
foreach: <binding-array>
```

Parameters

- Array name is passed as a parameter. HTML markup is processed for each item in a loop.
- A JavaScript object literal can be passed as a parameter and can be iterated over using a property called data.
- If the parameter is an observable one, then any modifications made are reflected in the UI as soon as the underlying observable changes.

Example

Let us take a look at the following example, which demonstrates the use of foreach binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS foreach binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>List of courses available:</p>
  <div data-bind="foreach: courseArray ">
    <li data-bind="text: $data"><span data-bind="text: $index"></span></li>
  </div>

  <script type="text/javascript">
    function AppViewModel() {
      this.courseArray = ([ 'JavaScript', 'KnockoutJS', 'BackboneJS', 'EmberJS' ]);
      this.courseArray.push('HTML');
    };
```

```

    var vm = new AppViewModel();
    ko.applyBindings(vm);
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **foreach-bind.htm** file.
- Open this HTML file in a browser.

List of courses available:

- JavaScript
- KnockoutJS
- BackboneJS
- EmberJS
- HTML

You can rewrite the above code using **as** keyword. Just change the binding line as shown in the following code.

```
<div data-bind="foreach: {data: courseArray, as : 'cA' }">
```

Example

Let's take a look at another example to populate the list details using Observable Array.

```

<!DOCTYPE html>
<head>
    <title>KnockoutJS foreach binding</title>
    <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
    type="text/javascript"></script>
</head>
<body>

    <p>List of product details:</p>

```

```
<ul data-bind="foreach: productArray ">
  <li>
    <span data-bind="text: productName"></span> <a href="#" data-
bind="click: $parent.removeProduct">Remove </a>
  </li>
</ul>

<script type="text/javascript">
function AppViewModel() {
  self=this;
  self.productArray = ko.observableArray([
    {productName: 'Milk'},
    {productName: 'Oil'},
    {productName: 'Shampoo'}  ]);

  self.removeProduct = function(){
    self.productArray.remove(this);
  }
};
var vm = new AppViewModel();
ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **foreach-bind-using-observable.htm** file.
- Open this HTML file in a browser.
- Item from the list is removed dynamically when you click Remove link.

List of product details:

- Milk [Remove](#)
- Oil [Remove](#)
- Shampoo [Remove](#)

Observations

Utilizing foreach without container

There might be some situation where it is not possible to use the container to include foreach in it. In that situation, container-less format can be used to invoke binding.

```
<ul>
  <!-- ko foreach: productArray -->
  <li>
    <span data-bind="text: productName"></span> <a href="#" data-bind="click:
    $parent.removeProduct">Remove </a>
  <!-- /ko -->
  </li>
</ul>
```

The <!--ko--> and <!--/ko--> works as start and end markers making it a virtual syntax and binds data as if it is a real container.

Handling destroyed items in array

Array items can be destroyed (made hidden currently and removed later) using **destroy** array function provided by KO. These items are not shown in foreach and are hidden automatically. To see these hidden items, KO provides a method called **includeDestroyed**. Hidden items are shown if this parameter is set to Boolean true.

```
<div data-bind="foreach: {data: courseArray, includeDestroyed: true }">
  ...
  ...
  ...
</div>
```


If Binding

This binding allows you to present the conditionally. If the specified condition is true, then show data, else don't show.

if binding is similar to **visible** binding. Difference being, in visible binding the underlying HTML markup actually stays on DOM and is made visible based on the condition whereas in **if** binding, HTML markup is added or removed from DOM based on the condition.

Syntax

```
if: <binding-condition>
```

Parameters

- Parameter is a condition you want to evaluate. If the condition evaluates to true or true-like value, then the given HTML markup will be processed. Else, it will be removed from DOM.
- If the condition in the parameter contains an observable value, then the condition is re-evaluated whenever the observable value changes. Correspondingly, related markup will be added or removed based on the condition result.

Example

Let us take a look at the following example which demonstrates the use of if binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS if binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p><strong>Product details</strong></p>
  <table border="1">
    <thead>
      <th>Product Name</th><th>Price</th><th>Nature</th>
    </thead>
    <tbody data-bind="foreach: productArray ">
      <tr>
        <td><span data-bind="text: productName"></span></td>
        <td><span data-bind="text: price"></span></td>
        <td data-bind="if: $data.price > 100 ">Expensive</td>
```

```

        </tr>
    </tbody>
</table>

<script type="text/javascript">
    function AppViewModel() {
        self=this;
        self.productArray = ko.observableArray([
            {productName: 'Milk', price: 100},
            {productName: 'Oil', price: 10},
            {productName: 'Shampoo', price: 1200}  ]);
    };
    var vm = new AppViewModel();
    ko.applyBindings(vm);
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **if-bind.htm** file.
- Open this HTML file in a browser.
- This example will populate the third column which talks about the products' nature (expensive or not) depending on the price. Note that the individual property is accessed using \$data binding context.

Product details

Product Name	Price	Nature
Milk	100	
Oil	10	
Shampoo	1200	Expensive

Observations

Container-less if

There might be a situation when it is not possible to place data-binding inside a DOM element. Essential checking can still be performed with the help of **container-less** syntax based on the comment tags shown as follows.

The `<!--ko-->` and `<!--/ko-->` works as start and end markers making it a virtual syntax and binds the data as if it is a real container.

Example

Let's us take a look at the following example which demonstrates the use of container-less syntax.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS if binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>

  <ul>
    <li>Monday</li>
    <li>Tuesday</li>
    <li>Wednesday</li>
    <li>Thursday</li>
    <li>Friday</li>
    <!-- ko {if: weekend} -->
    <li>Saturday - check if it is weekend.</li>
    <li>Sunday</li>
    <!-- /ko -->
  </ul>
<script>

  function AppViewModel() {
    this.weekend = false;
  }
  var vm = new AppViewModel();
```

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

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **if-container-less.htm** file.
- Open this HTML file in a browser.

- Monday
- Tuesday
- Wednesday
- Thursday
- Friday

Ifnot Binding

Ifnot binding is the negating of if binding. It is just another flavor of if binding.

Syntax

```
ifnot: <binding-condition>
```

Parameters

- Parameter is a condition you want to check. If the condition evaluates to true or true-like value, then the given HTML markup will be processed. Else, it will be removed from DOM.
- If the condition in the parameter contains an observable value, then the condition is re-evaluated whenever the observable value changes. Correspondingly, the related markup will be added or removed based on the condition result.

Example

Let us take a look at the following example which demonstrates the use of ifnot binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS ifnot binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <p><strong>Product details</strong></p>
  <table border="1">
    <thead>
      <th>Product Name</th><th>Price</th><th>Nature</th>
    </thead>
    <tbody data-bind="foreach: productArray ">
      <tr>
        <td><span data-bind="text: productName"></span></td>
        <td><span data-bind="text: price"></span></td>
        <td data-bind="ifnot: $data.price < 200 ">Expensive</td>
      </tr>
    </tbody>
  </table>
  <script type="text/javascript">
    function AppViewModel() {
      self=this;
      self.productArray = ko.observableArray([
        {productName: 'Milk', price: 100},
        {productName: 'Oil', price: 10},
        {productName: 'Shampoo', price: 1200}  ]);
    };
    var vm = new AppViewModel();
    ko.applyBindings(vm);
  </script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **if-not-bind.htm** file.
- Open this HTML file in a browser.
- This example will populate the third column which talks about the products' nature (expensive or not) depending on the price. Note that the individual property is accessed using \$data binding context.

Product details

Product Name	Price	Nature
Milk	100	
Oil	10	
Shampoo	1200	Expensive

with Binding

This binding is used to bind the child elements of an object in the specified object's context. This binding can also be nested with other type of bindings such as if and foreach.

Syntax

```
with: <binding-object>
```

Parameters

- Pass the object which you want to use as context for binding child elements as the parameter. Child elements will not be shown if the object or expression passed is evaluated to be null or undefined.
- The expression will be re-evaluated, if the parameter provided is an observable one. Correspondingly, the child elements will be re-processed based on the refreshed context result.

Example

Let us take a look at the following example which demonstrates the use of with binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS with binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
</head>
<body>
  <h1 data-bind="text: siteName"> </h1>
  <table border="1">
    <thead>
      <th>Course Name</th><th>Fees</th><th> Start Date</th>
    </thead>
    <tbody data-bind="with: courses ">
      <tr>
        <td><span data-bind="text: courseName"></span></td>
        <td><span data-bind="text: Fees"></span></td>
        <td><span data-bind="text: startDate"> </span></td>
      </tr>
    </tbody>
  </table>

  <script type="text/javascript">
    function AppViewModel() {
      self=this;
      self.siteName =ko.observable( 'TutorialsPoint');
      self.courses =ko.observable(
        {courseName: 'Microsoft .Net', Fees: 20000, startDate: '20-Mar-2016'});
    };
    var vm = new AppViewModel();
    ko.applyBindings(vm);
  </script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **with-bind.htm** file.
- Open this HTML file in a browser.

TutorialsPoint

Course Name	Fees	Start Date
Microsoft .Net	20000	20-Mar-2016

Observations

Container-less with

There might be a situation when it is not possible to place data-binding inside a DOM element. Essential binding can still be performed with the help of **container-less** syntax based on the comment tags as shown in the following code.

