

Knockout.js

Part I: Getting started

1. How KO works and what benefits it brings?

Knockout is a JavaScript library that helps you to create rich, responsive display and editor user interfaces with a clean underlying data model. Any time you have sections of UI that update dynamically (e.g., changing depending on the user's actions or when an external data source changes), KO can help you implement it more simply and maintainably.

Headline features:

- Elegant dependency tracking - automatically updates the right parts of your UI whenever your data model changes.
- Declarative bindings - a simple and obvious way to connect parts of your UI to your data model. You can construct complex dynamic UIs easily using arbitrarily nested binding contexts.
- Trivially extensible - implement custom behaviors as new declarative bindings for easy reuse in just a few lines of code.

Additional benefits:

- Pure JavaScript library - works with any server or client-side technology
- Can be added on top of your existing web application without requiring major architectural changes
- Compact - around 13kb after gzipping
- Works on any mainstream browser (IE 6+, Firefox 2+, Chrome, Safari, others)
- Comprehensive suite of specifications (developed BDD-style) means its correct functioning can easily be verified on new browsers and platforms

Developers familiar with Ruby on Rails, ASP.NET MVC, or other MV* technologies may see MVVM as a real-time form of MVC with declarative syntax. In another sense, you can think of KO as a general way to make UIs for editing JSON data... whatever works for you :)

OK, how do you use it?

The quickest and most fun way to get started is by working through the interactive tutorials. Once you've got to grips with the basics, explore the live examples and then have a go with it in your own project.

Is KO intended to compete with jQuery (or Prototype, etc.) or work with it?

Everyone loves jQuery! It's an outstanding replacement for the clunky, inconsistent DOM API we had to put up with in the past. jQuery is an excellent low-level way to manipulate elements and event handlers in a web page. I certainly still use jQuery for low-level DOM manipulation. KO solves a different problem.

As soon as your UI gets nontrivial and has a few overlapping behaviors, things can get tricky and expensive to maintain if you only use jQuery. Consider an example: you're displaying a list of items, stating the number of items in that list, and want to enable an 'Add' button only when there are fewer than 5 items. jQuery doesn't have a concept of an underlying data model, so to get the number of items you have to infer it from the number of TRs in a table or the number of DIVs with a certain CSS class. Maybe the number of items is displayed in some SPAN, and you have to remember to update that SPAN's text when the user adds an item. You also must remember to disable the 'Add' button when the number of TRs is 5. Later, you're asked also to implement a 'Delete' button and you have to figure out which DOM elements to change whenever it's clicked.

How is Knockout different?

It's much easier with KO. It lets you scale up in complexity without fear of introducing inconsistencies. Just represent your items as a JavaScript array, and then use a foreach binding to transform this array into a TABLE or set of DIVs. Whenever the array changes, the UI changes to match (you don't have to figure out how to inject new TRs or where to inject them). The rest of the UI stays in sync. For example, you can declaratively bind a SPAN to display the number of items as follows:

```
There are <span data-bind="text: myItems().count"></span> items
```

That's it! You don't have to write code to update it; it updates on its own when the myItems array changes. Similarly, to make the 'Add' button enable or disable depending on the number of items, just write:

```
<button data-bind="enable: myItems().count < 5">Add</button>
```

Later, when you're asked to implement the 'Delete' functionality, you don't have to figure out what bits of the UI it has to interact with; you just make it alter the underlying data model.

To summarise: KO doesn't compete with jQuery or similar low-level DOM APIs. KO provides a complementary, high-level way to link a data model to a UI. KO itself doesn't depend on jQuery, but you

can certainly use jQuery at the same time, and indeed that's often useful if you want things like animated transitions.

2. Downloading and installing

Knockout's core library is pure JavaScript and doesn't depend on any other libraries. So, to add KO to your project, just follow these steps:

- Download the latest version of the Knockout JavaScript file from the downloads page. For normal development and production use, use the default, minified version (knockout-x.y.z.js).

For debugging only, use the larger, non-minified version (knockout-x.y.z.debug.js). This behaves the same as the minified version, but has human-readable source code with full variable names and comments, and does not hide internal APIs.

- Reference the file using a `<script>` tag somewhere on your HTML pages.

For example,

```
<script type='text/javascript' src='knockout-2.1.0.js'></script>
```

... and now you're ready to use it. (Obviously, update the `src` attribute to match the location where you put the downloaded file.)

If you're new to Knockout, get started with interactive tutorials, see some live examples, or dig into documentation about observables.

Part II: Observables

1. Creating *view models with observables*

Knockout is built around three core features:

- Observables and dependency tracking
- Declarative bindings
- Templating

On this page, you'll learn about the first of these three. But before that, let me explain the MVVM pattern and the concept of a view model.

MVVM and View Models

Model-View-View Model (MVVM) is a design pattern for building user interfaces. It describes how you can keep a potentially sophisticated UI simple by splitting it into three parts:

- *A model*: your application's stored data. This data represents objects and operations in your business domain (e.g., bank accounts that can perform money transfers) and is independent of any UI. When using KO, you will usually make Ajax calls to some server-side code to read and write this stored model data.
- *A view model*: a pure-code representation of the data and operations on a UI. For example, if you're implementing a list editor, your view model would be an object holding a list of items, and exposing methods to add and remove items.

Note that this is not the UI itself: it doesn't have any concept of buttons or display styles. It's not the persisted data model either - it holds the unsaved data the user is working with. When using KO, your view models are pure JavaScript objects that hold no knowledge of HTML. Keeping the view model abstract in this way lets it stay simple, so you can manage more sophisticated behaviors without getting lost.

- *A view*: a visible, interactive UI representing the state of the view model. It displays information from the view model, sends commands to the view model (e.g., when the user clicks buttons), and updates whenever the state of the view model changes.

When using KO, your view is simply your HTML document with declarative bindings to link it to the view model. Alternatively, you can use templates that generate HTML using data from your view model.

To create a view model with KO, just declare any JavaScript object. For example,

```
var myViewModel = {  
  personName: 'Bob',  
  personAge: 123  
};
```

You can then create a very simple *view* of this view model using a declarative binding. For example, the following markup displays the `personName` value:

```
The name is <span data-bind="text: personName"></span>
```

Activating Knockout

The `data-bind` attribute isn't native to HTML, though it is perfectly OK (it's strictly compliant in HTML 5, and causes no problems with HTML 4 even though a validator will point out that it's an unrecognized attribute). But since the browser doesn't know what it means, you need to activate Knockout to make it take effect.

To activate Knockout, add the following line to a `<script>` block:

```
ko.applyBindings(myViewModel);
```

You can either put the script block at the bottom of your HTML document, or you can put it at the top and wrap the contents in a DOM-ready handler such as jQuery's `$` function.

That does it! Now, your view will display as if you'd written the following HTML:

```
The name is <span>Bob</span>
```

In case you're wondering what the parameters to `ko.applyBindings` do,

- The first parameter says what view model object you want to use with the declarative bindings it activates
- Optionally, you can pass a second parameter to define which part of the document you want to search for `data-bind` attributes. For example, `ko.applyBindings(myViewModel, document.getElementById('someElementId'))`. This restricts the activation to the element with ID `someElementId` and its descendants, which is useful if you want to have multiple view models and associate each with a different region of the page.

Pretty simple, really.

Observables

OK, you've seen how to create a basic view model and how to display one of its properties using a binding. But one of the key benefits of KO is that it updates your UI automatically when the view model changes. How can KO know when parts of your view model change? Answer: you need to declare your model properties as *observables*, because these are special JavaScript objects that can notify subscribers about changes, and can automatically detect dependencies.

For example, rewrite the preceding view model object as follows:

```
var myViewModel = {  
    personName: ko.observable('Bob'),  
    personAge: ko.observable(123)  
};
```

You don't have to change the view at all - the same data-bind syntax will keep working. The difference is that it's now capable of detecting changes, and when it does, it will update the view automatically.

Reading and writing observables

Not all browsers support JavaScript getters and setters (* cough * IE * cough *), so for compatibility, `ko.observable` objects are actually *functions*.

- To **read** the observable's current value, just call the observable with no parameters. In this example, `myViewModel.personName()` will return 'Bob', and `myViewModel.personAge()` will return 123.
- To **write** a new value to the observable, call the observable and pass the new value as a parameter. For example, calling `myViewModel.personName('Mary')` will change the name value to 'Mary'.
- To write values to **multiple observable properties** on a model object, you can use *chaining syntax*. For example, `myViewModel.personName('Mary').personAge(50)` will change the name value to 'Mary' and the age value to 50.

The whole point of observables is that they can be observed, i.e., other code can say that it wants to be notified of changes. That's what many of KO's built-in bindings do internally. So, when you wrote `data-bind="text: personName"`, the text binding registered itself to be notified when `personName` changes (assuming it's an observable value, which it is now).

When you change the name value to 'Mary' by calling `myViewModel.personName('Mary')`, the text binding will automatically update the text contents of the associated DOM element. That's how changes to the view model automatically propagate to the view.

Explicitly subscribing to observables

You won't normally need to set up subscriptions manually, so beginners should skip this section.

For advanced users, if you want to register your own subscriptions to be notified of changes to observables, you can call their subscribe function. For example,

```
myViewModel.personName.subscribe(function(newValue) {  
    alert("The person's new name is " + newValue);  
});
```

The subscribe function is how many parts of KO work internally. You can also terminate a subscription if you wish: first capture it as a variable, then you can call its dispose function, e.g.:

```
var subscription = myViewModel.personName.subscribe(function(newValue) { /* do stuff */ });  
  
// ...then later...  
  
subscription.dispose(); // I no longer want notifications
```

Most of the time you don't need to do this, because the built-in bindings and templating system take care of managing subscriptions.

2. Using *computed observables*

What if you've got an observable for `firstName`, and another for `lastName`, and you want to display the full name? That's where *computed observables* come in - these are functions that are dependent on one or more other observables, and will automatically update whenever any of these dependencies change.

For example, given the following view model class,

```
function AppViewModel() {  
    this.firstName = ko.observable('Bob');  
    this.lastName = ko.observable('Smith');  
}
```

... you could add a computed observable to return the full name:



Now you could bind UI elements to it, e.g.:

```
function AppViewModel() {  
    // ... leave firstName and lastName unchanged ...
```

```
this.fullName = ko.computed(function() {  
    return this.firstName() + " " + this.lastName();  
}, this);  
}
```

... and they will be updated whenever `firstName` or `lastName` changes (your evaluator function will be called once each time any of its dependencies change, and whatever value you return will be passed on to the observers such as UI elements or other computed observables).

Managing 'this'

Beginners may wish to skip this section - as long as you follow the same coding patterns as the examples, you won't need to know or care about it!

In case you're wondering what the second parameter to `ko.computed` is (the bit where I passed `this` in the preceding code), that defines the value of `this` when evaluating the computed observable. Without passing it in, it would not have been possible to refer to `this.firstName()` or `this.lastName()`. Experienced JavaScript coders will regard this as obvious, but if you're still getting to know JavaScript it might seem strange. (Languages like C# and Java never expect the programmer to set a value for `this`, but JavaScript does, because its functions themselves aren't part of any object by default.)

A popular convention that simplifies things

There's a popular convention for avoiding the need to track `this` altogether: if your viewmodel's constructor copies a reference to `this` into a different variable (traditionally called `self`), you can then use `self` throughout your viewmodel and don't have to worry about it being redefined to refer to something else. For example:

```
function AppViewModel() {  
    var self = this;  
  
    self.firstName = ko.observable('Bob');  
    self.lastName = ko.observable('Smith');  
    self.fullName = ko.computed(function() {  
        return self.firstName() + " " + self.lastName();  
    });  
}
```



```
});  
}
```

Because `self` is captured in the function's closure, it remains available and consistent in any nested functions, such as the `ko.computed` evaluator. This convention is even more useful when it comes to event handlers, as you'll see in many of the live examples.

Dependency chains just work

Of course, you can create whole chains of computed observables if you wish. For example, you might have:

- an **observable** called `items` representing a set of items
- another **observable** called `selectedIndexes` storing which item indexes have been 'selected' by the user
- a **computed observable** called `selectedItems` that returns an array of item objects corresponding to the selected indexes
- another **computed observable** that returns true or false depending on whether any of `selectedItems` has some property (like being new or being unsaved). Some UI element, like a button, might be enabled or disabled based on this value.

Then, changes to `items` or `selectedIndexes` will ripple through the chain of computed observables, which in turn updates any UI bound to them. Very tidy and elegant.