```
<ul>
  <li>Course Info</li>
  <!-- ko with: currentClasses -->
    ...
  <!-- /ko -->
  <!-- ko with: plannedClasses -->
    ...
  <!-- /ko -->
</ul>
```

The `<!--ko-->` and `<!--/ko-->` works as start and end markers making it a virtual syntax and binds the data as if it is a real container.

component Binding

This binding is used to insert a component into DOM elements and pass the parameters optionally. This binding can be achieved in the following two ways:

- Shorthand Syntax
- Full syntax

Shorthand Syntax

In this approach only the component name is specified without specifying any parameters.

Syntax

```
<div data-bind='component: "component-name"'></div>
```

The parameter value passed can be an observable. Hence, whenever the observable changes, the old component instance will be disposed and the new one will be created according to the refreshed parameter value.

Full Syntax

This approach accepts the parameter in the form of an object.

Syntax

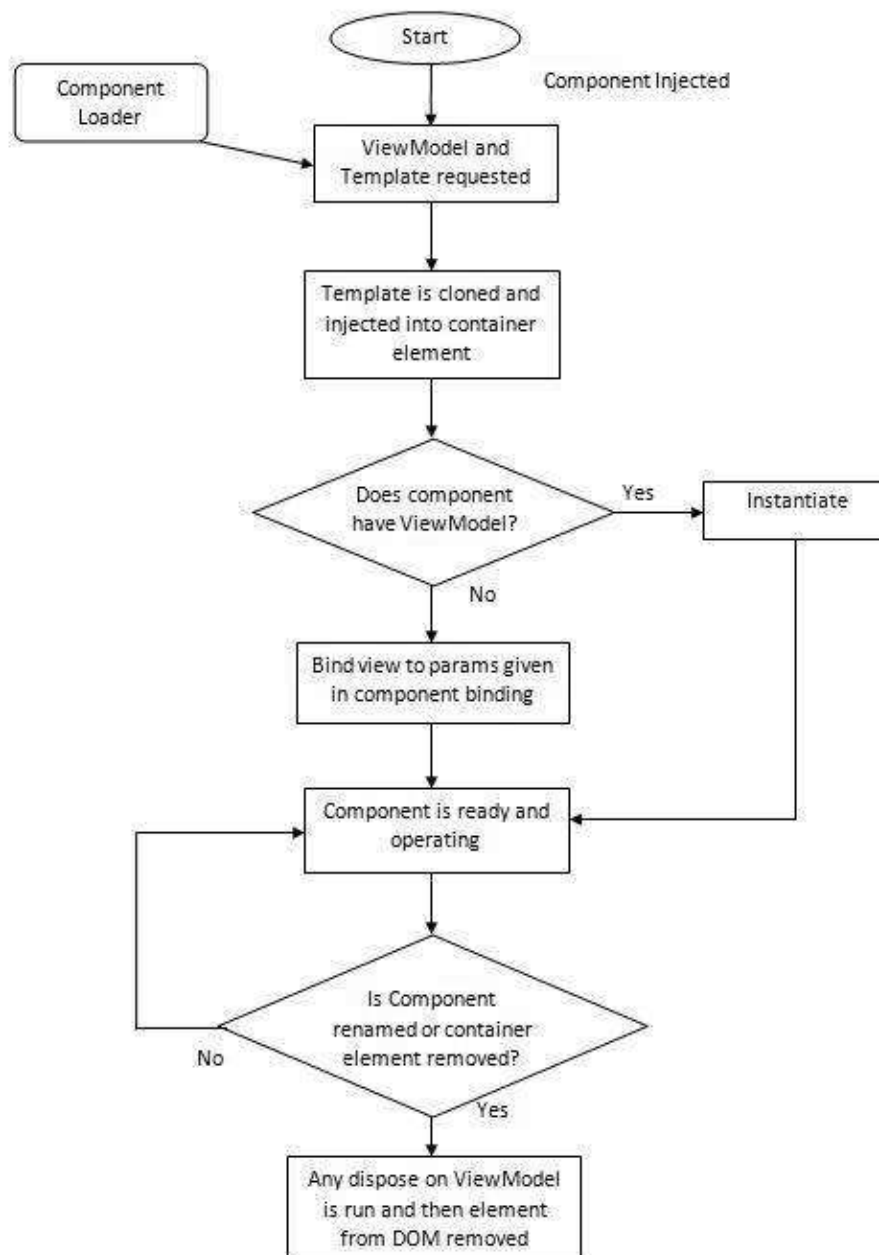
```
<div data-bind='component: {  
  name: "component-name",  
  params: { param1: value1, param2:value2 ...}  
'></div>
```

The object is made up of the following two items:

- **name** - This is the name of the component to be inserted. As mentioned earlier, this can be an observable.
- **params** - This is an object to be passed to the component. Mostly, this will be a key-value object including multiple parameters. This is most of the times received by the constructor of a ViewModel.

Component Processing Workflow

The following diagram explains the process that takes place when a component is injected by component binding.



Let us look at the process in detail:

Receive ViewModel factory and template from component loaders - The registered ViewModel and template is requested and received by the default loader. By default, this is an asynchronous process.

Clone the component template - In this step, cloning of component template and inserting it into DOM element takes place. Existing content, if any, will be removed.

Instantiate a ViewModel if any - In this step, ViewModel is instantiated. If the ViewModel is provided as a constructor function, then KO calls.

```
new ViewModelName(params)
```

If the ViewModel is provided in factory function format, i.e. createViewModel then KO calls.

```
createViewModel(params, yourcomponentInfo)
```

Here yourcomponentInfo.element is the element where the template is inserted.

Bind ViewModel to view - In this stage, ViewModel is bound to View. If the ViewModel is not provided, then binding is done with parameters mentioned in component binding.

Now component is ready - At this stage, the component is ready and in functioning form. The component keeps an eye on the parameters which are observable, if any, so as to write the changed values.

Dispose the ViewModel if the component is lost - The ViewModel's dispose function is called, if the component binding name value is changed observably or some control-flow binding removes the DOM element container itself, which was meant to hold the component output. The dispose takes place just before any element container is removed from DOM.

Example

Let us take a look at the following example which demonstrates the use of component binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Component binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>
  <h4>Component binding without parameters</h4>
  <div data-bind='component: "calculate-sum"'></div>

  <h4>Component binding passing parameters</h4>
  <div data-bind='component: {
    name: "calculate-sum",
    params: { number1: 2, number2: 3 }
  }'></div>

  <script>
    ko.components.register('calculate-sum', {
      viewModel: function(params) {
        this.number1 = ko.observable(params && params.number1);
```

```

        this.number2 = ko.observable(params && params.number2);

        this.result = ko.computed(function(){
            var sum = Number(this.number1()) + Number(this.number2());
            if ( isNaN(sum) )
                sum = 0;
            return sum;
        },this);
    },
    template: 'Enter Number One: <input data-bind="value: number1" /> <br>
<br>'+
        ' Enter Number Two: <input data-bind="value: number2" /> <br> <br>'+
        ' Sum = <span data-bind="text: result" />'
    ));

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

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **component-bind.htm** file.
- Open this HTML file in a browser.
- Enter the numbers in both textboxes and observe that the sum is calculated.

Component binding without parameters

Enter Number One:

Enter Number Two:
Sum = 0

Component binding passing parameters
Enter Number One:
Enter Number Two:
Sum = 5

Observations

Template Only Components

Components can be created without ViewModel. The component can just comprise of a template shown as follows.

```
ko.components.register('my-component', {
  template: '<div data-bind="text: productName"></div>'
});
```

And DOM will look like -

```
<div data-bind='component: {
  name: "my-component",
  params: { productName: someProduct.name }
}'></div>
```

Using Component binding without DOM container element

There might be situation where it is not possible to insert a component into DOM element. Essential binding can still be performed with the help of **container-less** syntax based on the comment tags shown as follows.

The `<!--ko-->` and `<!--/ko-->` works as start and end markers making it a virtual syntax and binds the data as if it is a real container.

Memory management and Disposal

Disposal function can be added optionally as part of ViewModel. If this function is included, then it will be invoked whenever the component is lost or the container element itself is removed. It is a good practice to use dispose function to release resources occupied by

unwanted objects, which are not garbage collectable by default. Following are a few examples:

- setInterval method will keep on running until cleared explicitly. clearInterval (handle) is used to stop this process.
- ko.computed properties need to be disposed explicitly. Else, they might still continue receiving notifications from their underlying observables. Manual disposal can be avoided using pure computed functions.
- Make sure to use dispose() method on the subscription, else it keeps on firing the changes until disposed.
- Custom event handlers created on DOM elements and created inside createViewModel needs to be disposed.

Working with Form Fields Bindings

Following is the list of Form Fields Binding types provided by KO.

Sr. No.	Binding Type & Usage
1	<u>click: <binding-function></u> This binding is used to invoke a JavaScript function associated with a DOM element based on a click.
2	<u>event: <DOM-event: handler-function></u> This binding is used to listen to the specified DOM events and call associated handler functions based on them.
3	<u>submit: <binding-function></u> This binding is used to invoke a JavaScript function when the associated DOM element is submitted.
4	<u>enable: <binding-value></u> This binding is used to enable certain DOM elements based on a specified condition.
5	<u>disable: <binding-value></u> This binding disables the associated DOM element when the parameter evaluates to true.
6	<u>value: <binding-value></u> This binding is used to link respective DOM element's value into ViewModel property.
7	<u>textInput: <binding-value></u>

	This binding is used to create 2-way binding between text box or textarea and ViewModel property.
8	<u>hasFocus: <binding-value></u> This binding is used to manually set the focus of a HTML DOM element through a ViewModel property.
9	<u>checked: <binding-value></u> This binding is used to create a link between a checkable form element and ViewModel property.
10	<u>options: <binding-array></u> This binding is used to define the options for a select element.
11	<u>selectedOptions: <binding-array></u> This binding is used to work with elements which are selected currently in multi list select form control.
12	<u>uniqueName: <binding-value></u> This binding is used to generate a unique name for a DOM element.

Click Binding

Click binding is one of the simplest binding and is used to invoke a JavaScript function associated with a DOM element based on a click. This binding works like an event handler.

This is most commonly used with elements such as **button**, **input**, and **a**, but actually works with any visible DOM element.

Syntax

```
click: <binding-function>
```

Parameters

The parameter here will be a JavaScript function which needs to be invoked based on a click. This can be any function and need not be a ViewModel function.

Example

Let us look at the following example which demonstrates the use of click binding.

```
<!DOCTYPE html>
```

```
<head>
  <title>KnockoutJS Click Binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

  <p>Enter your name: <input data-bind="value: someValue" /></p>
  <p><button data-bind="click: showMessage">Click here</button></p>

<script type="text/javascript">
  function ViewModel (){
    this.someValue = ko.observable();
    this.showMessage = function(){
      alert("Hello "+ this.someValue()+ "!!! How are you today?" + "\nClick
Binding is used here !!!");
    }
  };

  var vm = new ViewModel();
  ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **click-bind.htm** file.
- Open this HTML file in a browser.
- Click the Click here button and a message will be shown on the screen.

Enter your name:

Observations

Current Item can also be passed as a parameter

It is also possible to provide a current model value as a parameter when the handler function is called. This is useful when dealing with a collection of data, wherein the same action needs to be performed on a set of items.

Example

Let us look at the following example to understand it better.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Click binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>
  <p>List of product details:</p>
  <ul data-bind="foreach: productArray ">
    <li>
      <span data-bind="text: productName"></span>  <a href="#" data-
bind="click: $parent.removeProduct">Remove </a>
    </li>
  </ul>

  <script type="text/javascript">
    function AppViewModel() {
```

```
self=this;
self.productArray = ko.observableArray([
  {productName: 'Milk'},
  {productName: 'Oil'},
  {productName: 'Shampoo'}  ]);

self.removeProduct = function(){
  self.productArray.remove(this);
};

var vm = new AppViewModel();
ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **click-for-current-item.htm** file.
- Open this HTML file in a browser.
- **removeProduct** function is called every time the Remove link is clicked and is called for that particular item in array.
- Note that the \$parent binding context is used to reach the handler function.

List of product details:

- Milk [Remove](#)
- Oil [Remove](#)
- Shampoo [Remove](#)

Passing more parameters

DOM event along with the current model value can also be passed to the handler function.

Example

Let us take a look at the following example to understand it better.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Click Binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

  <p>Press Control key + click below button.</p>
  <p><button data-bind="click: showMessage">Click here to read message
</button></p>

<script type="text/javascript">
  function ViewModel (){
    this.showMessage = function(data,event){
      alert("Click Binding is used here !!!");
      if (event.ctrlKey) {
        alert("User was pressing down the Control key.");
      }
    }
  }
};

  var vm = new ViewModel();
  ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **click-bind-more-params.htm** file.
- Open this HTML file in a browser.
- Pressing of the control key is captured by this binding.

Press Control key + click below button.

Click here to read message

Allowing the default click action

KnockoutJS prevents click event to perform any default action by default. Meaning if Click binding is used on **<a>** tag, then the browser will only call the handler function and will not actually take you to the link mentioned in href.

If you want the default action to take place in click binding, then you just need to return true from your handler function.

Example

Let us look at the following example which demonstrates the default action performed by click binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Click Binding - allowing default action</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
  type="text/javascript"></script>
</head>
<body>

  <a href="http://www.tutorialspoint.com/" target="_blank" data-bind="click:
  callUrl">Click here to see
  how default Click binding works.
```

```
</a>

<script type="text/javascript">
    function ViewModel (){
        this.callUrl = function(){
            alert("Default action in Click Binding is allowed here !!! You are
redirected to link.");
            return true;
        }
    };

    var vm = new ViewModel();
    ko.applyBindings(vm);

</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **click-default-bind.htm** file.
- Open this HTML file in a browser.
- Click the link and a message will be shown on the screen. The URL mentioned in href opens in a new window.

[Click here to see how default Click binding works.](#)

Preventing the event from bubbling

KO will allow the click event to bubble up to the higher level event handlers. Meaning if you have 2 click events nested, then the click handler function for both of them will be called. If needed, this bubbling can be prevented by adding an extra binding called as clickBubble and passing false value to it.

Example

Let us look at the following example which demonstrates the use of clickBubble binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Click Binding - handling clickBubble</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

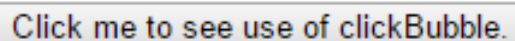
  <div data-bind="click: outerFunction">
    <button data-bind="click: innerFunction, clickBubble:false">
      Click me to see use of clickBubble.
    </button>
  </div>

<script type="text/javascript">
  function ViewModel (){
    this.outerFunction = function(){
      alert("Handler function from Outer loop called.");
    }
    this.innerFunction = function(){
      alert("Handler function from Inner loop called.");
    }
  };
  var vm = new ViewModel();
  ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **click-cllickbubble-bind.htm** file.
- Open this HTML file in a browser.
- Click the button and observe that adding of clickBubble binding with value false prevents the event from making it past innerFunction.



Event Binding

This binding is used to listen to specific DOM event and call associated with the handler function based on it.

Syntax

```
event: <{DOM-event: handler-function}>
```

Parameters

Parameter is inclusive of a JavaScript object, containing DOM event which will be listened to and a handler function which needs to be invoked based on that event. This function can be any JavaScript function and need not be necessarily ViewModel function.

Example

Let us take a look at the following example which demonstrates the use of event binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Event Binding</title>
```

```

    <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>
    <p>Enter Number :</p>
    <input data-bind="value: someValue , event: {keyup: showMessage},
valueUpdate: 'afterkeydown' " /><br><br>
    You Entered: <span data-bind="text: someValue"/>

<script type="text/javascript">
    function ViewModel (){
    this.someValue = ko.observable();
    this.showMessage = function(data,event){
        if ((event.keyCode < 47) || (event.keyCode > 58 )) { //check for number
            if (event.keyCode !== 8) //ignore backspace
                alert("Please enter a number.");
            this.someValue('');
        }
    }
    };
    var vm = new ViewModel();
    ko.applyBindings(vm);
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **event-bind.htm** file.
- Open this HTML file in a browser.
- Try to key in any non-numeric value and you will be prompted with an alert.

Enter Number :

You Entered:

Observations

Passing a current item as a parameter to the handler function

KO will pass the current item as parameter when calling the handler function. This is useful when working with a collection of items and need to work on each of them.

Example

Let us take a look at the following example in which the current item is passed in event binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Event Binding - passing current item </title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>
  <ul data-bind="foreach: icecreams">
    <li data-bind="text: $data, event: { mouseover: $parent.logMouseOver }">
</li>
  </ul>
  <p>You seem to be interested in: <span data-bind="text: flavorLiked">
</span></p>

<script type="text/javascript">
  function ViewModel (){
    var self=this;
```

```
self.flavorLiked = ko.observable();
self.icecreams = ko.observableArray(['Vanilla', 'Pista', 'Chocolate', 'Mango']);

// current item is passed here as the first parameter, so we know which
// flavor was hovered over
self.logMouseOver = function (flavor) {
    self.flavorLiked(flavor);
}
};
var vm = new ViewModel();
ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **event-bind-passing-curr-item.htm** file.
- Open this HTML file in a browser.
- Flavor, which has the mouse over it, is displayed.
- Note that self as an alias is used for this. This helps to avoid any problems with this being redefined to something else in event handlers.

- Vanilla
- Pista
- Chocolate
- Mango

You seem to be interested in: Vanilla

Passing more parameters, or referring the event object

There might be a situation where you need to access DOM event object associated with the event. KO passes the event as a second parameter to the handler function.

Example

Let us take a look at the following example in which the event is sent as a second parameter to function.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Event Binding - passing more params</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>
  <div data-bind="event: { mouseover: logMouseOver }">
    Press shiftKey + move cursor over this line.
  </div>

  <script type="text/javascript">
    function ViewModel (){
      self.logMouseOver = function (data, event) {
        if (event.shiftKey){
          alert("shift key is pressed.");
        }
        else {
          alert("shift key is not pressed.");
        }
      }
    };
    var vm = new ViewModel();
    ko.applyBindings(vm);
  </script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **event-bind-passing-more-params.htm** file.
- Open this HTML file in a browser.
- Press shiftKey + move cursor to the text. Observe that the message will pop up showing if you have pressed the shiftKey.

Press shiftKey + move cursor over this line.

Allowing default action

Knockout will avoid the event from performing any default action, by default. Meaning if you use keypress event for an input tag, then KO will just call the handler function and will not add the key value to input elements value.

If you want the event to perform a default action, then just return true from the handler function.

Example

Let us look at the following example which allows default action to take place.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Event Binding - allowing default action</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
  type="text/javascript"></script>
</head>
<body>
  <p>Enter the flavor you like from available menu: (Vanilla, Pista,
  Chocolate, Mango)</p>
  <input data-bind="event: { keypress: acceptInput}"></input>
```

```
<script type="text/javascript">
    function ViewModel (){
        self.acceptInput = function () {
            return true;
        }
    };
    var vm = new ViewModel();
    ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **event-bind-default-action.htm** file.
- Open this HTML file in a browser.
- Any key pressed is actually shown in the input box because the handler function returns true.

Enter the flavor you like from available menu: (Vanilla, Pista, Chocolate, Mango)

Preventing the event from bubbling

KO will allow the event to bubble up to the higher level event handlers. Meaning if you have two mouseover events nested, then the event handler function for both of them will be called. If needed, this bubbling can be prevented by adding an extra binding called as `youreventBubble` and passing false value to it.

Example

Let us take a look at the following example in which bubbling is handled.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Event Binding - preventing bubbling </title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>
  <div data-bind="event: { mouseover: myParentHandler }">
    <button data-bind="event: { mouseover: myChildHandler }, mouseoverBubble:
false">
      Click me to check bubbling.
    </button>
  </div>

  <script type="text/javascript">
    function ViewModel (){
      var self= this;
      self.myParentHandler  = function () {
        alert("Parent Function");
      }

      self.myChildHandler   = function () {
        alert("Child Function");
      }
    };
    var vm = new ViewModel();
    ko.applyBindings(vm);
  </script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **event-bind-prevent-bubble.htm** file.
- Open this HTML file in a browser.
- Move the mouseover button and you will see a message. Bubbling is prevented due to the use of mouseoverBubble set to false.

Click me to check bubbling.

Submit Binding

This binding is used to invoke a JavaScript function when the associated DOM element is submitted. This binding is used mostly for form elements.

The form is not actually submitted to the server when submit binding is used. KO prevents browsers default action. If you want submit binding to work as real submit element, then return true from your handler function.

Syntax

```
submit: <binding-function>
```

Parameters

- The binding function here will be the main function which needs to be invoked after submit event.
- This function can be any JavaScript function and need not be necessarily ViewModel function.