Writeable computed observables

Beginners may wish to skip this section - writeable computed observables are fairly advanced and are not necessary in most situations

As you've learned, computed observables have a value that is computed from other observables. In that sense, computed observables are normally *read-only*. What may seem surprising, then, is that it is possible to make computed observables *writeable*. You just need to supply your own callback function that does something sensible with written values.

You can then use your writeable computed observable exactly like a regular observable - performing two-way data binding with DOM elements; with your own custom logic intercepting all reads and writes. This is a powerful feature with a wide range of possible uses.

Example 1: Decomposing user input

Going back to the classic “first name + last name = full name” example, you can turn things back-to-front: make the `fullName` computed observable writeable, so that the user can directly edit the full name, and their supplied value will be parsed and mapped back to the underlying `firstName` and `lastName` observables:

```
function MyViewModel() {
  this.firstName = ko.observable('Planet');
  this.lastName = ko.observable('Earth');

  this.fullName = ko.computed({
    read: function () {
      return this.firstName() + " " + this.lastName();
    },
    write: function (value) {
      var lastSpacePos = value.lastIndexOf(" ");
      if (lastSpacePos > 0) { // Ignore values with no space character
        this.firstName(value.substring(0, lastSpacePos)); // Update "firstName"
        this.lastName(value.substring(lastSpacePos + 1)); // Update "lastName"
      }
    },
    owner: this
  });
}
ko.applyBindings(new MyViewModel());
```

In this example, the write callback handles incoming values by splitting the incoming text into “firstName” and “lastName” components, and writing those values back to the underlying observables. You can bind this view model to your DOM in the obvious way, as follows:

```
<p>First name: <span data-bind="text: firstName"></span></p>
<p>Last name: <span data-bind="text: lastName"></span></p>
<h2>Hello, <input data-bind="value: fullName"/>!</h2>
```

This is the exact opposite of the Hello World example, in that here the first and last names are not editable, but the combined full name is editable.

The preceding view model code demonstrates the *single parameter syntax* for initialising computed observables. You can pass a JavaScript object with any of the following properties:

- `read` — Required. A function that is used to evaluate the computed observable’s current value.

- `write` — Optional. If given, makes the computed observable writable. This is a function that receives values that other code is trying to write to your computed observable. It's up to you to supply custom logic to handle the incoming values, typically by writing the values to some underlying observable(s).
- `owner` — Optional. If given, defines the value of `this` whenever KO invokes your read or write callbacks. See the section "Managing this" earlier on this page for more information.

Example 2: A value converter

Sometimes you might want to represent a data point on the screen in a different format from its underlying storage. For example, you might want to store a price as a raw float value, but let the user edit it with a currency symbol and fixed number of decimal places. You can use a writable computed observable to represent the formatted price, mapping incoming values back to the underlying float value:

```
function MyViewModel() {
  this.price = ko.observable(25.99);
  this.formattedPrice = ko.computed({
    read: function () {
      return '$' + this.price().toFixed(2);
    },
    write: function (value) {
      // Strip out unwanted characters, parse as float, then write the raw data back to the underlying
      "price" observable
      value = parseFloat(value.replace(/[^\.d]/g, ""));
      this.price(isNaN(value) ? 0 : value); // Write to underlying storage
    },
    owner: this
  });
}
ko.applyBindings(new MyViewModel());
```

It's trivial to bind the formatted price to a text box:

```
<p>Enter bid price: <input data-bind="value: formattedPrice"/></p>
```

Now, whenever the user enters a new price, the text box immediately updates to show it formatted with the currency symbol and two decimal places, no matter what format they entered the value in. This gives a great user experience, because the user sees how the software has understood their data entry as a price. They know they can't enter more than two decimal places, because if they try to, the additional

decimal places are immediately removed. Similarly, they can't enter negative values, because the write callback strips off any minus sign.

Example 3: Filtering and validating user input

Example 1 showed how a writable computed observable can effectively *filter* its incoming data by choosing not to write certain values back to the underlying observables if they don't meet some criteria. It ignored full name values that didn't include a space.

Taking this a step further, you could also toggle an `isValid` flag depending on whether the latest input was satisfactory, and display a message in the UI accordingly. I'll explain in a moment an easier way of doing validation, but first consider the following view model, which demonstrates the mechanism:

```
function MyViewModel() {
  this.acceptedNumericValue = ko.observable(123);
  this.lastInputWasValid = ko.observable(true);
  this.attemptedValue = ko.computed({
    read: this.acceptedNumericValue,
    write: function (value) {
      if (isNaN(value))
        this.lastInputWasValid(false);
      else {
        this.lastInputWasValid(true);
        this.acceptedNumericValue(value); // Write to underlying storage
      }
    },
    owner: this
  });
}
ko.applyBindings(new MyViewModel());
```

... with the following DOM elements:

```
<p>Enter a numeric value: <input data-bind="value: attemptedValue"/></p>
<div data-bind="visible: !lastInputWasValid()">That's not a number!</div>
```

Now, `acceptedNumericValue` will only ever contain numeric values, and any other values entered will trigger the appearance of a validation message instead of updating `acceptedNumericValue`.

Note: For such trivial requirements as validating that an input is numeric, this technique is overkill. It would be far easier just to use jQuery Validation and its number class on the `<input>` element. Knockout and jQuery Validation work together nicely, as demonstrated on the grid editor example. However, the

preceding example demonstrates a more general mechanism for filtering and validating with custom logic to control what kind of user feedback appears, which may be of use if your scenario is more complex than jQuery Validation handles natively.

How dependency tracking works

Beginners don't need to know about this, but more advanced developers will want to know why I keep making all these claims about KO automatically tracking dependencies and updating the right parts of the UI...

It's actually very simple and rather lovely. The tracking algorithm goes like this:

- Whenever you declare a computed observable, KO immediately invokes its evaluator function to get its initial value.
- While your evaluator function is running, KO keeps a log of any observables (or computed observables) that your evaluator reads the value of.
- When your evaluator is finished, KO sets up subscriptions to each of the observables (or computed observables) that you've touched. The subscription callback is set to cause your evaluator to run again, looping the whole process back to step 1 (disposing of any old subscriptions that no longer apply).
- KO notifies any subscribers about the new value of your computed observable.

So, KO doesn't just detect your dependencies the first time your evaluator runs - it redetects them every time. This means, for example, that your dependencies can vary dynamically: dependency A could determine whether you also depend on B or C. Then, you'll only be re-evaluated when either A or your current choice of B or C changes. You don't have to declare dependencies: they're inferred at runtime from the code's execution.

The other neat trick is that declarative bindings are simply implemented as computed observables. So, if a binding reads the value of an observable, that binding becomes dependent on that observable, which causes that binding to be re-evaluated if the observable changes.

Note: Why circular dependencies aren't meaningful

Computed observables are supposed to map a set of observable inputs into a single observable output. As such, it doesn't make sense to include cycles in your dependency chains. Cycles would *not* be analogous to recursion; they would be analogous to having two spreadsheet cells that are computed as functions of each other. It would lead to an infinite evaluation loop.

So what does Knockout do if you have got a cycle in your dependency graph? It avoids infinite loops by enforcing the following rule: a computed observable cannot trigger its own re-evaluation. This is very unlikely to affect your code. It's relevant only if you have a ko.computed (call it A) whose evaluator writes to another observable (call it B) that (directly or via a dependency chain) affects the value of A. In that case, KO will not restart evaluation of A while it is already evaluating, so the resulting value of A will respect only the original value of B, ignoring any update made to B while A's evaluator is running.

Determining if a property is a computed observable

In some scenarios, it is useful to programmatically determine if you are dealing with a computed observable. Knockout provides a utility function, ko.isComputed to help with this situation. For example, you might want to exclude computed observables from data that you are sending back to the server.

```
for (var prop in myObject) {  
  if (myObject.hasOwnProperty(prop) && !ko.isComputed(myObject[prop])) {  
    result[prop] = myObject[prop];  
  }  
}
```

Additionally, Knockout provides similar functions that can operate on observables and computed observables:

- ko.isObservable - returns true for observables, observableArrays, and all computed observables.
- ko.isWriteableObservable - returns true for observable, observableArrays, and writeable computed observables.

What happened to dependent observables?

Prior to Knockout 2.0, computed observables were known as *dependent observables*. With version 2.0, we decided to rename ko.dependentObservable to ko.computed because it is easier to explain, say, and type. But don't worry: this won't break any existing code. At runtime, ko.dependentObservable refers to the same function instance as ko.computed — the two are equivalent.

3. Working with *observable arrays*

If you want to detect and respond to changes on one object, you'd use observables. If you want to detect and respond to changes of a *collection of things*, use an observableArray. This is useful in many scenarios where you're displaying or editing multiple values and need repeated sections of UI to appear and disappear as items are added and removed.

Example

```
var myObservableArray = ko.observableArray(); // Initially an empty array
myObservableArray.push('Some value'); // Adds the value and notifies observers
```

To see how you can bind the observableArray to a UI and let the user modify it, see the simple list example.

Key point: An observableArray tracks which objects are *in* the array, *not* the state of those objects

Simply putting an object into an observableArray doesn't make all of that object's properties themselves observable. Of course, you can make those properties observable if you wish, but that's an independent choice. An observableArray just tracks which objects it holds, and notifies listeners when objects are added or removed.

Prepopulating an observableArray

If you want your observable array **not** to start empty, but to contain some initial items, pass those items as an array to the constructor. For example,

```
// This observable array initially contains three objects
var anotherObservableArray = ko.observableArray([
  { name: "Bungle", type: "Bear" },
  { name: "George", type: "Hippo" },
  { name: "Zippy", type: "Unknown" }
]);
```

Reading information from an observableArray

Behind the scenes, an observableArray is actually an observable whose value is an array (plus, observableArray adds some additional features I'll describe in a moment). So, you can get the underlying JavaScript array by invoking the observableArray as a function with no parameters, just like any other observable. Then you can read information from that underlying array. For example,

```
alert('The length of the array is ' + myObservableArray().length);
alert('The first element is ' + myObservableArray()[0]);
```

Technically you can use any of the native JavaScript array functions to operate on that underlying array, but normally there's a better alternative. KO's observableArray has equivalent functions of its own, and they're more useful because:

- They work on all targeted browsers. (For example, the native JavaScript `indexOf` function doesn't work on IE 8 or earlier, but KO's `indexOf` works everywhere.)
- For functions that modify the contents of the array, such as `push` and `splice`, KO's methods automatically trigger the dependency tracking mechanism so that all registered listeners are notified of the change, and your UI is automatically updated.
- The syntax is more convenient. To call KO's `push` method, just write `myObservableArray.push(...)`. This is slightly nicer than calling the underlying array's `push` method by writing `myObservableArray().push(...)`.

The rest of this page describes `observableArray`'s functions for reading and writing array information.

indexOf

The `indexOf` function returns the index of the first array item that equals your parameter. For example, `myObservableArray.indexOf('Blah')` will return the zero-based index of the first array entry that equals `Blah`, or the value `-1` if no matching value was found.

slice

The `slice` function is the `observableArray` equivalent of the native JavaScript `slice` function (i.e., it returns the entries of your array from a given start index up to a given end index).

Calling `myObservableArray.slice(...)` is equivalent to calling the same method on the underlying array (i.e., `myObservableArray().slice(...)`).

Manipulating an observableArray

`observableArray` exposes a familiar set of functions for modifying the contents of the array and notifying listeners.

pop, push, shift, unshift, reverse, sort, splice

All of these functions are equivalent to running the native JavaScript array functions on the underlying array, and then notifying listeners about the change:

- `myObservableArray.push('Some new value')` adds a new item to the end of array
- `myObservableArray.pop()` removes the last value from the array and returns it
- `myObservableArray.unshift('Some new value')` inserts a new item at the beginning of the array
- `myObservableArray.shift()` removes the first value from the array and returns it
- `myObservableArray.reverse()` reverses the order of the array

- `myObservableArray.sort()` sorts the array contents.
 - By default, it sorts alphabetically (for strings) or numerically (for numbers).
 - Optionally, you can pass a function to control how the array should be sorted. Your function should accept any two objects from the array and return a negative value if the first argument is smaller, a positive value if the second is smaller, or zero to treat them as equal. For example, to sort an array of 'person' objects by last name, you could write `myObservableArray.sort(function(left, right) { return left.lastName == right.lastName ? 0 : (left.lastName < right.lastName ? -1 : 1) })`
- `myObservableArray.splice()` removes and returns a given number of elements starting from a given index. For example, `myObservableArray.splice(1, 3)` removes three elements starting from index position 1 (i.e., the 2nd, 3rd, and 4th elements) and returns them as an array.