Example

Let us take a look at the following example which demonstrates the use of submit binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Submit Binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>
  <form data-bind="submit: addition">
    <p>Enter first number: <input data-bind="value: var1" /></p>
    <p>Enter second number: <input data-bind="value: var2" /></p>
    <p><button type="submit" >Click here for addition</button></p>
  </form>

  <script type="text/javascript">
    function ViewModel (){
      self = this;
      self.var1 = ko.observable(10);
      self.var2 = ko.observable(30);
      self.var3 = ko.observable(0);

      this.addition = function(){
        self.var1(Number(self.var1()));
        self.var2(Number(self.var2()));
        this.var3 = self.var1() + self.var2();
        alert("Addition is = "+ this.var3 );
      };
    };

    var vm = new ViewModel();
    ko.applyBindings(vm);
  </script>
</body>
</html>
```


Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **submit-bind.htm** file.
- Open this HTML file in a browser.
- This program adds 2 numbers. In KO, any accepted data from UI is considered in String format by default. Hence, it needs to be converted to Number format in case of Numeric operation.

Enter first number:

Enter second number:

Please refer to click binding for additional notes such as passing extra parameters, etc. All notes on that page also apply to submit binding.

Enable Binding

This binding is used to enable certain DOM element based on specified condition. This is useful with form elements such as **input**, **select**, and **textarea**.

Syntax

```
enable: <binding-value>
```

Parameters

- Parameter consists of Boolean like value which decides whether the element should be enabled or not. Element is enabled, if the parameter is true or true like value.
- Non-Boolean values are considered as loosely Boolean values. Meaning 0 and null are considered as false-like value, and Integer and non null objects are considered as true-like value.
- If the condition in the parameter contains any observable value, then the condition is re-evaluated whenever observable value changes. Correspondingly, related markup will be enabled based on the condition result.

Example

Let us take a look at the following example which demonstrates the use of enable binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Enable Binding</title>

  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
  type="text/javascript"></script>
</head>
<body>

  <p> Enter your feedback here:<br><br>
  <textarea rows=5 data-bind="value: hasFeedback, valueUpdate: 'afterkeydown'"
></textarea></p>
  <p><button data-bind="enable: hasFeedback">Save Feedback</button></p>

<script type="text/javascript">
  function ViewModel (){
    hasFeedback = ko.observable('');
  };

  var vm = new ViewModel();
  ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **enable-bind.htm** file.
- Open this HTML file in a browser.
- The save button is enabled only when the user has entered a feedback.

Enter your feedback here:

Save Feedback

Using random expressions to implement enable binding

You can also use a random expression to decide whether the element should be enabled or not.

Example

Let us take a look at the following example which demonstrates the use of random expression to invoke enable binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Enable binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
  type="text/javascript"></script>
</head>
<body>
  <p>Below button will be enabled only when product stock is available.</p>
  <button data-bind="enable: productStock() > 0 ">
    Product Details
  </button>

  <script type="text/javascript">
    function AppViewModel() {
      this.productStock= ko.observable(-10);
    };
    var vm = new AppViewModel();
    ko.applyBindings(vm);
```

```
</script>  
</body>  
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **enable-random-bind.htm** file.
- Open this HTML file in a browser.
- The Product Details button is enabled only when the product stock is available.

Below button will be enabled only when product stock is available.

Product Details

Disable Binding

This binding is the negation of enable binding. This binding disables the associated DOM element when the parameter evaluates to true.

Syntax

```
disable: <binding-value>
```

Parameters

- Parameter consists of Boolean like value, which decides whether the element should be disabled or not. If the parameter is true or true-like value, then the element is disabled.
- Non-Boolean values are considered as loosely Boolean values. Meaning 0 and null are considered as false-like value and Integer and non-null objects are considered as true-like value.

- If the condition in parameter contains an observable value, then the condition is re-evaluated whenever the observable value changes. Correspondingly, related markup will be disabled based on the condition result.

Example

Let us take a look at the following example which demonstrates the use of disable binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Disable Binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

  <p> Enter your feedback here:<br><br>
  <textarea rows=5 data-bind="value: hasFeedback, valueUpdate: 'afterkeydown'"
></textarea></p>
  <p><button data-bind="disable: !(hasFeedback())">Save Feedback</button></p>

<script type="text/javascript">
  function ViewModel (){
    hasFeedback = ko.observable('');
  };

  var vm = new ViewModel();
  ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **disable-bind.htm** file.
- Open this HTML file in a browser.
- The save button is disabled when the user has not entered any feedback.

Enter your feedback here:

Save Feedback

You can also use a random expression to decide whether the element should be disabled or not.

Value Binding

This binding is used to link respective DOM element's value into ViewModel property. Mostly, this is used with elements such as **input**, **select**, and **textarea**. This is similar to text binding, the difference being, in value binding data can be changed by the user and the ViewModel will update it automatically.

Syntax

```
value: <binding-value>
```

Parameters

- HTML DOM element's value property is set to parameter value. Earlier values will be overwritten.
- If the parameter is an observable value, then the element's value is updated as and when the underlying observable is changed. Element is processed only once if no observable is used.
- valueUpdate is an extra parameter which can also be supplied for extra features. KO uses additional events to detect extra changes when valueUpdate parameter is used in binding. Following are some common events:
 - input - ViewModel is updated when the value of input element changes.
 - keyup - ViewModel is updated when the key is released by the user.
 - keypress - ViewModel is updated when the key is typed.
 - afterkeydown - ViewModel keeps on updating as soon as the user starts typing the character.

Example

Let us look at the following example which demonstrates the use of value binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Value Binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

  <p>Enter your name: <input data-bind="value: yourName, valueUpdate:
'afterkeydown'" /></p>
  <p>Your name is : <span data-bind="text: yourName" /></span>

<script type="text/javascript">
  function ViewModel (){
    this.yourName = ko.observable('');
  };

  var vm = new ViewModel();
  ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **value-bind.htm** file.
- Open this HTML file in a browser.
- The data entered in the textbox is updated immediately due to the use of valueUpdate.

Enter your name:

Your name is :

Observations

Receiving value updates immediately from inputs

If you want the input element to give immediate updates to your ViewModel, then use the `textInput` binding. It is better than `valueUpdate` options, taking into consideration the weird behavior of browsers.

Dealing with drop-down list (<select> elements)

KO supports drop-down list (<select> elements) in a special way. The value binding and options binding work together allowing you to read and write values, which are random JavaScript objects and not just String values.

Using `valueAllowUnset` with <select> elements

Using this parameter, it is possible to set the model property with value which does not actually exist in the select element. This way one can keep the default option as blank when the user views drop-down for the very first time.

Example

Let us take a look at the following example in which `valueAllowUnset` option is used.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Value Binding - working with drop-down lists</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

  <p>Select a City:
```



```
<select data-bind=" options: cities,
                    optionsCaption: 'Choose City...',
                    value: selectedCity,
                    valueAllowUnset: true"></select>

</p>

<script type="text/javascript">
    function ViewModel (){
        this.cities = ko.observableArray(['Washington D.C.', 'Boston',
        'Baltimore']);
        selectedCity = ko.observable('Newark')
    };
    var vm = new ViewModel();
    ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **value-bind-drop-down.htm** file.
- Open this HTML file in a browser.
- selectedCity is assigned with value which is not present in the list. This makes the drop-down blank for the first time.

Select a City:

Updating observable and non-observable property values

KO is able to create a two-way binding if you use value to link a form element to an Observable property, so that the changes between them are exchanged among them.

If you use a non-observable property (a plain String or a JavaScript expression), then KO will do the following:

- If you refer a simple property on ViewModel, KO will set the form element's initial state to property value. If the form element is changed, then KO will write back the new values to property but it cannot detect any changes in the property, thus making it a one-way binding.
- If you refer something which is not simple, such as the result of comparison or a function call then, KO will set the form element's initial state to that value but cannot write any more changes made to the form element by the user. We can call this as one-time value setter.

Example

Following code snippet shows the use of observable and non-observable properties.

```
<!-- Two-way binding. Populates textbox; syncs both ways. -->
<p>First value: <input data-bind="value: firstVal" /></p>

<!-- One-way binding. Populates textbox; syncs only from textbox to model. -->
<p>Second value: <input data-bind="value: secondVal" /></p>

<!-- No binding. Populates textbox, but doesn't react to any changes. -->
<p>Third value: <input data-bind="value: secondVal.length > 8" /></p>

<script type="text/javascript">
    function viewModel() {
        firstVal = ko.observable("hi there"), // Observable
        secondVal = "Wats up!!!"           // Not observable
    };
</script>
```

Using value binding with the checked binding

If you include value binding with the checked binding, then the value binding will behave like checkedValue option, which can be used with checked binding. It will control the value used for updating ViewModel.

textInput Binding

This binding is used to create two-way binding between text box or textarea and ViewModel property. The difference between this and value binding is that this binding makes immediate updates available from HTML DOM for various input types.

Syntax

```
textInput: <binding-value>
```

Parameters

- HTML DOM element's value property is set to parameter value. Earlier values will be overwritten.
- If the parameter is something other than a Number or a String (such as an object or an array), the displayed text is equivalent to a String format.
- If the parameter is an observable, then the elements value is updated as and when the underlying observable is changed. Element is processed only once, if no observable is used.
- In most of the situations textInput is preferred over value binding due to the capacity of textInput to provide live updates from DOM for every input type and the ability to handle weird behavior of browsers.

Example

Let us take a look at the following example which demonstrates the use of textInput binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS textInput Binding </title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>
  <p> Enter your reviews here: <br><br><textarea rows=5 data-bind="textInput:
someReview" ></textarea><br></p>
  <p> You entered : <span data-bind="text: someReview"/></p>

  <script type="text/javascript">
    function ViewModel (){
      this.someReview = ko.observable('');
```

```
};  
var vm = new ViewModel();  
ko.applyBindings(vm);  
  
</script>  
</body>  
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **textInput-bind.htm** file.
- Open this HTML file in a browser.
- The data entered in textarea is updated in ViewModel immediately.

Enter your reviews here:

You entered :

Observations

textInput vs Value binding

textInput binding provides immediate live updates. The main differences between textInput and value Binding are:

Immediate updates - By default, the value binding only updates the model when the user moves the focus out of the textbox. The textInput binding updates the model instantly after each keystroke or on other text entry mechanism.

Browser event weirdness handling - Browsers are highly unpredictable in the events that fire in response to the unusual text entry mechanism such as dragging, cutting, or allowing auto-complete suggestion. The value binding does not handle all text entry cases on all browsers.

The textInput binding is especially designed to handle a wide range of weird behavior of browsers. This way it provides consistent and instant model updates, even in case of unusual text entry mechanisms.

hasFocus Binding

This binding is used to manually set the focus of a HTML DOM element through a ViewModel property. This is also a two-way binding method. When the element is focused from UI, Boolean value of ViewModel property is also changed and vice versa.

Syntax

```
hasFocus: <binding-value>
```

Parameters

- If the parameter evaluates to true or true-like value (such as Integer or non-null objects or non-empty string) then the DOM element is focused, else it is unfocused.
- When the element is focused or unfocused by the user manually, the Boolean ViewModel property is also changed accordingly.
- If the parameter is observable, then the elements value is focused or unfocused as and when the underlying observable is changed. Element is processed only once, if no observable is used.

Example

Let us take a look at the following example which demonstrates the use of hasFocus binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS hasFocus Binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

  <p>Enter primary contact number : <input data-bind="
    value: primaryContact,
    hasFocus: contactFlag,
    style: { 'background-color': contactFlag() ? 'pink' : 'white' } " /></p>
```

```
<button data-bind="click: setFocusFlag">Set Focus</button>

<script type="text/javascript">
    function ViewModel (){
        this.primaryContact = ko.observable();
        this.contactFlag = ko.observable(false);

        this.setFocusFlag = function(){
            this.contactFlag(true);
        }
    };

    var vm = new ViewModel();
    ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **hasfocus-bind.htm** file.
- Open this HTML file in a browser.
- Click Set Focus button to set the focus on the textbox.
- The background color of the textbox is changed, when the focus is set on it.

Enter primary contact number :

checked Binding

This binding is used to create a link between a checkable form element and ViewModel property. Mostly these form elements are inclusive of check box and radio buttons. This is also a two-way binding method wherein the moment the user checks form control, the respective ViewModel property is changed and vice versa.

Syntax

```
checked: <binding-value>
```

Parameters

Main Parameters

- The checkable element's state is set to parameter value. Earlier the value will be overwritten.
- Checkbox - The DOM element is checked when the ViewModel parameter value is true and is unchecked if it is false. Non-zero numbers, non-empty string, and non-null objects are interpreted at true Boolean value, whereas undefined, zero, and empty strings are considered as false value.
- Radio Buttons - Radio buttons work in a form of a String format. Meaning, KnockoutJS will set the elements value only when the parameter value matches exactly with Radio Button node's value. The property is set with the new value the moment the user selects a new Radio button value.
- If the parameter is an observable, then elements value is checked or unchecked as and when the underlying observable is changed. Element is processed only once if no observable is used.

Additional Parameters

- checkedValue - checkedValue option is used to hold the value used by the checkedbinding instead of the element's value attribute. This is very useful when the checked value is something other than a String (like an Integer or an object).

For example, take a look at the following code snippet where the item object themselves are included into chosenValue array, when the respective checkboxes are checked.

```
<!-- ko foreach: items -->
  <input type="checkbox" data-bind="checkedValue: $data, checked:
  $root.chosenValue" />
  <span data-bind="text: itemName"></span>
<!-- /ko -->
```

```

<script type="text/javascript">
    var viewModel = {
        itemsToBeSeen: ko.observableArray([
            { itemName: 'Item Number One' },
            { itemName: 'Item Number Two' }
        ]),
        chosenValue: ko.observableArray()
    };
</script>

```

If the checkedValue parameter is an Observable value, then the binding will update the checked model property whenever the underlying value changes. For radio buttons, KO will just update the model value. For checkboxes, it will replace the old value with the new value.

Example

Let us take a look at the following example which demonstrates the use of checkbox control.

```

<!DOCTYPE html>
<head>
    <title>KnockoutJS Checked checkbox Binding</title>
    <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>
    <p> The required files are installed. Please check below to complete
installation </p>
    <p><input type="checkbox" data-bind="checked: agreeFlag" />I agree to all
terms and conditions applied.</p>
    <button data-bind="enable: agreeFlag">Finish</button>

<script type="text/javascript">
    function ViewModel () {
        this.agreeFlag= ko.observable(false) // Initially unchecked
    };

    var vm = new ViewModel();
    ko.applyBindings(vm);

```



```

</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **checked-checkbox-bind.htm** file.
- Open this HTML file in a browser.
- The Finish button is activated only when the user agrees with the terms and conditions.

The required files are installed. Please check below to complete installation

☐ I agree to all terms and conditions applied.

Finish

Example

Let us take a look at the following example which demonstrates the use of radio-button control.

```

<!DOCTYPE html>
<head>
  <title>KnockoutJS Checked Radio Button Binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>
  <p> Select gender type from below:</p>
  <div><input type="radio" name="gender" value="Male" data-bind="checked:
checkGender" /> Male</div>
  <div><input type="radio" name="gender" value="Female" data-bind="checked:
checkGender" /> Female</div>

```

```
<div><p>You have selected: <span data-bind="text:checkGender
"></span></p></div>

<script type="text/javascript">
    function ViewModel () {
        checkGender= ko.observable("Male") // Initially male is selected
    };

    var vm = new ViewModel();
    ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **checked-radio-button-bind.htm** file.
- Open this HTML file in a browser.
- The radio button holds the gender type value.

Select gender type from below:

- ☒ Male
☐ Female

You have selected: Male

options Binding

This binding is used to define the options for a select element. This can be used for either drop-down list or a multi-select list. This binding cannot be used with anything other than <select> elements.

Syntax

options: <binding-array>

Parameters

- Parameter to be passed here is an array. For each entry in array, the option will be added for respective select node. Earlier option will be removed.
- If the parameter is an observable value, then the element's available options will be updated as and when the underlying observable is changed. Element is processed only once, if no observable is used.
- Additional Parameters
 - optionsCaption - This is just a default dummy value, which reads as 'Select item from below' or 'Choose from below'.
 - optionsText - This parameter allows you to specify which object property you want to set as text in the dropdown list. This parameter can also include a function, which returns the property to be used.
 - optionsValue - Similar to optionsText. This parameter allows to specify which object property can be used to set the value attribute of the option elements.
 - optionsIncludeDestroyed - Specify this parameter if you want to see array items which are marked as destroyed and are not actually deleted from observable array.
 - optionsAfterRender - Use this for running some custom logic on the existing option elements.
 - selectedOptions - This is used to read and write the selected options from a multi select list.
 - valueAllowUnset - Using this parameter, it is possible to set model property with the value which does not actually exist in the select element. This way one can keep default option as blank when the user views drop-down for the very first time.

Example

Let us take a look at the following example which demonstrates the use of options binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Options Binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

  <p>Tutorials Library:
    <select data-bind="
      options: availableTutorials,
      value: selectedTutorial,
      optionsCaption: 'Choose tutuorial...',
    "></select></p>

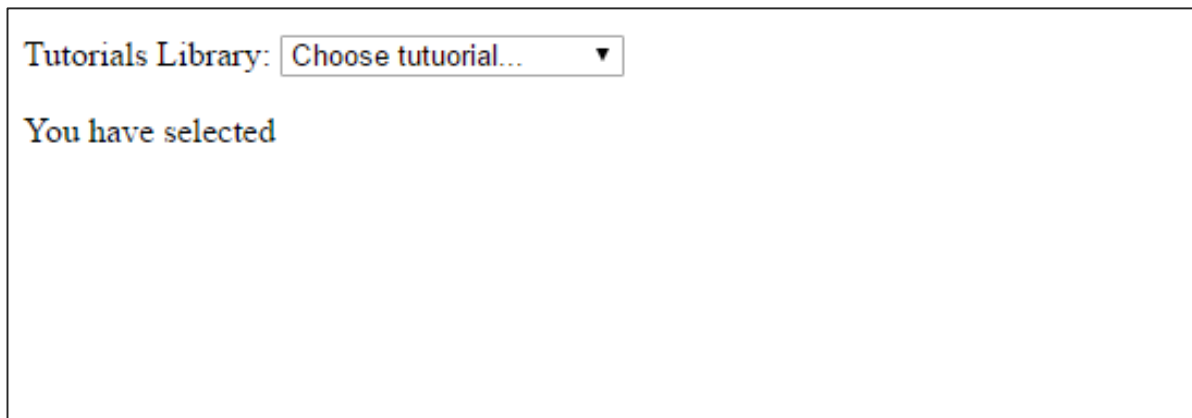
  <p>You have selected <b><span data-bind="text:selectedTutorial"></b></span>

<script type="text/javascript">
  function ViewModel (){
    this.selectedTutorial = ko.observable();
    this.availableTutorials = ko.observableArray(['Academic','Big
Data','Databases','Java Technologies','Mainframe',
  'Management','Microsoft Technologies','Mobile
Development','Programming','Software Quality']);
  };
  var vm = new ViewModel();
  ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **options-bind.htm** file.
- Open this HTML file in a browser.
- Note that the value binding is used to read the current selected item from the drop-down.



The screenshot shows a web application interface. At the top, it says "Tutorials Library:" followed by a dropdown menu with the text "Choose tutorial..." and a downward arrow. Below this, it says "You have selected" followed by a large, empty rectangular box, presumably for displaying the selected item.

Observations

Selection is preserved when setting/changing options

KO will leave the user's selection unchanged where possible, while the options binding updates the set of options in **<select>** element. For a single select in the drop-down list, the previously selected value will still be preserved. For multi select list, all previously selected options will be preserved.

Post-processing the generated options

The generated options can be post processed for some further custom logic with the help of **optionsAfterRender** callback. This function is executed after each element is inserted into the list, with the following parameters:

- The option element which is inserted.
- The data item against which it is bound; this will be undefined for the caption element.

Example

Let us take a look at the following example which uses `optionsAfterRender` to add a disable binding to each option.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Options Binding - using optionsAfterRender </title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

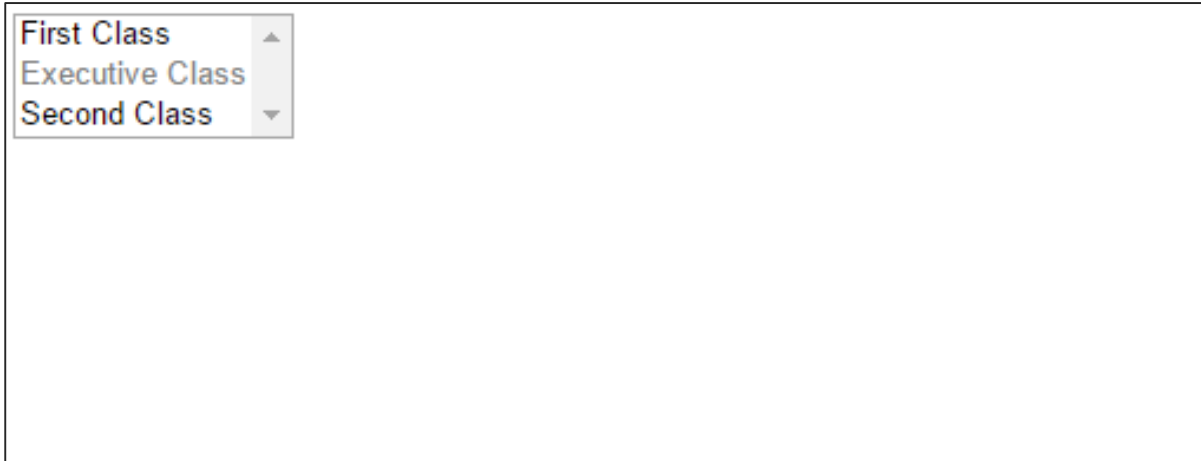
<select size=3 data-bind="
  options: myItems,
  optionsText: 'name',
  optionsValue: 'id',
  optionsAfterRender: setOptionDisable">
</select>

<script type="text/javascript">
  function ViewModel() {
    myItems = [
      { name: 'First Class', id: 1, disable: ko.observable(false)},
      { name: 'Executive Class', id: 2, disable: ko.observable(true)},
      { name: 'Second Class', id: 3, disable: ko.observable(false)}
    ];
    setOptionDisable = function(option, item) {
      ko.applyBindingsToNode(option, {disable: item.disable}, item);
    }
  };
  var vm = new ViewModel();
  ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **options-bind-optionsafterrender.htm** file.
- Open this HTML file in a browser.
- Option with id 2 is disabled using optionsAfterRender callback.



selectedOptions Binding

This binding is used to work with elements which are selected currently in the multi list select form control. This binding can be used with option binding and `<select>` form control only.

When the user selects or de-selects an item in the multi-select list, this adds or removes the corresponding value to an array on the view model. If it is an Observable array then then the items selected or de-selected from the UI are also updated in the array in ViewModel, making it a two-way binding method.

Syntax