For more details about these observableArray functions, see the equivalent documentation of the standard JavaScript array functions.

remove and removeAll

`observableArray` adds some more useful methods that aren't found on JavaScript arrays by default:

- `myObservableArray.remove(someItem)` removes all values that equal `someItem` and returns them as an array
- `myObservableArray.remove(function(item) { return item.age < 18 })` removes all values whose age property is less than 18, and returns them as an array
- `myObservableArray.removeAll(['Chad', 132, undefined])` removes all values that equal 'Chad', 123, or undefined and returns them as an array
- `myObservableArray.removeAll()` removes all values and returns them as an array

destroy and destroyAll (Note: Usually relevant to Ruby on Rails developers only)

The `destroy` and `destroyAll` functions are mainly intended as a convenience for developers using Ruby on Rails:

- `myObservableArray.destroy(someItem)` finds any objects in the array that equal `someItem` and gives them a special property called `_destroy` with value `true`

- `myObservableArray.destroy(function(someItem) { return someItem.age < 18 })` finds any objects in the array whose age property is less than 18, and gives those objects a special property called `_destroy` with value `true`
- `myObservableArray.destroyAll(['Chad', 132, undefined])` finds any objects in the array that equal 'Chad', 123, or undefined and gives them a special property called `_destroy` with value `true`
- `myObservableArray.destroyAll()` gives a special property called `_destroy` with value `true` to all objects in the array

So, what's this `_destroy` thing all about? As I mentioned, it's only really interesting to Rails developers. The convention in Rails is that, when you pass into an action a JSON object graph, the framework can automatically convert it to an ActiveRecord object graph and then save it to your database. It knows which of the objects are already in your database, and issues the correct INSERT or UPDATE statements. To tell the framework to DELETE a record, you just mark it with `_destroy` set to `true`.

Note that when KO renders a `foreach` binding, it automatically hides any objects marked with `_destroy` equal to `true`. So, you can have some kind of "delete" button that invokes the `destroy(someItem)` method on the array, and this will immediately cause the specified item to vanish from the visible UI. Later, when you submit the JSON object graph to Rails, that item will also be deleted from the database (while the other array items will be inserted or updated as usual).

Part III: Using built-in bindings

1. Controlling text and appearance

The visible binding

The visible binding causes the associated DOM element to become hidden or visible according to the value you pass to the binding.

Example

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

<script type="text/javascript">
  var viewModel = {
    shouldShowMessage: ko.observable(true) // Message initially visible
  };
  viewModel.shouldShowMessage(false); // ... now it's hidden
  viewModel.shouldShowMessage(true); // ... now it's visible again
</script>
```

Parameters

- Main parameter
 - When the parameter resolves to a **false-like value** (e.g., the boolean value false, or the numeric value 0, or null, or undefined), the binding sets `yourElement.style.display` to none, causing it to be hidden. This takes priority over any display style you've defined using CSS.
 - When the parameter resolves to a **true-like value** (e.g., the boolean value true, or a non-null object or array), the binding removes the `yourElement.style.display` value, causing it to become visible.
 - Note that any display style you've configured using CSS will then apply (so CSS rules like `display:table-row` work fine in conjunction with this binding).
 - If this parameter is an observable value, the binding will update the element's visibility whenever the value changes. If the parameter isn't observable, it will only set the element's visibility once and will not update it again later.

- Additional parameters
 - None

Note: Using functions and expressions to control element visibility

You can also use a JavaScript function or arbitrary JavaScript expression as the parameter value. If you do, KO will run your function/evaluate your expression, and use the result to determine whether to hide the element.

For example,

```
<div data-bind="visible: myValues().length > 0">
  You will see this message only when 'myValues' has at least one member.
</div>

<script type="text/javascript">
  var viewModel = {
    myValues: ko.observableArray([]) // Initially empty, so message hidden
  };
  viewModel.myValues.push("some value"); // Now visible
</script>
```

Dependencies

None, other than the core Knockout library.

The text binding

The text binding causes the associated DOM element to display the text value of your parameter.

Typically this is useful with elements like `` or `` that traditionally display text, but technically you can use it with any element.

Example

```
Today's message is: <span data-bind="text: myMessage"></span>
<script type="text/javascript">
  var viewModel = {
    myMessage: ko.observable() // Initially blank
  };
  viewModel.myMessage("Hello, world!"); // Text appears
</script>
```

Parameters

- Main parameter
 - KO sets the element's `innerText` (for IE) or `textContent` (for Firefox and similar) property to your parameter value. Any previous text content will be overwritten.
 - If this parameter is an observable value, the binding will update the element's text whenever the value changes. If the parameter isn't observable, it will only set the element's text once and will not update it again later.
 - If you supply something other than a number or a string (e.g., you pass an object or an array), the displayed text will be equivalent to `yourParameter.toString()`
- Additional parameters
 - None

Note 1: Using functions and expressions to determine text values

If you want to determine text programmatically, one option is to create a computed observable, and use its evaluator function as a place for your code that works out what text to display.

For example,

The item is `` today.

```
<script type="text/javascript">
  var viewModel = {
    price: ko.observable(24.95)
  };
  viewModel.priceRating = ko.computed(function() {
    return this.price() > 50 ? "expensive" : "affordable";
  }, viewModel);
</script>
```

Now, the text will switch between "expensive" and "affordable" as needed whenever price changes.

Alternatively, you don't need to create a computed observable if you're doing something simple like this.

You can pass an arbitrary JavaScript expression to the text binding. For example,

```
The item is <span data-bind="text: price() > 50 ? 'expensive' : 'affordable'"></span> today.
```

This has exactly the same result, without requiring the `priceRating` computed observable.

Note 2: About HTML encoding

Since this binding sets your text value using `innerText` or `textContent` (and not using `innerHTML`), it's safe to set any string value without risking HTML or script injection. For example, if you wrote:

```
viewModel.myMessage("<i>Hello, world!</i>");
```

... this would *not* render as italic text, but would render as literal text with visible angle brackets.

If you need to set HTML content in this manner, see the `html` binding.

Note 3: About an IE 6 whitespace quirk

IE 6 has a strange quirk whereby it sometimes ignores whitespace that immediately follows an empty `span`. This is nothing directly to do with Knockout, but in case you do want write:

```
Welcome, <span data-bind="text: userName"></span> to our web site.
```

... and IE 6 renders no whitespace before the words to our web site, you can avoid the problem by putting any text into the ``, e.g.:

```
Welcome, <span data-bind="text: userName">&nbsp;</span> to our web site.
```

Other browsers, and newer versions of IE, don't have this quirk.

Dependencies

None, other than the core Knockout library.

The "html" binding

The `html` binding causes the associated DOM element to display the HTML specified by your parameter.

Typically this is useful when values in your view model are actually strings of HTML markup that you want to render.

Example

```
<div data-bind="html: details"></div>
<script type="text/javascript">
  var viewModel = {
    details: ko.observable() // Initially blank
  };
  viewModel.details("<em>For further details, view the report <a href='report.html'>here</a>.</em>");
// HTML content appears
</script>
```

Parameters

- Main parameter

- KO sets the element's **innerHTML** property to your parameter value. Any previous content will be overwritten.
- If this parameter is an observable value, the binding will update the element's content whenever the value changes. If the parameter isn't observable, it will only set the element's content once and will not update it again later.
- If you supply something other than a number or a string (e.g., you pass an object or an array), the innerHTML will be equivalent to `yourParameter.toString()`
- Additional parameters
 - None

Note: About HTML encoding

Since this binding sets your element's content using `innerHTML`, you should be careful not to use it with untrusted model values, because that might open the possibility of a script injection attack. If you cannot guarantee that the content is safe to display (for example, if it is based on a different user's input that was stored in your database), then you can use the text binding, which will set the element's text value using `innerText` or `textContent` instead.

Dependencies

None, other than the core Knockout library.

The "css" binding

The `css` binding adds or removes one or more named CSS classes to the associated DOM element. This is useful, for example, to highlight some value in red if it becomes negative.

(Note: If you don't want to apply a CSS class but instead want to assign a style attribute value directly, see the `style` binding.)

Example

```
<div data-bind="css: { profitWarning: currentProfit() < 0 }">
  Profit Information
</div>
<script type="text/javascript">
  var viewModel = {
    currentProfit: ko.observable(150000) // Positive value, so initially we don't apply the "profitWarning"
  }
  class
```

```
};  
viewModel.currentProfit(-50); // Causes the "profitWarning" class to be applied  
</script>
```

This will apply the CSS class `profitWarning` whenever the `currentProfit` value dips below zero, and remove that class whenever it goes above zero.

Parameters

- Main parameter
 - You should pass a JavaScript object in which the property names are your CSS classes, and their values evaluate to true or false according to whether the class should currently be applied.
 - You can set multiple CSS classes at once. For example, if your view model has a property called `isSevere`,

```
<div data-bind="css: { profitWarning: currentProfit() < 0, majorHighlight: isSevere }">
```
 - Non-boolean values are interpreted loosely as boolean. For example, 0 and null are treated as false, whereas 21 and non-null objects are treated as true.
 - If your parameter references an observable value, the binding will add or remove the CSS class whenever the observable value changes. If the parameter doesn't reference an observable value, it will only add or remove the class once and will not do so again later.
 - As usual, you can use arbitrary JavaScript expressions or functions as parameter values. KO will evaluate them and use the resulting values to determine whether to apply the CSS class or remove it.
- Additional parameters
 - None

Note: Applying CSS classes whose names aren't legal JavaScript variable names

If you want to apply the CSS class `my-class`, you *can't* write this:

```
<div data-bind="css: { my-class: someValue }">...</div>
```

... because `my-class` isn't a legal identifier name at that point. The solution is simple: just wrap the identifier name in quotes so that it becomes a string literal. This is legal in a JavaScript object literal (technically, according to the JSON spec, you should always do this anyway, though in practice you don't have to). For example,


```
<div data-bind="css: { 'my-class': someValue }">...</div>
```

Dependencies

None, other than the core Knockout library

The "style" binding

The style binding adds or removes one or more style values to the associated DOM element. This is useful, for example, to highlight some value in red if it becomes negative, or to set the width of a bar to match a numerical value that changes.

(Note: If you don't want to apply an explicit style value but instead want to assign a CSS class, see the `css` binding.)

Example

```
<div data-bind="style: { color: currentProfit() < 0 ? 'red' : 'black' }">
  Profit Information
</div>

<script type="text/javascript">
  var viewModel = {
    currentProfit: ko.observable(150000) // Positive value, so initially black
  };
  viewModel.currentProfit(-50); // Causes the DIV's contents to go red
</script>
```

This will set the element's `style.color` property to red whenever the `currentProfit` value dips below zero, and to black whenever it goes above zero.

Parameters

- Main parameter
 - You should pass a JavaScript object in which the property names correspond to style names, and the values correspond to the style values you wish to apply.
 - You can set multiple style values at once. For example, if your view model has a property called `isSevere`,

```
<div data-bind="style: { color: currentProfit() < 0 ? 'red' : 'black', fontWeight:
isSevere() ? 'bold' : '' }">...</div>
```

- If your parameter references an observable value, the binding will update the styles whenever the observable value changes. If the parameter doesn't reference an observable value, it will only set the styles once and will not update them later.
- As usual, you can use arbitrary JavaScript expressions or functions as parameter values. KO will evaluate them and use the resulting values to determine the style values to apply.
- Additional parameters
 - None

Note: Applying styles whose names aren't legal JavaScript variable names

If you want to apply a font-weight or text-decoration style, or any other style whose name isn't a legal JavaScript identifier (e.g., because it contains a hyphen), you must use the *JavaScript name* for that style. For example,

- Don't write `{ font-weight: someValue }`; do write `{ fontWeight: someValue }`
- Don't write `{ text-decoration: someValue }`; do write `{ textDecoration: someValue }`

See also: a longer list of style names and their JavaScript equivalents

Dependencies

None, other than the core Knockout library.

The "attr" binding

The attr binding provides a generic way to set the value of any attribute for the associated DOM element. This is useful, for example, when you need to set the title attribute of an element, the src of an img tag, or the href of a link based on values in your view model, with the attribute value being updated automatically whenever the corresponding model property changes.

Example

```
<a data-bind="attr: { href: url, title: details }">
  Report
</a>

<script type="text/javascript">
  var viewModel = {
    url: ko.observable("year-end.html"),
    details: ko.observable("Report including final year-end statistics")
```

```
};  
</script>
```

This will set the element's href attribute to year-end.html and the element's title attribute to Report including final year-end statistics.

Parameters

- Main parameter
 - You should pass a JavaScript object in which the property names correspond to attribute names, and the values correspond to the attribute values you wish to apply.
 - If your parameter references an observable value, the binding will update the attribute whenever the observable value changes. If the parameter doesn't reference an observable value, it will only set the attribute once and will not update it later.
- Additional parameters
 - None

Note: Applying attributes whose names aren't legal JavaScript variable names

If you want to apply the attribute data-something, you *can't* write this:

```
<div data-bind="attr: { data-something: someValue }">...</div>
```

... because data-something isn't a legal identifier name at that point. The solution is simple: just wrap the identifier name in quotes so that it becomes a string literal. This is legal in a JavaScript object literal (technically, according to the JSON spec, you should always do this anyway, though in practice you don't have to). For example,

```
<div data-bind="attr: { 'data-something': someValue }">...</div>
```

Dependencies

None, other than the core Knockout library.

2. Control flow

The "foreach" binding

The foreach binding duplicates a section of markup for each entry in an array, and binds each copy of that markup to the corresponding array item. This is especially useful for rendering lists or tables.

Assuming your array is an observable array, whenever you later add or remove array entries, the binding will efficiently update the UI to match - inserting or removing more copies of the markup, without affecting any other DOM elements.

Of course, you can arbitrarily nest any number of foreach bindings along with other control-flow bindings such as if and with.

Example 1: Iterating over an array

This example uses foreach to produce a read-only table with a row for each array entry.