```
selectedOptions: <binding-array>
```

Parameters

- Parameter here will be an array (can also be an Observable). The active items from select element are stored into this array. Earlier items will be overwritten.
- If the parameter is Observable array, then the items selected are updated as and when the underlying observable is changed. Element is processed only once, if no Observable array is used.

Example

Let us take a look at the following example which demonstrates the use of selectedOptions Binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS selectedOptions Binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

  <p>Tutorials Library:<br><br>
    <select size= 10 multiple='true' data-bind="
    options: availableTutorials,
    selectedOptions: selectedTutorials
    "></select></p>

  <p>(Press control and select for multiple options.)</p>
  <p>You have chosen below Tutorials:</p>
  <p><ul data-bind="foreach: selectedTutorials">
    <li>
      <span data-bind="text: $data"> </span>
    </li>
  </ul></p>

  <script type="text/javascript">
    function ViewModel() {
      self = this;

      self.availableTutorials = ko.observableArray(['Academic','Big
Data','Databases','Java Technologies','Mainframe',
      'Management','Microsoft Technologies','Mobile
Development','Programming','Software Quality']);

      self.selectedTutorials = ko.observableArray();
    };
    var vm = new ViewModel();
    ko.applyBindings(vm);
```



```

</script>
</body>
</html>

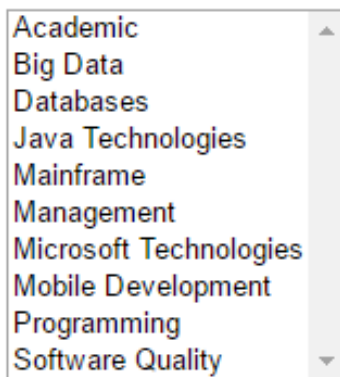
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **selectedoptions-bind.htm** file.
- Open this HTML file in a browser.
- selectedTutorials is an array to store the selected options.

Tutorials Library:



(Press control and select for multiple options.)

You have chosen below Tutorials:

uniqueName Binding

This binding is used to generate a unique name for a DOM element. If the DOM element did not have a name attribute, this binding gives it one and sets it to some unique string value.

You won't need to use this often. It's only useful in a few rare cases, for example:

- jQuery Validation currently will only validate elements that have names. To use this with a Knockout UI, it's sometimes necessary to apply the uniqueName binding to avoid confusing jQuery Validation.
- IE 6 does not allow radio buttons to be checked if they don't have a name attribute. KO will internally use uniqueName on those elements to ensure they can be checked.

Syntax

```
uniqueName: <binding-value>
```

Parameters

Parameter here will be Boolean value true or false or an expression resulting in Boolean like value. A unique name is generated by KO for the element for which this parameter is set to *true* or *true*-like value.

Example

Let us take a look at the following example which demonstrates the use of uniqueName binding.

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS UniqueName Binding</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

  <p>Enter your pet's name: <input data-bind="value: someValue, uniqueName:
true" /></p>

  <p><button data-bind="click: showMessage">Click here to read message
</button></p>

<script type="text/javascript">
  function ViewModel (){
    this.someValue = ko.observable();
    this.showMessage = function(){
      alert(" Nice Name"+ "\nSee rendered markup to view unique name generated!!!");
    }
  };

  var vm = new ViewModel();
  ko.applyBindings(vm);
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **uniquename-bind.htm** file.
- Open this HTML file in a browser.
- Press F12 and observe the rendered markup. Unique name is generated by KO.

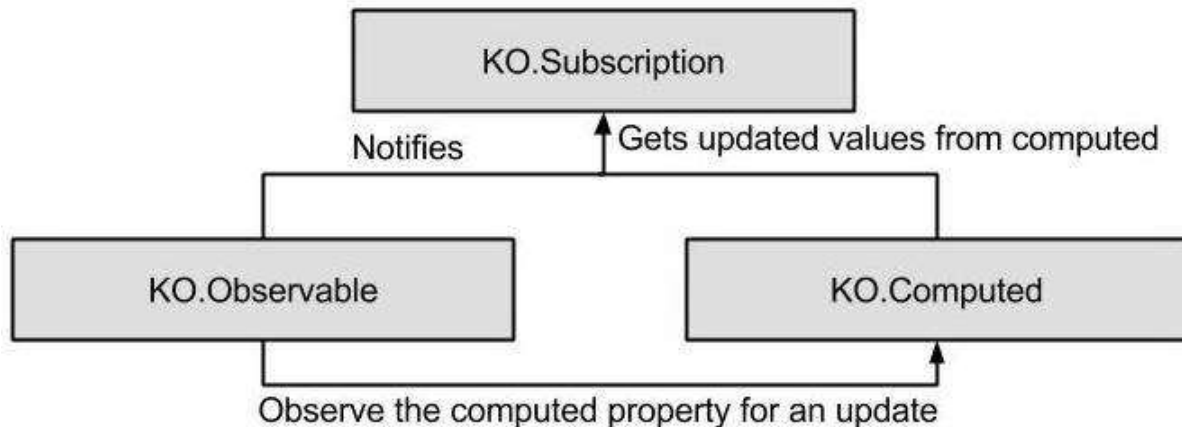
Enter your pet's name:

[Click here to read message](#)

8. KnockoutJS – Dependency Tracking

KnockoutJs automatically tracks the dependencies when the values get updated. It has a single object called **dependency tracker** (ko.dependencyDetection) which acts as an intermediate between the two parties for subscribing the dependencies.

Following is the algorithm for dependency tracking.



The process is as follows.

Step 1: Whenever you declare a computed observable, KO immediately invokes its evaluator function to get its initial value.

Step 2: Subscription is set up to any observable that the evaluator reads. In an application, the old subscriptions which are no longer used are disposed.

Step 3: KO finally notifies the updated computed observable.

Example

```
<!DOCTYPE html>
<html>
  <head>
    <title>KnockoutJS How Dependency Tracking Works</title>
    <!-- CDN's-->
    <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js" type="text/javascript"></script>
  </head>
  <body>
    <div>
      <form data-bind="submit: addFruits">
        <b>Add Fruits:</b>
      </form>
    </div>
  </body>
</html>
```

```

        <input data-bind='value: fruitToAdd, valueUpdate: "afterkeydown"' />
        <button type="submit" data-bind="enable: fruitToAdd().length >
0">Add</button>
        <p><b>Your fruits list:</b></p>
        <select multiple="multiple" width="50" data-bind="options: fruits">
</select>
    </form>
</div>
<script>
    var Addfruit = function(fruits) {
        this.fruits = ko.observableArray(fruits);
        this.fruitToAdd = ko.observable("");
        this.addFruits = function() {
            if (this.fruitToAdd() != "") {
                this.fruits.push(this.fruitToAdd()); // Adds a fruit
                this.fruitToAdd(""); // Clears the text box
            }
        }.bind(this); // "this" is the view model
    };

    ko.applyBindings(new Addfruit(["Apple", "Orange", "Banana"]));
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **dependency_tracking.htm** file.
- Open this HTML file in a browser.
- Enter any fruit name and click the Add button.

Add Fruits:

Your fruits list:

Apple
Orange
Banana

Controlling Dependencies Using Peek

The Computed Observable can be accessed without creating a dependency, by using the **peek** function. It controls the Observable by updating the computed property.

Example

```

<!DOCTYPE html>
<html>
  <head>
    <title>KnockoutJs Controlling Dependencies Using Peek</title>
    <!-- CDN's-->
    <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js"
type="text/javascript"></script>
  </head>
  <body>
    <div class="logblock">
      <h3>Computed Log</h3>
      <pre class="log" data-bind="html: computedLog"></pre>
    </div>

    <script>
      function AppData() {
        this.firstName = ko.observable('John');
        this.lastName = ko.observable('Burns');
        this.computedLog = ko.observable('Log: ');
        this.fullName = ko.computed(function () {

```

```
        var value = this.firstName() + " " + this.lastName();
        this.computedLog(this.computedLog.peek() + value + '; <br/>');
        return value;
    }, this);

    this.step = ko.observable(0);
    this.next = function () {
        this.step(this.step() === 2 ? 0 : this.step()+1);
    };
};
ko.applyBindings(new AppData());
</script>
</body>
</html>
```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **dependency_tracking_peek.htm** file.
- Open this HTML file in a browser.

Computed Log

Log: John Burns;

Observations

Ignoring Dependencies Within a Computed Dependency

The **ko.ignoreDependencies** function helps ignore those dependencies that you don't want to track within the computed dependencies. Following is its syntax.

```
ko.ignoreDependencies( callback, callbackTarget, callbackArgs );
```

Why Circular Dependencies Aren't Meaningful

If KO is evaluating a Computed Observable, then it will not restart an evaluation of the dependent Computed Observable. Hence, it doesn't make sense to include cycles in your dependency chains.

9. KnockoutJS – Templating

Template is a set of DOM elements which can be used repetitively. Templating makes it easy to build complex applications due to its property of minimizing duplication of DOM elements.

There are 2 ways of creating templates.

Native templating: This method supports the control flow bindings such as `foreach`, `with`, and `if`. These bindings capture HTML markup existing in the element and use it as template for random items. No external library is required for this templating.

String-based templating: KO connects to the third party engine to pass ViewModel values into it and injects the resulting markup into the document. For example, `jQuery.tmpl` and `Underscore Engine`.

Syntax

```
template: <parameter-value>

<script type="text/html" id="template-name">
...
...    // DOM elemets to be processed
...
</script>
```

Note that **type** is provided as **text/html** in the script block to notify KO that, it is not an executable block rather just a template block which needs to be rendered.

Parameters

Combination of the following properties can be sent as parameter-value to template.

- **name** - This represents the name of the template.
- **nodes** - This represents an array of DOM nodes to be used as the template. This parameter is ignored if the name parameter is passed.
- **data** - This is nothing but data to be shown via the template.
- **if** - Template will be served if the given condition results in true or true-like value.
- **foreach** - To serve template in foreach format.
- **as** - This is just to create an alias in foreach element.
- **afterAdd, afterRender, beforeRemove** - These are all to represent callable functions to be executed depending on the operation performed.

Observations

Rendering a named Template

Templates are defined implicitly by HTML markup inside DOM when used with control flow bindings. However if you want to, you can factor out templates into a separate element and then reference them by name.

Example

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Templating - Named Template</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

  <h2>Friends List</h2>
  Here are the Friends from your contact page:
  <div data-bind="template: { name: 'friend-template', data: friend1 }"></div>
  <div data-bind="template: { name: 'friend-template', data: friend2 }"></div>

  <script type="text/html" id="friend-template">
    <h3 data-bind="text: name"></h3>
    <p>Contact Number: <span data-bind="text: contactNumber"></span></p>
    <p>Email-id: <span data-bind="text: email"></span></p>
  </script>

  <script type="text/javascript">
    function MyViewModel() {
      this.friend1= { name: 'Smith', contactNumber: 4556750345, email:
'smith123@gmail.com' };
      this.friend2 = { name: 'Jack', contactNumber: 6789358001, email:
'jack123@yahoo.com' };
    }

    var vm = new MyViewModel();
    ko.applyBindings(vm);
```

```

</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **template-named.htm** file.
- Open this HTML file in a browser.
- Here, friend-template is used 2 times.

Friends List

Here are the Friends from your contact page:

Smith

Contact Number: 4556750345

Email-id: smith123@gmail.com

Jack

Contact Number: 6789358001

Email-id: jack123@yahoo.com

Using "foreach" in Template

Following is an example of using **foreach** parameter along with the template name.

Example

```

<!DOCTYPE html>
<head>
  <title>KnockoutJS Templating - foreach used with Template</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>

```

```

<body>

    <h2>Friends List</h2>
    Here are the Friends from your contact page:
    <div data-bind="template: { name: 'friend-template', foreach:
friends }"></div>

    <script type="text/html" id="friend-template">
        <h3 data-bind="text: name"></h3>
        <p>Contact Number: <span data-bind="text: contactNumber"></span></p>
        <p>Email-id: <span data-bind="text: email"></span></p>
    </script>

    <script type="text/javascript">
        function MyViewModel() {
            this.friends = [
                {name: 'Smith', contactNumber: 4556750345, email:
'smith123@gmail.com' },
                { name: 'Jack', contactNumber: 6789358001, email:
'jack123@yahoo.com' },
                { name: 'Lisa', contactNumber: 4567893131, email:
'lisa343@yahoo.com' }
            ]
        }

        var vm = new MyViewModel();
        ko.applyBindings(vm);
    </script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **template-foreach.htm** file.
- Open this HTML file in a browser.
- Here, foreach control is used in template binding.

Friends List

Here are the Friends from your contact page:

Smith

Contact Number: 4556750345

Email-id: smith123@gmail.com

Jack

Contact Number: 6789358001

Email-id: jack123@yahoo.com

Lisa

Contact Number: 4567893131

Email-id: lisa343@yahoo.com

Creating alias Using as Keyword for foreach Items

Following is how an alias can be created for a foreach item:

```
<div data-bind="template: { name: 'friend-template', foreach: friends, as:
'frnz' }"></div>
```

It becomes easy to refer to parent objects from inside of foreach loops by creating alias. This feature is useful when the code is complex and nested at multiple levels.

Example

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Templating - using alias in Template</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>
```

```

<h2>Friends List</h2>
Here are the Friends from your contact page:
<ul data-bind="template: { name: 'friend-template', foreach: friends, as:
'frnz' }"></ul>

<script type="text/html" id="friend-template">
  <li>
    <h3 data-bind="text: name"></h3>
    <span>Contact Numbers</span>
    <ul data-bind="template: { name : 'contacts-template',
foreach:contactNumber, as: 'cont'} "></ul>
    <p>Email-id: <span data-bind="text: email"></span></p>
  </li>
</script>

<script type="text/html" id="contacts-template">
  <li>
    <p><span data-bind="text: cont"></span></p>
  </li>
</script>

<script type="text/javascript">
  function MyViewModel() {
    this.friends = ko.observableArray( [
      {name: 'Smith', contactNumber: [ 4556750345, 4356787934 ] ,
email: 'smith123@gmail.com' },
      { name: 'Jack', contactNumber: [ 6789358001, 3456895445 ],
email: 'jack123@yahoo.com' },
      { name: 'Lisa', contactNumber: [ 4567893131, 9876456783,
1349873445 ], email: 'lisa343@yahoo.com' }
    ]);
  }

  var vm = new MyViewModel();
  ko.applyBindings(vm);
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **template-as-alias.htm** file.
- Open this HTML file in a browser.
- Alias is used instead of full name of arrays.

```

Friends List
Here are the Friends from your contact page:
Smith
Contact Numbers
4556750345
4356787934
Email-id: smith123@gmail.com
Jack
Contact Numbers
6789358001
3456895445
Email-id: jack123@yahoo.com
Lisa
Contact Numbers
4567893131
9876456783
1349873445
Email-id: lisa343@yahoo.com

```

Using afterAdd, beforeRemove, and afterRender

There are situations wherein extra custom logic needs to be run on DOM elements created by the template. In such case, following callbacks can be used. Consider that you are using foreach element then -

afterAdd - This function is invoked when a new item is added to the array mentioned in foreach.

beforeRemove - This function is invoked just before removing the item from an array mentioned in foreach.

afterRender - Function mentioned here is invoked every time foreach is rendered and new entries are added to the array.

Example

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Templating - Use of afterRender Template</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
  <script src="https://code.jquery.com/jquery-2.1.3.min.js"
type="text/javascript"></script>
</head>
<body>

  <h2>Friends List</h2>
  Here are the Friends from your contact page:
  <div data-bind="template: { name: 'friend-template', foreach: friends ,
afterRender: afterProcess}"></div>

  <script type="text/html" id="friend-template">
    <h3 data-bind="text: name"></h3>
    <p>Contact Number: <span data-bind="text: contactNumber"></span></p>
    <p>Email-id: <span data-bind="text: email"></span></p>
    <button data-bind="click: $root.removeContact">remove </button>
  </script>

  <script type="text/javascript">
    function MyViewModel() {
      self= this;
      this.friends = ko.observableArray([
        {name: 'Smith', contactNumber: 4556750345, email:
'smith123@gmail.com' },
        { name: 'Jack', contactNumber: 6789358001, email:
'jack123@yahoo.com' },
      ])

      this.afterProcess = function(elements, data){
```



```

        $(elements).css({color: 'magenta' });
    }
    self.removeContact = function(){
        self.friends.remove(this);
    }
}
var vm = new MyViewModel();
ko.applyBindings(vm);
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **template-afterrender.htm** file.
- Open this HTML file in a browser.
- Here, afterProcess function is executed every time foreach is rendered.

Friends List

Here are the Friends from your contact page:

Smith

Contact Number: 4556750345

Email-id: smith123@gmail.com

remove

Jack

Contact Number: 6789358001

Email-id: jack123@yahoo.com

remove

Choosing Template Dynamically

If there are multiple templates available, then one can be chosen dynamically by making the name as **observable** parameter. Hence, the template value will be re-evaluated as the name parameter changes and in turn data will be re-rendered.

Example

```
<!DOCTYPE html>
<head>
  <title>KnockoutJS Templating - Dynamic Template</title>
  <script src="https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js"
type="text/javascript"></script>
</head>
<body>

  <h2>Friends List</h2>
  Here are the Friends from your contact page:
  <div data-bind="template: { name: whichTemplate, foreach: friends }"></div>

  <script type="text/html" id="only-phon">
    <h3 data-bind="text: name"></h3>
    <p>Contact Number: <span data-bind="text: contactNumber"></span></p>
  </script>

  <script type="text/html" id="only-email">
    <h3 data-bind="text: name"></h3>
    <p>Email-id: <span data-bind="text: email"></span></p>
  </script>

  <script type="text/javascript">
    function MyViewModel() {
      this.friends = ko.observableArray([
        {name: 'Smith', contactNumber: 4556750345, email:
'smith123@gmail.com' ,active: ko.observable(true) },
        {name: 'Jack', contactNumber: 6789358001, email:
'jack123@yahoo.com', active: ko.observable(false) },
      ]);
      this.whichTemplate = function(friends){
```

```

        return friends.active() ? "only-phon" : "only-email";
    }
}
var vm = new MyViewModel();
ko.applyBindings(vm);
</script>
</body>
</html>

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **template-dynamic.htm** file.
- Open this HTML file in a browser.
- Template to be used is decided depending on the active flag value.