```
<table>
  <thead>
    <tr><th>First name</th><th>Last name</th></tr>
  </thead>
  <tbody data-bind="foreach: people">
    <tr>
      <td data-bind="text: firstName"></td>
      <td data-bind="text: lastName"></td>
    </tr>
  </tbody>
</table>

<script type="text/javascript">
  ko.applyBindings({
    people: [
      { firstName: 'Bert', lastName: 'Bertington' },
      { firstName: 'Charles', lastName: 'Charlesforth' },
      { firstName: 'Denise', lastName: 'Dentiste' }
    ]
  });
</script>
```

Example 2: Live example with add/remove

The following example shows that, if your array is observable, then the UI will be kept in sync with changes to that array.

People

- Name at position 0: Bert [Remove](#)
- Name at position 1: Charles [Remove](#)
- Name at position 2: Denise [Remove](#)

Add

Source code: View

```
<h4>People</h4>
<ul data-bind="foreach: people">
  <li>
    Name at position <span data-bind="text: $index"> </span>:
    <span data-bind="text: name"> </span>
    <a href="#" data-bind="click: $parent.removePerson">Remove</a>
  </li>
</ul>
<button data-bind="click: addPerson">Add</button>
```

Source code: View model

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

  self.people = ko.observableArray([
    { name: 'Bert' },
    { name: 'Charles' },
    { name: 'Denise' }
  ]);

  self.addPerson = function() {
    self.people.push({ name: "New at " + new Date() });
  };

  self.removePerson = function() {
    self.people.remove(this);
  }
}
```

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

Parameters

- Main parameter
 - Pass the array that you wish to iterate over. The binding will output a section of markup for each entry.
 - Alternatively, pass a JavaScript object literal with a property called `data` which is the array you wish to iterate over. The object literal may also have other properties, such as `afterAdd` or `includeDestroyed`— see below for details of these extra options and examples of their use.
 - If the array you supply is observable, the `foreach` binding will respond to any future changes in the array's contents by adding or removing corresponding sections of markup in the DOM.
- Additional parameters
 - None

Note 1: Referring to each array entry using `$data`

As shown in the above examples, bindings within the `foreach` block can refer to properties on the array entries. For example, Example 1 referenced the `firstName` and `lastName` properties on each array entry.

But what if you want to refer to the array entry itself (not just one of its properties)? In that case, you can use the special context property `$data`. Within a `foreach` block, it means “the current item”. For example,

```
<ul data-bind="foreach: months">
  <li>
    The current item is: <b data-bind="text: $data"></b>
  </li>
</ul>

<script type="text/javascript">
  ko.applyBindings({
    months: [ 'Jan', 'Feb', 'Mar', 'etc' ]
  });
</script>
```

If you wanted, you could use `$data` as a prefix when referencing properties on each entry. For example, you could rewrite part of Example 1 as follows:

```
<td data-bind="text: $data.firstName"></td>
```

... but you don't have to, because `firstName` will be evaluated within the context of `$data` by default anyway.

Note 2: Using `$index`, `$parent`, and other context properties

As you can see from Example 2 above, it's possible to use **`$index`** to refer to the zero-based index of the current array item. `$index` is an observable and is updated whenever the index of the item changes (e.g., if items are added to or removed from the array).

Similarly, you can use `$parent` to refer to data from outside the `foreach`, e.g.:

```
<h1 data-bind="text: blogPostTitle"></h1>
<ul data-bind="foreach: likes">
  <li>
    <b data-bind="text: name"></b> likes the blog post <b data-bind="text:
$parent.blogPostTitle"></b>
  </li>
</ul>
```

For more information about `$index` and other context properties such as `$parent`, see documentation for binding context properties.

Note 3: Using `foreach` without a container element

In some cases, you might want to duplicate a section of markup, but you don't have any container element on which to put a `foreach` binding. For example, you might want to generate the following:

```
<ul>
  <li class="header">Header item</li>
  <!-- The following are generated dynamically from an array -->
  <li>Item A</li>
  <li>Item B</li>
  <li>Item C</li>
</ul>
```

In this example, there isn't anywhere to put a normal `foreach` binding. You can't put it on the `` (because then you'd be duplicating the header item), nor can you put a further container inside the `` (because only `` elements are allowed inside ``s).

To handle this, you can use the *containerless control flow syntax*, which is based on comment tags. For example,

```
<ul>
  <li class="header">Header item</li>
  <!-- ko foreach: myItems -->
    <li>Item <span data-bind="text: $data"></span></li>
  <!-- /ko -->
</ul>

<script type="text/javascript">
  ko.applyBindings({
    myItems: [ 'A', 'B', 'C' ]
  });
</script>
```

The `<!-- ko -->` and `<!-- /ko -->` comments act as start/end markers, defining a “virtual element” that contains the markup inside. Knockout understands this virtual element syntax and binds as if you had a real container element.

Note 4: Destroyed entries are hidden by default

Sometimes you may want to mark an array entry as **deleted**, but without actually losing record of its existence. This is known as a *non-destructive delete*. For details of how to do this, see the `destroy` function on `observableArray`.

By default, the `foreach` binding will skip over (i.e., hide) any array entries that are marked as destroyed. If you want to show destroyed entries, use the **includeDestroyed** option. For example,

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

Note 5: Post-processing or animating the generated DOM elements

If you need to run some further custom logic on the generated DOM elements, you can use any of the following callbacks:

- `afterRender` — is invoked each time the `foreach` block is duplicated and inserted into the document, both when `foreach` first initializes, and when new entries are added to the associated array later. Knockout will supply the following parameters to your callback:

- An array of the inserted DOM elements
 - The data item against which they are being bound
- `afterAdd` — is like `afterRender`, except it is invoked only when new entries are added to your array (and *not* when `foreach` first iterates over your array's initial contents). A common use for `afterAdd` is to call a method such as jQuery's `$(domNode).fadeIn()` so that you get animated transitions whenever items are added. Knockout will supply the following parameters to your callback:
 - A DOM node being added to the document
 - The index of the added array element
 - The added array element
- `beforeRemove` — is invoked when an array item has been removed, but before the corresponding DOM nodes have been removed. If you specify a `beforeRemove` callback, then *it becomes your responsibility to remove the DOM nodes*. The obvious use case here is calling something like jQuery's `$(domNode).fadeOut()` to animate the removal of the corresponding DOM nodes — in this case, Knockout cannot know how soon it is allowed to physically remove the DOM nodes (who knows how long your animation will take?), so it is up to you to remove them. Knockout will supply the following parameters to your callback:
 - A DOM node that you should remove
 - The index of the removed array element
 - The removed array element

Here's a trivial example that uses `afterRender`. It simply uses jQuery's `$.css` to make the rendered element turn red:

```
<ul data-bind="foreach: { data: myItems, afterRender: handleAfterRender }">
  <li data-bind="text: $data"></li>
</ul>

<script type="text/javascript">
  ko.applyBindings({
    myItems: ko.observableArray([ 'A', 'B', 'C' ]),
    handleAfterRender: function(elements, data) {
      $(elements).css({ color: 'red' });
    }
  });
</script>
```

```
});  
</script>
```

For examples of `afterAdd` and `beforeRemove` see animated transitions.

Dependencies

None, other than the core Knockout library.

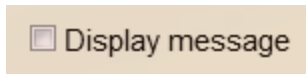
The "if" binding

The if binding causes a section of markup to appear in your document (and to have its data-bind attributes applied), only if a specified expression evaluates to true (or a true-ish value such as a non-null object or nonempty string).

if plays a similar role to the visible binding. The difference is that, with visible, the contained markup always remains in the DOM and always has its data-bind attributes applied - the visible binding just uses CSS to toggle the container element's visibility. The if binding, however, **physically** adds or removes the contained markup in your DOM, and only applies bindings to descendants if the expression is true.

Example 1

This example shows that the if binding can dynamically add and remove sections of markup as observable values change.



Source code: View

```
<label><input type="checkbox" data-bind="checked: displayMessage" /> Display message</label>  
<div data-bind="if: displayMessage">Here is a message. Astonishing.</div>
```

Source code: View model

```
ko.applyBindings({  
  displayMessage: ko.observable(false)  
});
```

Example 2

In the following example, the `<div>` element will be empty for "Mercury", but populated for "Earth". That's because Earth has a non-null capital property, whereas "Mercury" has null for that property.

```
<ul data-bind="foreach: planets">  
  <li>
```

```
Planet: <b data-bind="text: name"> </b>
<div data-bind="if: capital">
  Capital: <b data-bind="text: capital.cityName"> </b>
</div>
</li>
</ul>

<script>
ko.applyBindings({
  planets: [
    { name: 'Mercury', capital: null },
    { name: 'Earth', capital: { cityName: 'Barnsley' } }
  ]
});
</script>
```

It's important to understand that the if binding really is vital to make this code work properly. Without it, there would be an error when trying to evaluate `capital.cityName` in the context of "Mercury" where `capital` is null. In JavaScript, you're not allowed to evaluate subproperties of null or undefined values.

Parameters

- Main parameter
 - The expression you wish to evaluate. If it evaluates to true (or a true-ish value), the contained markup will be present in the document, and any `data-bind` attributes on it will be applied. If your expression evaluates to false, the contained markup will be removed from your document without first applying any bindings to it.
 - If your expression involves any observable values, the expression will be re-evaluated whenever any of them change. Correspondingly, the markup within your if block can be added or removed dynamically as the result of the expression changes. `data-bind` attributes will be applied to a new copy of the contained markup whenever it is re-added.
- Additional parameters
 - None

Note: Using "if" without a container element

Sometimes you may want to control the presence/absence of a section of markup *without* having any container element that can hold an if binding. For example, you might want to control whether a certain `` element appears alongside siblings that always appear:

```
<ul>
  <li>This item always appears</li>
  <li>I want to make this item present/absent dynamically</li>
</ul>
```

In this case, you can't put `if` on the `` (because then it would affect the first `` too), and you can't put any other container around the second `` (because HTML doesn't allow extra containers within ``s).

To handle this, you can use the *containerless control flow syntax*, which is based on comment tags. For example,

```
<ul>
  <li>This item always appears</li>
  <!-- ko if: someExpressionGoesHere -->
    <li>I want to make this item present/absent dynamically</li>
  <!-- /ko -->
</ul>
```

The `<!-- ko -->` and `<!-- /ko -->` comments act as start/end markers, defining a "virtual element" that contains the markup inside. Knockout understands this virtual element syntax and binds as if you had a real container element.

Dependencies

None, other than the core Knockout library.

The "ifnot" binding

The `ifnot` binding is exactly the same as the `if` binding, except that it inverts the result of whatever expression you pass to it. For more details, see documentation for the `if` binding.

Note: "ifnot" is the same as a negated "if"

The following markup:

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

... is equivalent to the following:

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

... assuming that `someProperty` is *observable* and hence you need to invoke it as a function to obtain the current value.

The only reason to use `ifnot` instead of a negated `if` is just as a matter of taste: many developers feel that it looks tidier.

The "with" binding

The `with` binding creates a new binding context, so that descendant elements are bound in the context of a specified object.

Of course, you can arbitrarily nest `with` bindings along with the other control-flow bindings such as `if` and `foreach`.

Example 1

Here is a very basic example of switching the binding context to a child object. Notice that in the `data-bind` attributes, it is *not* necessary to prefix `latitude` or `longitude` with `coords.`, because the binding context is switched to `coords`.

```
<h1 data-bind="text: city"> </h1>
<p data-bind="with: coords">
  Latitude: <span data-bind="text: latitude"> </span>,
  Longitude: <span data-bind="text: longitude"> </span>
</p>

<script type="text/javascript">
  ko.applyBindings({
    city: "London",
    coords: {
      latitude: 51.5001524,
      longitude: -0.1262362
    }
  });
</script>
```

Example 2

This interactive example demonstrates that:

- The with binding will dynamically add or remove descendant elements depending on whether the associated value is null/undefined or not
- If you want to access data/functions from parent binding contexts, you can use special context properties such as \$parent and root.

Try it out:

Twitter account:

Source code: View

```
<form data-bind="submit: getTweets">
  Twitter account:
  <input data-bind="value: twitterName" />
  <button type="submit">Get tweets</button>
</form>

<div data-bind="with: resultData">
  <h3>Recent tweets fetched at <span data-bind="text: retrievalDate"> </span></h3>
  <ol data-bind="foreach: topTweets">
    <li data-bind="text: text"></li>
  </ol>

  <button data-bind="click: $parent.clearResults">Clear tweets</button>
</div>
```

Source code: View model

```
function AppViewModel() {
  var self = this;
  self.twitterName = ko.observable('@StephenFry');
  self.resultData = ko.observable(); // No initial value

  self.getTweets = function() {
    twitterApi.getTweetsForUser(self.twitterName(), function(data) {
      self.resultData({
        retrievalDate: new Date(),
        topTweets: data.slice(0, 5)
      });
    });
  };
}
```

```
});  
}  
  
self.clearResults = function() {  
    self.resultData(undefined);  
}  
}  
ko.applyBindings(new AppViewModel());
```

Parameters

- Main parameter
 - The object that you want to use as the context for binding descendant elements.
 - If the expression you supply evaluates to null or undefined, descendant elements will *not* be bound at all, but will instead be removed from the document.
 - If the expression you supply involves any observable values, the expression will be re-evaluated whenever any of those observables change. Then, descendant elements will be cleared out, and a new copy of the markup will be added to your document and bound in the context of the new evaluation result.
- Additional parameters
 - None

Note 1: Using “with” without a container element

Just like other control flow elements such as `if` and `foreach`, you can use `with` without any container element to host it. This is useful if you need to use `with` in a place where it would not be legal to introduce a new container element just to hold the `with` binding. See the documentation for `if` or `foreach` for more details.

Example:

```
<ul>  
  <li>Header element</li>  
  <!-- ko with: outboundFlight -->  
  ...  
  <!-- /ko -->  
  <!-- ko with: inboundFlight -->  
  ...
```

```
<!-- /ko -->
</ul>
```

The `<!-- ko -->` and `<!-- /ko -->` comments act as start/end markers, defining a “virtual element” that contains the markup inside. Knockout understands this virtual element syntax and binds as if you had a real container element.

Dependencies

None, other than the core Knockout library.

3. Working with form fields

The "click" binding

The click binding adds an event handler so that your chosen JavaScript function will be invoked when the associated DOM element is clicked. This is most commonly used with elements like button, input, and a, but actually works with any visible DOM element.

Example

```
<div>
  You've clicked <span data-bind="text: numberOfClicks"></span> times
  <button data-bind="click: incrementClickCounter">Click me</button>
</div>

<script type="text/javascript">
  var viewModel = {
    numberOfClicks : ko.observable(0),
    incrementClickCounter : function() {
      var previousCount = this.numberOfClicks();
      this.numberOfClicks(previousCount + 1);
    }
  };
</script>
```

Each time you click the button, this will invoke `incrementClickCounter()` on the view model, which in turn changes the view model state, which causes the UI to update.

Parameters

- Main parameter

- The function you want to bind to the element's click event.
- You can reference any JavaScript function - it doesn't have to be a function on your view model. You can reference a function on any object by writing click:
someObject.someFunction.
- Additional parameters
 - None

Note 1: Passing a “current item” as a parameter to your handler function

When calling your handler, Knockout will supply the **current model value** as the first parameter. This is particularly useful if you're rendering some UI for each item in a collection, and you need to know which item's UI was clicked. For example,

```
<ul data-bind="foreach: places">
  <li>
    <span data-bind="text: $data"></span>
    <button data-bind="click: $parent.removePlace">Remove</button>
  </li>
</ul>

<script type="text/javascript">
  function MyViewModel() {
    var self = this;
    self.places = ko.observableArray(['London', 'Paris', 'Tokyo']);

    // The current item will be passed as the first parameter, so we know which place to remove
    self.removePlace = function(place) {
      self.places.remove(place)
    }
  }
  ko.applyBindings(new MyViewModel());
</script>
```

Two points to note about this example:

- If you're inside a nested binding context, for example if you're inside a foreach or a with block, but your handler function is on the root viewmodel or some other parent context, you'll need to use a prefix such as `$parent` or `$root` to locate the handler function.

- In your viewmodel, it's often useful to declare `self` (or some other variable) as an alias for this. Doing so avoids any problems with this being redefined to mean something else in event handlers or Ajax request callbacks.

Note 2: Accessing the event object, or passing more parameters

In some scenarios, you may need to access the DOM event object associated with your click event.

Knockout will pass the event as the second parameter to your function, as in this example:

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

<script type="text/javascript">
  var viewModel = {
    myFunction: function(data, event) {
      if (event.shiftKey) {
        //do something different when user has shift key down
      } else {
        //do normal action
      }
    }
  };
  ko.applyBindings(viewModel);
</script>
```

If you need to pass more parameters, one way to do it is by wrapping your handler in a function literal that takes in a parameter, as in this example:

```
<button data-bind="click: function(data, event) { myFunction('param1', 'param2', data, event) }">
  Click me
</button>
```

Now, KO will pass the data and event objects to your function literal, which are then available to be passed to your handler.

Alternatively, if you prefer to avoid the function literal in your view, you can use the `bind` function, which attaches specific parameter values to a function reference:

```
<button data-bind="click: myFunction.bind($data, 'param1', 'param2')">
  Click me
```

```
</button>
```

Note 3: Allowing the default click action

By default, Knockout will prevent the click event from taking any default action. This means that if you use the click binding on an a tag (a link), for example, the browser will only call your handler function and will *not* navigate to the link's href. This is a useful default because when you use the click binding, it's normally because you're using the link as part of a UI that manipulates your view model, not as a regular hyperlink to another web page.

However, if you *do* want to let the default click action proceed, just **return true from your click handler function**.

Note 4: Preventing the event from bubbling

By default, Knockout will **allow** the click event to continue to bubble up to any higher level event handlers. For example, if your element and a parent of that element are both handling the click event, then the click handler for both elements will be triggered. If necessary, you can prevent the event from bubbling by including an additional binding that is named `clickBubble` and passing `false` to it, as in this example:

```
<div data-bind="click: myDivHandler">
  <button data-bind="click: myButtonHandler, clickBubble: false">
    Click me
  </button>
</div>
```

Normally, in this case `myButtonHandler` would be called first, then the click event would bubble up to `myDivHandler`. However, the `clickBubble` binding that we added with a value of `false` prevents the event from making it past `myButtonHandler`.

Dependencies

None, other than the core Knockout library.

The "event" binding

The event binding allows you to add an event handler for a specified event so that your chosen JavaScript function will be invoked when that event is triggered for the associated DOM element. This can be used to bind to any event, such as `keypress`, `mouseover` or `mouseout`.

Example

```
<div>
  <div data-bind="event: { mouseover: enableDetails, mouseout: disableDetails }">
```

```
    Mouse over me
  </div>
  <div data-bind="visible: detailsEnabled">
    Details
  </div>
</div>

<script type="text/javascript">
  var viewModel = {
    detailsEnabled: ko.observable(false),
    enableDetails: function() {
      this.detailsEnabled(true);
    },
    disableDetails: function() {
      this.detailsEnabled(false);
    }
  };
  ko.applyBindings(viewModel);
</script>
```

Now, moving your mouse pointer on or off of the first element will invoke methods on the view model to toggle the `detailsEnabled` observable. The second element reacts to changes to the value of `detailsEnabled` by either showing or hiding itself.

Parameters

- Main parameter
 - You should pass a JavaScript object in which the property names correspond to event names, and the values correspond to the function that you want to bind to the event.
 - You can reference any JavaScript function - it doesn't have to be a function on your view model. You can reference a function on any object by writing `event { mouseover: someObject.someFunction }`.
- Additional parameters
 - None

Note 1: Passing a “current item” as a parameter to your handler function

When calling your handler, Knockout will supply the **current model value** as the first parameter. This is particularly useful if you're rendering some UI for each item in a collection, and you need to know which item the event refers to. For example,

```
<ul data-bind="foreach: places">
  <li data-bind="text: $data, event: { mouseover: $parent.logMouseOver }"> </li>
</ul>
<p>You seem to be interested in: <span data-bind="text: lastInterest"> </span></p>

<script type="text/javascript">
  function MyViewModel() {
    var self = this;
    self.lastInterest = ko.observable();
    self.places = ko.observableArray(['London', 'Paris', 'Tokyo']);

    // The current item will be passed as the first parameter, so we know which place was
    hovered over
    self.logMouseOver = function(place) {
      self.lastInterest(place);
    }
  }
  ko.applyBindings(new MyViewModel());
</script>
```

Two points to note about this example:

- If you're inside a nested binding context, for example if you're inside a `foreach` or a `with` block, but your handler function is on the root viewmodel or some other parent context, you'll need to use a prefix such as `$parent` or `$root` to locate the handler function.
- In your viewmodel, it's often useful to declare `self` (or some other variable) as an alias for `this`. Doing so avoids any problems with this being redefined to mean something else in event handlers or Ajax request callbacks.

Note 2: Accessing the event object, or passing more parameters

In some scenarios, you may need to access the DOM event object associated with your event. Knockout will pass the event as the second parameter to your function, as in this example:

```
<div data-bind="event: { mouseover: myFunction }">
```

```
    Mouse over me
</div>

<script type="text/javascript">
    var viewModel = {
        myFunction: function(data, event) {
            if (event.shiftKey) {
                //do something different when user has shift key down
            } else {
                //do normal action
            }
        }
    };
    ko.applyBindings(viewModel);
</script>
```

If you need to pass more parameters, one way to do it is by wrapping your handler in a function literal that takes in a parameter, as in this example:

```
<div data-bind="event: { mouseover: function(data, event) { myFunction('param1', 'param2', data,
event) } }">
    Mouse over me
</div>
```

Now, KO will pass the event to your function literal, which is then available to be passed to your handler.

Alternatively, if you prefer to avoid the function literal in your view, you can use the `bind` function, which attaches specific parameter values to a function reference:

```
<button data-bind="event: { mouseover: myFunction.bind($data, 'param1', 'param2') }">
    Click me
</button>
```

Note 3: Allowing the default action

By default, Knockout will prevent the event from taking any default action. For example if you use the eventbinding to capture the keypress event of an input tag, the browser will only call your handler function and will *not* add the value of the key to the input element's value. A more common example is using the click binding, which internally uses this binding, where your handler function will be called, but the browser will *not* navigate to the link's href. This is a useful default because when you use

the click binding, it's normally because you're using the link as part of a UI that manipulates your view model, not as a regular hyperlink to another web page.

However, if you *do* want to let the default action proceed, just return true from your event handler function.

Note 4: Preventing the event from bubbling

By default, Knockout will allow the event to continue to bubble up to any higher level event handlers. For example, if your element is handling a mouseover event and a parent of the element also handles that same event, then the event handler for both elements will be triggered. If necessary, you can prevent the event from bubbling by including an additional binding that is named `youreventBubble` and passing false to it, as in this example:

```
<div data-bind="event: { mouseover: myDivHandler }">
  <button data-bind="event: { mouseover: myButtonHandler }, mouseoverBubble: false">
    Click me
  </button>
</div>
```

Normally, in this case `myButtonHandler` would be called first, then the event would bubble up to `myDivHandler`. However, the `mouseoverBubble` binding that we added with a value of false prevents the event from making it past `myButtonHandler`.

Dependencies

None, other than the core Knockout library.

The "submit" binding

The submit binding adds an event handler so that your chosen JavaScript function will be invoked when the associated DOM element is submitted. Typically you will only want to use this on form elements.

When you use the submit binding on a form, Knockout will prevent the browser's default submit action for that form. In other words, the browser will call your handler function but will *not* submit the form to the server. This is a useful default because when you use the submit binding, it's normally because you're using the form as an interface to your view model, not as a regular HTML form. If you *do* want to let the form submit like a normal HTML form, just return true from your submit handler.

Example

```
<form data-bind="submit: doSomething">
  ... form contents go here ...
</form>
```

```
<button type="submit">Submit</button>
</div>

<script type="text/javascript">
  var viewModel = {
    doSomething : function(formElement) {
      // ... now do something
    }
  };
</script>
```

As illustrated in this example, KO passes the **form element** as a parameter to your submit handler function. You can ignore that parameter if you want, but for an example of when it's useful to have a reference to that element, see the docs for the **ko.postJson** utility.

Why not just put a click handler on the submit button?

Instead of using `submit` on the form, you *could* use `click` on the submit button. However, `submit` has the advantage that it also captures alternative ways to submit the form, such as pressing the *enter* key while typing into a text box.

Parameters

- Main parameter
 - The function you want to bind to the element's submit event.
 - You can reference any JavaScript function - it doesn't have to be a function on your view model. You can reference a function on any object by writing `submit: someObject.someFunction`.
 - Functions on your view model are slightly special because you can reference them by name, i.e., you can write `submit: doSomething` and *don't* have to write `submit: viewModel.doSomething` (though technically that's also valid).
- Additional parameters
 - None

Notes

For information about how to pass additional parameters to your submit handler function, or how to control the `this` handle when invoking functions that aren't on your view model, see the notes relating to the click binding. All the notes on that page apply to submit handlers too.

Dependencies

None, other than the core Knockout library.

The "enable" binding

The enable binding causes the associated DOM element to be enabled only when the parameter value is true. This is useful with form elements like input, select, and textarea.

Example

```
<p>
  <input type='checkbox' data-bind="checked: hasCellphone" />
  I have a cellphone
</p>
<p>
  Your cellphone number:
  <input type='text' data-bind="value: cellphoneNumber, enable: hasCellphone" />
</p>

<script type="text/javascript">
  var viewModel = {
    hasCellphone : ko.observable(false),
    cellphoneNumber: ""
  };
</script>
```

In this example, the "Your cellphone number" text box will initially be disabled. It will be enabled only when the user checks the box labelled "I have a cellphone".

Parameters

- Main parameter
 - A value that controls whether or not the associated DOM element should be enabled.
 - Non-boolean values are interpreted loosely as boolean. For example, 0 and null are treated as false, whereas 21 and non-null objects are treated as true.
 - If your parameter references an observable value, the binding will update the enabled/disabled state whenever the observable value changes. If the parameter doesn't

reference an observable value, it will only set the state once and will not do so again later.

- Additional parameters
 - None

Note: Using arbitrary JavaScript expressions

You're not limited to referencing variables - you can reference arbitrary expressions to control an element's enabledness. For example,

```
<button data-bind="enable: parseAreaCode(viewModel.cellphoneNumber()) != '555'">
  Do something
</button>
```

Dependencies

None, other than the core Knockout library.

The "disable" binding

The disable binding causes the associated DOM element to be disabled only when the parameter value is true. This is useful with form elements like input, select, and textarea.

This is the mirror image of the enable binding. For more information, see documentation for the enable binding, because disable works in exactly the same way except that it negates whatever parameter you pass to it.

The "value" binding

The value binding links the associated DOM element's value with a property on your view model. This is typically useful with form elements such as <input>, <select> and <textarea>.

When the user edits the value in the associated form control, it updates the value on your view model. Likewise, when you update the value in your view model, this updates the value of the form control on screen.

Note: If you're working with checkboxes or radio buttons, use the checked binding to read and write your element's checked state, not the value binding.

Example

```
<p>Login name: <input data-bind="value: userName" /></p>
<p>Password: <input type="password" data-bind="value: userPassword" /></p>
```

```
<script type="text/javascript">
  var viewModel = {
    userName: ko.observable(""), // Initially blank
    userPassword: ko.observable("abc"), // Prepopulate
  };
</script>
```

Parameters

- Main parameter
 - KO sets the element's value property to your parameter value. Any previous value will be overwritten.
 - If this parameter is an observable value, the binding will update the element's value whenever the value changes. If the parameter isn't observable, it will only set the element's value once and will not update it again later.
 - If you supply something other than a number or a string (e.g., you pass an object or an array), the displayed text will be equivalent to `yourParameter.toString()` (that's usually not very useful, so it's best to supply string or numeric values).
 - Whenever the user edits the value in the associated form control, KO will update the property on your view model. By default, KO updates your view model when the user transfers focus to another DOM node (i.e., on the change event), but you can control when the value is updated using the `valueUpdate` parameter described below.
- Additional parameters
 - `valueUpdate`

If your binding also includes a parameter called `valueUpdate`, this defines which browser event KO should use to detect changes. The following string values are the most commonly useful choices:

- "change" (default) - updates your view model when the user moves the focus to a different control, or in the case of `<select>` elements, immediately after any change
- "keyup" - updates your view model when the user releases a key

- "keypress" - updates your view model when the user has typed a key. Unlike keyup, this updates repeatedly while the user holds a key down
- "afterkeydown" - updates your view model as soon as the user begins typing a character. This works by catching the browser's keydown event and handling the event asynchronously.

Of these options, "**afterkeydown**" is the best choice if you want to keep your view model updated in real-time.

Example:

```
<p>Your value: <input data-bind="value: someValue, valueUpdate: 'afterkeydown'" /></p>
<p>You have typed: <span data-bind="text: someValue"></span></p> <!-- updates in real-time -->

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

Note 1: Working with drop-down lists (i.e., SELECT nodes)

Knockout has special support for drop-down lists (i.e., `<select>` nodes). The value binding works in conjunction with the options binding to let you read and write values that are arbitrary JavaScript objects, not just string values. This is very useful if you want to let the user select from a set of **model objects**. For examples of this, see the options binding

Similarly, if you want to create a multi-select list, see the documentation for the `selectedOptions` binding.

Note 2: Updating observable and non-observable property values

If you use `value` to link a form element to an observable property, KO is able to set up a 2-way binding so that changes to either affect the other.

However, if you use `value` to link a form element to a *non*-observable property (e.g., a plain old string, or an arbitrary JavaScript expression), KO will do the following:

* If you reference a *simple property*, i.e., it is just a regular property on your view model, KO will set the form element's initial state to the property value, and when the form element is edited, KO will write the changes back to your property. It cannot detect when the property changes (because it isn't observable), so this is only a 1-way binding.

* If you reference something that is *not* a simple property, e.g., a complex JavaScript expression or a sub-property, KO will set the form element's initial state to that value, but it will not be able to write any changes back when the user edits the form element. In this case it's a one-time-only value setter, not a real binding.

Example:

```
<p>First value: <input data-bind="value: firstValue" /></p>      <!-- two-way binding -->
<p>Second value: <input data-bind="value: secondValue" /></p>    <!-- one-way binding -->
<p>Third value: <input data-bind="value: secondValue.length" /></p> <!-- no binding -->

<script type="text/javascript">
  var viewModel = {
    firstValue: ko.observable("hello"), // Observable
    secondValue: "hello, again"       // Not observable
  };
</script>
```

Dependencies

None, other than the core Knockout library.

The "hasfocus" binding

The hasfocus binding links a DOM element's focus state with a viewmodel property. It is a two-way binding, so:

- If you set the viewmodel property to true or false, the associated element will become focused or unfocused.
- If the user manually focuses or unfocuses the associated element, the viewmodel property will be set to true or false accordingly.

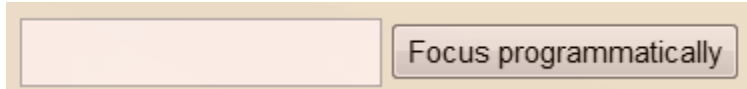
This is useful if you're building sophisticated forms in which editable elements appear dynamically, and you would like to control where the user should start typing, or respond to the location of the caret.

Tip

If multiple elements have hasfocus bindings with associated values set to true, the browser will switch focus to whichever element had its hasfocus binding set **most recently**. So, you can simply write `data-bind="hasfocus: true"` if you want to make an element gain focus as soon as it is dynamically inserted into the document. This will not prevent the focus from later moving to a different element.

Example 1: The basics

This example simply displays a message if the textbox currently has focus, and uses a button to show that you can trigger focus programmatically.



Source code: View

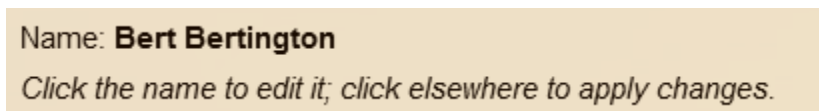
```
<input data-bind="hasfocus: isSelected" />
<button data-bind="click: setIsSelected">Focus programmatically</button>
<span data-bind="visible: isSelected">The textbox has focus</span>
```

Source code: View model

```
var viewModel = {
  isSelected: ko.observable(false),
  setIsSelected: function() { this.isSelected(true) }
};
ko.applyBindings(viewModel);
```

Example 2: Click-to-edit

Because the `hasfocus` binding works in both directions (setting the associated value focuses or unfocuses the element; focusing or unfocusing the element sets the associated value), it's a convenient way to toggle an "edit" mode. In this example, the UI displays either a `` or an `<input>` element depending on the model's `editing` property. Unfocusing the `<input>` element sets editing to false, so the UI switches out of "edit" mode.



Source code: View

```
<p>
  Name:
  <b data-bind="visible: !editing(), text: name, click: edit">&nbsp;</b>
  <input data-bind="visible: editing, value: name, hasfocus: editing" />
</p>
<p><em>Click the name to edit it; click elsewhere to apply changes.</em></p>
```

Source code: View model

```
function PersonViewModel(name) {
```

```
// Data
this.name = ko.observable(name);
this.editing = ko.observable(false);

// Behaviors
this.edit = function() { this.editing(true) }
}

ko.applyBindings(new PersonViewModel("Bert Bertington"));
```

Parameters

- Main parameter
 - Pass true (or some value that evaluates as true) to focus the associated element. Otherwise, the associated element will be unfocused.
 - When the user manually focuses or unfocuses the element, your value will be set to true or false accordingly.
 - If the value you supply is observable, the hasfocus binding will update the element's focus state whenever that observable value changes.
- Additional parameters
 - None

Dependencies

None, other than the core Knockout library.

The "checked" binding

The checked binding links a checkable form control — i.e., a checkbox (`<input type='checkbox'>`) or a radio button (`<input type='radio'>`) — with a property on your view model.

When the user checks the associated form control, this updates the value on your view model. Likewise, when you update the value in your view model, this checks or unchecks the form control on screen.

Note: For text boxes, drop-down lists, and all non-checkable form controls, use the value binding to read and write the element's value, not the checked binding.