Friends List

Here are the Friends from your contact page:

Smith

Contact Number: 4556750345

Jack

Email-id: jack123@yahoo.com

Using External String-based Engines

Native templating works perfectly with various control flow elements even with nested code blocks. KO also offers a way to integrate with external templating library such as Underscore templating Engine or JQuery.tmpl.

As mentioned on the official site JQuery.tmpl is no longer under active development since December 2011. Hence, KO's native templating is only recommended instead of JQuery.tmpl or any other string-based template engine.

Please refer to the [official site](#) for more details on this.

10. KnockoutJS – Components

Components are a huge way of organizing the UI code for structuring a large application and promoting code reusability.

It is inherited or nested from other component. For loading and configuration, it defines its own conventions or logic.

It is packaged to reuse throughout the application or the project. Represents the complete sections of application or small controls/widgets. It can be loaded or preloaded on demand.

Component Registration

Components can register using the **ko.components.register()** API. It helps to load and represent the components in KO. Component name with configuration is expected for registration. The configuration specifies how to determine the viewModel and template.

Syntax

Components can be registered as follows:

```
ko.components.register('component-name', {  
    viewModel: {...},    //function code  
    template: {...}      //function code  
});
```

- The **component-name** can be any nonempty string.
- **viewModel** is optional, and can take any of the viewModel formats listed in the next sections.
- **template** is required, and can take any of the template formats listed in the next sections.

Stating a ViewModel

Following table lists the viewModel formats that can be used to register the components.

Sr. No.	viewModel Forms & Description
1	<p>constructor function</p> <p>It creates a separate viewModel object for each component. The object or function is used to bind in components view.</p> <pre>function SomeComponentViewModel(params) { this.someProperty = params.something; } ko.components.register('component name', { viewModel: SomeComponentViewModel, template: ... });</pre>
2	<p>shared object instance</p> <p>The viewModel object instance is shared. The instance property is passed to use the object directly.</p> <pre>var sharedViewModelInstance = { ... }; ko.components.register('component name', { viewModel: { instance: sharedViewModelInstance }, template: ... });</pre>

3	<p>createViewModel</p> <p>It calls a function which acts as a factory and can be used as view model that can return an object.</p> <pre> ko.components.register('component name', { viewModel: { createViewModel: function(params, componentInfo) { ... //function code } }, template: }); </pre>
4	<p>AMD module</p> <p>It is a module format for defining modules where module and dependencies both are loaded asynchronously.</p> <pre> ko.components.register('component name', { viewModel: { require: 'some/module/name' }, template: ... }); define(['knockout'], function(ko) { function MyViewModel() { // ... } return MyViewModel; }); </pre>

Stating a Template

Following table lists the template formats that can be used to register the components.

Sr. No.	Template Forms
1	<p>element ID</p> <pre>ko.components.register('component name', { template: { element: 'component-template' }, viewModel: ... });</pre>
2	<p>element instance</p> <pre>var elemInstance = document.getElementById('component-template'); ko.components.register('component name', { template: { element: elemInstance }, viewModel: ... });</pre>
3	<p>string of markup</p> <pre>ko.components.register('component name', { template: '<input data-bind="value: yourName" />\n <button data-bind="click: addEmp">Add Emp </button>', viewModel: ... });</pre>

4	<p>DOM nodes</p> <pre> var emp = [document.getElementById('node 1'), document.getElementById('node 2'),]; ko.components.register('component name', { template: emp, viewModel: ... }); </pre>
5	<p>document fragement</p> <pre> ko.components.register('component name', { template: someDocumentFragmentInstance, viewModel: ... }); </pre>
6	<p>AMD module</p> <pre> ko.components.register('component name', { template: { require: 'some/template' }, viewModel: ... }); </pre>

Components Registered as a Single AMD Module

The AMD module can register a component by itself without using viewModel/template pair.

```
ko.components.register('component name',{ require: 'some/module'});
```


Component Binding

There are two ways of component binding.

- **Full syntax:** It passes the parameter and object to the component. It can pass using the following properties:
 - **name:** It adds the component name.
 - **params:** It can pass multiple parameters in the object on the component.

```
<div data-bind='component: {
  name: "tutorials point",
  params: { mode: "detailed-list", items: productsList }
}'>
</div>
```

- **Shorthand syntax:** It passes the string as a component name and it does not include parameter in it.

```
<div data-bind='component: "component name"'></div>
```

- **Template-only components:** Components can only define template without specifying the viewModel.

```
ko.components.register('component name', {
  template:'<input data-bind="value: someName" />',
});
```

- **Using Component without a container element:** Components can be used without using extra container element. This can be done using **containerless flow control** which is similar as the comment tag.

```
<!--ko.component: ""-->
<!--/ko-->
```

Custom Element

Custom element is a way for rendering a component. Here, you can directly write a self-descriptive markup element name instead of defining a placeholder, where the components are binded through it.

```
<products-list params="name: userName, type: userType"></products-list>
```

Passing Parameter

params attribute is used to pass the parameter to component viewModel. It is similar to data-bind attribute. The contents of the params attribute are interpreted like a JavaScript object literal (just like a *data-bind* attribute), so you can pass arbitrary values of any type. It can pass the parameter in following ways:

- **Communication between parent and child components:** The component is not instantiated by itself so the viewModel properties are referred from outside of the component and thus would be received by child component viewModel. For example, you can see in the following syntax that **ModelValue** is the parent viewModel, which is received by child viewModel constructor **ModelProperty**.
- **Passing observable expressions:** It has three values in params parameter:
 - **simpleExpression:** It is a numeric value. It does not involve any observables.
 - **simpleObservable:** It is an instance that is defined on parent viewModel. The parent viewModel will automatically get the changes on observable done by child viewModel.
 - **observableExpression:** Expression reads the observable when the expression is evaluated by itself. When the observable value changes, then the result of expression can also change over time.

We can pass the parameters as follows:

```
<some-component
  params='simpleExpression: 1 + 1,
         simpleObservable: myObservable,
         observableExpression: myObservable() + 1'>
</some-component>
```

We can pass the parameters in viewModel as follows:

```
<some-component
  params='objectValue:{a: 3, b: 2},
         dateValue: new date(),
         stringValue: "Hi",
         numericValue:123,
         boolValue: true/false,
         ModelProperty: ModelValue'>
</some-component>
```

Passing Markup into Components

The received markup is used to create a component and is selected as a part of the output. Following nodes are passed as part of the output in the component template.

```
template: { nodes: $componentTemplateNodes }
```

Controlling Custom Element Tag Names

The names which you register in the components using **ko.components.register**, the same name corresponds to the custom element tag names. We can change the custom element tag names by overriding it to control using **getComponentNameForNode**.

```
ko.components.getComponentNameForNode = function(node) {
  ...
  ... //function code
  ...
}
```

Registering Custom Elements

The custom elements can be made available immediately, if the default component loader is used and hence the component is registered using **ko.components.register**. If we are not using the **ko.components.register** and implementing the custom component loader, then the custom element can be used by defining any element name of choice. There is no need to specify configuration when you are using **ko.components.register** as the custom component loader does not use it anymore.

```
ko.components.register('custom-element', { ..... });
```

Example

```
<!DOCTYPE html>
<head>
<title>KnockoutJS Components</title>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></scrip
t>
<script src="https://cdnjs.cloudflare.com/ajax/libs/knockout/3.2.0/knockout-
min.js"></script>
</head>
<body>
<!--params attribute is used to pass the parameter to component viewModel.-->
<click params="a: a, b: b"></click>
```

```

<!--template is used for a component by specifying its ID -->
<template id="click-1">
    <div data-bind="text: a"></div>

    <!--Use data-bind attribute to bind click:function() to ViewModel. -->
    <button data-bind="click:function(){callback(1)}">Increase</button>
    <button data-bind="click:function(){callback(-1)}">Decrease</button>
</template>
<script>
    // Here components are registered
    ko.components.register('click', {
        viewModel: function(params) {
            self = this;
            this.a = params.a;
            this.b = params.b;

            this.callback = function(num){
                self.b(parseInt(num));
                self.a( self.a() + parseInt(num) );
            };
        },
        template: { element: 'click-1' }
    });

    // keeps an eye on variable for any modification in data
    function viewModel(){
        this.a = ko.observable(2);
        this.b = ko.observable(0);
    }

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

```

Output

Let's carry out the following steps to see how the above code works:

- Save the above code in **component_register.htm** file.
- Open this HTML file in a browser.

2

Component Loaders

Component loaders are used to pass the template/viewModel pair asynchronously for the given component name.

The default Component Loader

The default component loader depends on the explicitly registered configuration. Each component is registered before using the component.

```
ko.components.defaultLoader
```

Component Loader Utility Functions

The default component loader can read and write using the following functions.

Sr. No.	Utility Functions & Description
1	ko.components.register(name, configuration) Component is registered.

2	ko.components.isRegistered(name) If the particular component name is already registered, then it returns as true else false.
3	ko.components.unregister(name) The component name is removed from the registry.
4	ko.components.get(name, callback) This function goes turn by turn to each registered loader to find who has passed the viewModel/template definition for component name as first. Then it returns viewModel/template declaration by invoking callback . If the registered loader could not find anything about the component, then it invokes callback(null) .
5	ko.components.clearCachedDefinition(name) This function can be called when we want to clear the given component cache entry. If the component is needed next time, again the loaders will be consulted.

Implementing a Custom Component Loader

The custom component loader can be implemented in the following ways:

- **getConfig(name, callback)**: Depending on the names, we can pass configurations programatically. We can call `callback(componentConfig)` to pass the configurations, where the object `componentConfig` can be used by the `loadComponent` or any other loader.
- **loadComponent(name, componentConfig, callback)**: This function resolves the `viewModel` and the `template` portion of `config` depending upon the way it is configured. We can call `callback(result)` to pass the `viewModel/template` pair, where the object `result` is defined by the following properties:
 - **template** - Required. Return array of DOM nodes.
 - **createViewModel(params, componentInfo)** - Optional. Returns the `viewModel` Object depending on how the `viewModel` property was configured.
- **loadTemplate(name, templateConfig, callback)**: DOM nodes is passed in a `template` using custom logic. The object `templateConfig` is a property of the `template` from an object `componentConfig`. `callback(domNodeArray)` is called to pass an array of DOM nodes.
- **loadViewModel(name, templateConfig, callback)**: `viewModel` factory is passed in a `viewModel` configuration using custom logic.