Example with checkbox

```
<p>Send me spam: <input type="checkbox" data-bind="checked: wantsSpam" /></p>
```

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

  // ... then later ...
  viewModel.wantsSpam(false); // The checkbox becomes unchecked
</script>
```

Example adding checkboxes bound to an array

```
<p>Send me spam: <input type="checkbox" data-bind="checked: wantsSpam" /></p>
<div data-bind="visible: wantsSpam">
  Preferred flavors of spam:
  <div><input type="checkbox" value="cherry" data-bind="checked: spamFlavors" /> Cherry</div>
  <div><input type="checkbox" value="almond" data-bind="checked: spamFlavors" /> Almond</div>
  <div><input type="checkbox" value="msg" data-bind="checked: spamFlavors" /> Monosodium
  Glutamate</div>
</div>

<script type="text/javascript">
  var viewModel = {
    wantsSpam: ko.observable(true),
    spamFlavors: ko.observableArray(["cherry", "almond"]) // Initially checks the Cherry and Almond
    checkboxes
  };

  // ... then later ...
  viewModel.spamFlavors.push("msg"); // Now additionally checks the Monosodium Glutamate checkbox
</script>
```

Example adding radio buttons

```
<p>Send me spam: <input type="checkbox" data-bind="checked: wantsSpam" /></p>
<div data-bind="visible: wantsSpam">
  Preferred flavor of spam:
```



```
<div><input type="radio" name="flavorGroup" value="cherry" data-bind="checked: spamFlavor" />
Cherry</div>
  <div><input type="radio" name="flavorGroup" value="almond"
data-bind="checked: spamFlavor" /> Almond</div>
  <div><input type="radio" name="flavorGroup" value="msg" data-bind="checked: spamFlavor" />
Monosodium Glutamate</div>
</div>

<script type="text/javascript">
  var viewModel = {
    wantsSpam: ko.observable(true),
    spamFlavor: ko.observable("almond") // Initially selects only the Almond radio button
  };

  // ... then later ...
  viewModel.spamFlavor("msg"); // Now only Monosodium Glutamate is checked
</script>
```

Parameters

- Main parameter

KO sets the element's checked state to match your parameter value. Any previous checked state will be overwritten. The way your parameter is interpreted depends on what type of element you're binding to:

- For **checkboxes**, KO will set the element to be *checked* when the parameter value is true, and *unchecked* when it is false. If you give a value that isn't actually boolean, it will be interpreted loosely. This means that nonzero numbers and non-null objects and non-empty strings will all be interpreted as true, whereas zero, null, undefined, and empty strings will be interpreted as false.

When the user checks or unchecks the checkbox, KO will set your model property to true or false accordingly.

Special consideration is given if your parameter resolves to an array. In this case, KO will set the element to be *checked* if the value matches an item in the array, and *unchecked* if it is not contained in the array.

When the user checks or unchecks the checkbox, KO will add or remove the value from the array accordingly.

- For **radio buttons**, KO will set the element to be *checked* if and only if the parameter value equals the radio button node's value attribute. So, your parameter value should be a string. In the previous example, the radio button with value="almond" was checked only when the view model's spamFlavor property was equal to "almond".

When the user changes which radio button is selected, KO will set your model property to equal the value attribute of the selected radio button. In the preceding example, clicking on the radio button with value="cherry" would set viewModel.spamFlavor to be "cherry".

Of course, this is most useful when you have multiple radio button elements bound to a single model property. To ensure that only *one* of those radio buttons can be checked at any one time, you should set all their name attributes to an arbitrary common value (e.g., the value flavorGroup in the preceding example) - doing this puts them into a group where only one can be selected.

If your parameter is an observable value, the binding will update the element's checked state whenever the value changes. If the parameter isn't observable, it will only set the element's checked state once and will not update it again later.

- Additional parameters
 - None

The "options" binding

The options binding controls what options should appear in a drop-down list (i.e., a <select> element) or multi-select list (e.g., <select size='6'>). This binding cannot be used with anything other than <select>elements.

The value you assign should be an array (or observable array). The <select> element will then display one item for each item in your array.

Note: For a multi-select list, to set which of the options are selected, or to read which of the options are selected, use the **selectedOptions** binding. For a single-select list, you can also read and write the selected option using the value binding.

Example 1: Drop-down list

```
<p>Destination country: <select data-bind="options: availableCountries"></select></p>
```

```

<script type="text/javascript">
  var viewModel = {
    availableCountries : ko.observableArray(['France', 'Germany', 'Spain']) // These are the initial
options
  };

  // ... then later ...

  viewModel.availableCountries.push('China'); // Adds another option
</script>

```

Example 2: Multi-select list

```

<p>Choose some countries you'd like to visit: <select data-bind="options: availableCountries" size="5"
multiple="true"></select></p>

```

```

<script type="text/javascript">
  var viewModel = {
    availableCountries : ko.observableArray(['France', 'Germany', 'Spain'])
  };
</script>

```

Example 3: Drop-down list representing arbitrary JavaScript objects, not just strings

```

<p>
  Your country:
  <select data-bind="options: availableCountries, optionsText: 'countryName', value:
selectedCountry, optionsCaption: 'Choose...'"></select>
</p>

<div data-bind="visible: selectedCountry"> <!-- Appears when you select something -->
  You have chosen a country with population
  <span data-bind="text: selectedCountry() ? selectedCountry().countryPopulation :
'unknown'"></span>.
</div>

<script type="text/javascript">
  // Constructor for an object with two properties

```

```
var country = function(name, population) {
    this.countryName = name;
    this.countryPopulation = population;
};

var viewModel = {
    availableCountries : ko.observableArray([
        new country("UK", 65000000),
        new country("USA", 320000000),
        new country("Sweden", 29000000)
    ]),
    selectedCountry : ko.observable() // Nothing selected by default
};
</script>
```

Example 4: Drop-down list representing arbitrary JavaScript objects, with displayed text computed as a function of the represented item

```
<!-- Same as example 3, except the <select> box expressed as follows: -->
<select data-bind="options: availableCountries,
    optionsText: function(item) {
        return item.countryName + ' (pop: ' + item.countryPopulation + ')'
    },
    value: selectedCountry,
    optionsCaption: 'Choose...'"></select>
```

Note that the only difference between examples 3 and 4 is the optionsText value.

Parameters

- Main parameter
 - You should supply an array (or observable array). For each item, KO will add an `<option>` to the associated `<select>` node. Any previous options will be removed.
 - If your parameter's value is an array of strings, you don't need to give any other parameters. The `<select>` element will display an option for each string value. However, if you want to let the user choose from an array of *arbitrary JavaScript objects* (not merely strings), then see the optionsText and optionsValue parameters below.

- If this parameter is an observable value, the binding will update the element's available options whenever the value changes. If the parameter isn't observable, it will only set the element's available options once and will not update them again later.

- Additional parameters

- optionsCaption

Sometimes, you might not want to select any particular option by default. But a single-select drop-down list *always* has one item selected, so how can you avoid preselecting something? The usual solution is to prefix the list of options with a special dummy option that just reads "Select an item" or "Please choose an option" or similar, and have that one selected by default.

This easy to do: just add an additional parameter with name **optionsCaption**, with its value being a string to display. For example:

```
<select data-bind='options: myOptions, optionsCaption: "Select an item...", value: myChosenValue'></select>
```

KO will prefix the list of items with one that displays the text "Select an item..." and has the value undefined. So, if myChosenValue holds the value undefined (which observables do by default), then the dummy option will be selected.

- optionsText

See Example 3 above to see how you can bind options to an array of arbitrary JavaScript object - not just strings. In this case, you need to choose which of the objects' properties should be displayed as the text in the drop-down list or multi-select list. Example 3 shows how you can specify that property name by passing an additional parameter called optionsText.

If you don't want to display just a simple property value as the text for each item in the dropdown, you can pass a JavaScript function for the optionsText option and supply your own arbitrary logic for computing the displayed text in terms of the represented object. See Example 4 above, which shows how you could generate the displayed text by concatenating together multiple property values.

- optionsValue

Similar to optionsText, you can also pass an additional parameter called optionsValue to specify which of the objects' properties should be used to set the value attribute on the <option> elements that KO generates.

Typically you'd only want to do this as a way of ensuring that KO can correctly retain selection when you update the set of available options. For example, if you're repeatedly getting a list of "car" objects via Ajax calls and want to ensure that the selected car is preserved, you might need to set `optionsValue` to "carId" or whatever unique identifier each "car" object has, otherwise KO won't necessarily know which of the previous "car" objects corresponds to which of the new ones.

- o `selectedOptions`

For a multi-select list, you can read and write the selection state using `selectedOptions`.

Technically this is a separate binding, so it has its own documentation.

Note: Selection is preserved when setting/changing options

When the options binding changes the set of options in your `<select>` element, KO will leave the user's selection unchanged where possible. So, for a single-select drop-down list, the previously selected option value will still be selected, and for a multi-select list, all the previously selected option values will still be selected (unless, of course, you're removed one or more of those options).

That's because the options binding tries to be independent of the value binding (which controls selection for a single-select list) and the `selectedOptions` binding (which controls selection for a multi-select list).

The "selectedOptions" binding

The `selectedOptions` binding controls which elements in a multi-select list are currently selected. This is intended to be used in conjunction with a `<select>` element and the options binding.

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 your view model. Likewise, assuming it's an *observable* array on your view model, then whenever you add or remove (e.g., via `push` or `splice`) items to this array, the corresponding items in the UI become selected or deselected. It's a 2-way binding.

Note: To control which element in a single-select drop-down list is selected, you can use the `value` binding instead.

Example

```
<p>
  Choose some countries you'd like to visit:
  <select data-bind="options: availableCountries, selectedOptions: chosenCountries" size="5"
multiple="true"></select>
</p>
```

```
<script type="text/javascript">
  var viewModel = {
    availableCountries : ko.observableArray(['France', 'Germany', 'Spain']),
    chosenCountries : ko.observableArray(['Germany']) // Initially, only Germany is selected
  };

  // ... then later ...
  viewModel.chosenCountries.push('France'); // Now France is selected too
</script>
```

Parameters

- Main parameter
 - This should be an array (or an observable array). KO sets the element's selected options to match the contents of the array. Any previous selection state will be overwritten.
 - If your parameter is an observable array, the binding will update the element's selection whenever the array changes (e.g., via push, pop or other observable array methods). If the parameter isn't observable, it will only set the element's selection state once and will not update it again later.
 - Whether or not the parameter is an observable array, KO will detect when the user selects or deselects an item in the multi-select list, and will update the array to match. This is how you can read which of the options is selected.
- Additional parameters
 - None

Note: Letting the user select from arbitrary JavaScript objects

In the example code above, the user can choose from an array of string values. You're *not* limited to providing strings - your options array can contain arbitrary JavaScript objects if you wish.

See the options binding for details on how to control how arbitrary objects should be displayed in the list.

In this scenario, the values you can read and write using `selectedOptions` are those objects themselves, *not* their textual representations. This leads to much cleaner and more elegant code in most cases. Your view model can imagine that the user chooses from an array of arbitrary objects, without having to care how those objects are mapped to an on-screen representation.

The "uniqueName" binding

The uniqueName binding ensures that the associated DOM element has a nonempty name attribute. 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, e.g.:

* Other technologies may depend on the assumption that certain elements have names, even though names might be irrelevant when you're using KO. 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. See an example of using jQuery Validation with KO.

* IE 6 does not allow radio buttons to be checked if they don't have a name attribute. Most of the time this is irrelevant because your radio button elements *will* have name attributes to put them into mutually-exclusive groups. However, just in case you didn't add a name attribute because it's unnecessary in your case, KO will internally use uniqueName on those elements to ensure they can be checked.

Example

```
<input data-bind="value: someModelProperty, uniqueName: true" />
```

Parameters

- Main parameter
 - Pass true (or some value that evaluates as true) to enable the uniqueName binding, as in the preceding example.
- Additional parameters
 - None

Part IV: Creating custom bindings

1. Creating custom bindings

You're not limited to using the built-in bindings like click, value, and so on — you can create your own ones. This is how to control how observables interact with DOM elements, and gives you a lot of flexibility to encapsulate sophisticated behaviors in an easy-to-reuse way.

For example, you can create interactive components like grids, tabsets, and so on, in the form of custom bindings (see the grid example).

Registering your binding

To register a binding, add it as a subproperty of `ko.bindingHandlers`:

```
ko.bindingHandlers.yourBindingName = {
  init: function(element, valueAccessor, allBindingsAccessor, viewModel) {
    // This will be called when the binding is first applied to an element
    // Set up any initial state, event handlers, etc. here
  },
  update: function(element, valueAccessor, allBindingsAccessor, viewModel) {
    // This will be called once when the binding is first applied to an element,
    // and again whenever the associated observable changes value.
    // Update the DOM element based on the supplied values here.
  }
};
```

... and then you can use it on any number of DOM elements:

```
<div data-bind="yourBindingName: someValue"> </div>
```

Note: you don't actually have to provide both *init* and *update* callbacks — you can just provide one or the other if that's all you need.

The “update” callback

Whenever the associated observable changes, KO will call your update callback, passing the following parameters:

- `element` — The DOM element involved in this binding
- `valueAccessor` — A JavaScript function that you can call to get the current model property that is involved in this binding. Call this without passing any parameters (i.e., call `valueAccessor()`) to get the current model property value.
- `allBindingsAccessor` — A JavaScript function that you can call to get *all* the model properties bound to this DOM element. Like `valueAccessor`, call it without any parameters to get the current bound model properties.
- `viewModel` — The view model object that was passed to `ko.applyBindings`. Inside a nested binding context, this parameter will be set to the current data item (e.g., inside a `with`: `person binding,viewModel` will be set to `person`).

For example, you might have been controlling an element's visibility using the visible binding, but now you want to go a step further and animate the transition. You want elements to slide into and out of existence according to the value of an observable. You can do this by writing a custom binding that calls jQuery's `slideUp/slideDown` functions:

```
ko.bindingHandlers.slideVisible = {
  update: function(element, valueAccessor, allBindingsAccessor) {
    // First get the latest data that we're bound to
    var value = valueAccessor(), allBindings = allBindingsAccessor();

    // Next, whether or not the supplied model property is observable, get its current value
    var valueUnwrapped = ko.utils.unwrapObservable(value);

    // Grab some more data from another binding property
    var duration = allBindings.slideDuration || 400; // 400ms is default duration unless otherwise
specified

    // Now manipulate the DOM element
    if (valueUnwrapped == true)
      $(element).slideDown(duration); // Make the element visible
    else
      $(element).slideUp(duration); // Make the element invisible
  }
};
```

Now you can use this binding as follows:

```
<div data-bind="slideVisible: giftWrap, slideDuration:600">You have selected the option</div>
<label><input type="checkbox" data-bind="checked: giftWrap" /> Gift wrap</label>

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

Of course, this is a lot of code at first glance, but once you've created your custom bindings they can very easily be reused in many places.

The “init” callback

Knockout will call your init function once for each DOM element that you use the binding on. There are two main uses for init:

- To set any initial state for the DOM element
- To register any event handlers so that, for example, when the user clicks on or modifies the DOM element, you can change the state of the associated observable

KO will pass exactly the same set of parameters that it passes to the update callback.

Continuing the previous example, you might want `slideVisible` to set the element to be instantly visible or invisible when the page first appears (without any animated slide), so that the animation only runs when the user changes the model state. You could do that as follows:

```
ko.bindingHandlers.slideVisible = {
  init: function(element, valueAccessor) {
    var value = ko.utils.unwrapObservable(valueAccessor()); // Get the current value of the current
    property we're bound to
    $(element).toggle(value); // jQuery will hide/show the element depending on whether "value" or
    true or false
  },
  update: function(element, valueAccessor, allBindingsAccessor) {
    // Leave as before
  }
};
```

This means that if `giftWrap` was defined with the initial state `false` (i.e., `giftWrap: ko.observable(false)`) then the associated DIV would initially be hidden, and then would slide into view when the user later checks the box.

Modifying observables after DOM events

You’ve already seen how to use `update` so that, when an observable changes, you can update an associated DOM element. But what about events in the other direction? When the user performs some action on a DOM element, you might want to update an associated observable.

You can use the `init` callback as a place to register an event handler that will cause changes to the associated observable. For example,

```
ko.bindingHandlers.hasFocus = {
```

```
init: function(element, valueAccessor) {
    $(element).focus(function() {
        var value = valueAccessor();
        value(true);
    });
    $(element).blur(function() {
        var value = valueAccessor();
        value(false);
    });
},
update: function(element, valueAccessor) {
    var value = valueAccessor();
    if (ko.utils.unwrapObservable(value))
        element.focus();
    else
        element.blur();
}
};
```

Now you can both read and write the “focusedness” of an element by binding it to an observable:

```
<p>Name: <input data-bind="hasFocus: editingName" /></p>
```

Note: Supporting virtual elements

If you want a custom binding to be usable with Knockout’s *virtual elements* syntax, e.g.:

```
<!-- ko mybinding: somedata --> ... <!-- /ko -->
```

... then see the documentation for virtual elements.

2. Creating custom bindings that control descendant bindings

Note: This is an advanced technique, typically used only when creating libraries of reusable bindings. It’s not something you’ll normally need to do when building applications with Knockout.

By default, bindings only affect the element to which they are applied. But what if you want to affect all descendant elements too? This is possible. Your binding can tell Knockout *not* to bind descendants at all, and then your custom binding can do whatever it likes to bind them in a different way.

To do this, simply return { **controlsDescendantBindings**: true } from your binding’s init function.

Example: Controlling whether or not descendant bindings are applied

For a very simple example, here's a custom binding called `allowBindings` that allows descendant bindings to be applied only if its value is true. If the value is false, then `allowBindings` tells Knockout that it is responsible for descendant bindings so they won't be bound as usual.

```
ko.bindingHandlers.allowBindings = {
  init: function(elem, valueAccessor) {
    // Let bindings proceed as normal *only if* my value is false
    var shouldAllowBindings = ko.utils.unwrapObservable(valueAccessor());
    return { controlsDescendantBindings: !shouldAllowBindings };
  }
};
```

To see this take effect, here's a sample usage:

```
<div data-bind="allowBindings: true">
  <!-- This will display Replacement, because bindings are applied -->
  <div data-bind="text: 'Replacement'">Original</div>
</div>
```

Example: Supplying additional values to descendant bindings

Normally, bindings that use `controlsDescendantBindings` will also call `ko.applyBindingsToDescendants(someBindingContext, element)` to apply the descendant bindings against some modified binding context. For example, you could have a binding called `withProperties` that attaches some extra properties to the binding context that will then be available to all descendant bindings:

```
ko.bindingHandlers.withProperties = {
  init: function(element, valueAccessor, allBindingsAccessor, viewModel, bindingContext) {
    // Make a modified binding context, with a extra properties, and apply it to descendant elements
    var newProperties = valueAccessor(), innerBindingContext = bindingContext.extend(newProperties);
    ko.applyBindingsToDescendants(innerBindingContext, element);

    // Also tell KO *not* to bind the descendants itself, otherwise they will be bound twice
    return { controlsDescendantBindings: true };
  }
};
```

As you can see, binding contexts have an **extend** function that produces a clone with extra properties. This doesn't affect the original binding context, so there is no danger of affecting sibling-level elements - it will only affect descendants.

Here's an example of using the above custom binding:

```
<div data-bind="withProperties: { emotion: 'happy' }">
  Today I feel <span data-bind="text: emotion"></span>. <!-- Displays: happy -->
</div>

<div data-bind="withProperties: { emotion: 'whimsical' }">
  Today I feel <span data-bind="text: emotion"></span>. <!-- Displays: whimsical -->
</div>
```

Example: Adding extra levels in the binding context hierarchy

Bindings such as `with` and `foreach` create extra levels in the binding context hierarchy. This means that their descendants can access data at outer levels by using **`$parent`**, **`$parents`**, **`$root`**, or **`$parentContext`**.

If you want to do this in custom bindings, then instead of using `bindingContext.extend()`, use `bindingContext.createChildContext(someData)`. This returns a new binding context whose `viewModel` is `someData` and whose `$parentContext` is `bindingContext`. If you want, you can then extend the child context with extra properties using `ko.utils.extend`. For example,

```
ko.bindingHandlers.withProperties = {
  init: function(element, valueAccessor, allBindingsAccessor, viewModel, bindingContext) {
    // Make a modified binding context, with a extra properties, and apply it to descendant elements
    var newProperties = valueAccessor(),
        childBindingContext = bindingContext.createChildContext(viewModel);
    ko.utils.extend(childBindingContext, newProperties);
    ko.applyBindingsToDescendants(childBindingContext, element);

    // Also tell KO *not* to bind the descendants itself, otherwise they will be bound twice
    return { controlsDescendantBindings: true };
  }
};
```

This updated `withProperties` binding could now be used in a nested way, with each level of nesting able to access the parent level via `$parentContext`:

```
<div data-bind="withProperties: { displayMode: 'twoColumn' }">
  The outer display mode is <span data-bind="text: displayMode"></span>.
  <div data-bind="withProperties: { displayMode: 'doubleWidth' }">
    The inner display mode is <span data-bind="text: displayMode"></span>, but I haven't forgotten
    that the outer display mode is <span data-bind="text: $parentContext.displayMode"></span>.
  </div>
</div>
```

By modifying binding contexts and controlling descendant bindings, you have a powerful and advanced tool to create custom binding mechanisms of your own.

3. Creating custom bindings that support virtual elements

Note: This is an advanced technique, typically used only when creating libraries of reusable bindings. It's not something you'll normally need to do when building applications with Knockout.

Knockout's *control flow bindings* (e.g., `if` and `foreach`) can be applied not only to regular DOM elements, but also to "virtual" DOM elements defined by a special **comment-based syntax**. For example:

```
<ul>
  <li class="heading">My heading</li>
  <!-- ko foreach: items -->
    <li data-bind="text: $data"></li>
  <!-- /ko -->
</ul>
```

Custom bindings can work with virtual elements too, but to enable this, you must explicitly tell Knockout that your binding understands virtual elements, by using the **`ko.virtualElements.allowedBindings`** API.

Example

To get started, here's a custom binding that randomises the order of DOM nodes:

```
ko.bindingHandlers.randomOrder = {
  init: function(elem, valueAccessor) {
    // Pull out each of the child elements into an array
    var childElems = [];
    while(elem.firstChild)
      childElems.push(elem.removeChild(elem.firstChild));

    // Put them back in a random order
```

```
while(childElems.length) {
    var randomIndex = Math.floor(Math.random() * childElems.length),
        chosenChild = childElems.splice(randomIndex, 1);
    elem.appendChild(chosenChild[0]);
}
}
};
```

This works nicely with regular DOM elements. The following elements will be shuffled into a random order:

```
<div data-bind="randomOrder: true">
  <div>First</div>
  <div>Second</div>
  <div>Third</div>
</div>
```

However, it does *not* work with virtual elements. If you try the following:

```
<!-- ko randomOrder: true -->
  <div>First</div>
  <div>Second</div>
  <div>Third</div>
<!-- /ko -->
```

... then you'll get the error The binding 'randomOrder' cannot be used with virtual elements. Let's fix this.

To make randomOrder usable with virtual elements, start by telling Knockout to allow it. Add the following:

```
ko.virtualElements.allowedBindings.randomOrder = true;
```

Now there won't be an error. However, it still won't work properly, because our randomOrder binding is coded using normal DOM API calls (firstChild, appendChild, etc.) which don't understand virtual elements. This is the reason why KO requires you to explicitly opt in to virtual element support: unless your custom binding is coded using virtual element APIs, it's not going to work properly!

Let's update the code for randomOrder, this time using KO's virtual element APIs:

```
ko.bindingHandlers.randomOrder = {
  init: function(elem, valueAccessor) {
    // Build an array of child elements
    var child = ko.virtualElements.firstChild(elem),
        childElems = [];
    while (child) {
      childElems.push(child);
    }
  }
};
```



```
    child = ko.virtualElements.nextSibling(child);
  }

  // Remove them all, then put them back in a random order
  ko.virtualElements.emptyNode(elem);
  while(childElems.length) {
    var randomIndex = Math.floor(Math.random() * childElems.length),
        chosenChild = childElems.splice(randomIndex, 1);
    ko.virtualElements.prepend(elem, chosenChild[0]);
  }
}
};
```

Notice how, instead of using APIs like `domElement.firstChild`, we're now

using `ko.virtualElements.firstChild(domOrVirtualElement)`. The `randomOrder` binding will now correctly work with virtual elements, e.g., `<!-- ko randomOrder: true -->...<!-- /ko -->`.

Also, `randomOrder` will still work with regular DOM elements, because all of the `ko.virtualElements` APIs are backwardly compatible with regular DOM elements.

Virtual Element APIs

Knockout provides the following functions for working with virtual elements.

- `ko.virtualElements.allowedBindings`

An object whose keys determine which bindings are usable with virtual elements. Set **`ko.virtualElements.allowedBindings.mySuperBinding = true`** to allow `mySuperBinding` to be used with virtual elements.

`ko.virtualElements.emptyNode(containerElem)`: Removes all child nodes from the real or virtual element `containerElem` (cleaning away any data associated with them to avoid memory leaks).

- `ko.virtualElements.firstChild(containerElem)`

Returns the first child of the real or virtual element `containerElem`, or null if there are no children.

- `ko.virtualElements.insertAfter(containerElem, nodeToInsert, insertAfter)`

Inserts `nodeToInsert` as a child of the real or virtual element `containerElem`, positioned immediately after `insertAfter` (where `insertAfter` must be a child of `containerElem`).

- `ko.virtualElements.nextSibling(node)`

Returns the sibling node that follows `node` in its real or virtual parent element, or null if there is no following sibling.
- `ko.virtualElements.prepend(containerElem, nodeToPrepend)`

Inserts `nodeToPrepend` as the first child of the real or virtual element `containerElem`.
- `ko.virtualElements.setDomNodeChildren(containerElem, arrayOfNodes)`

Removes all child nodes from the real or virtual element `containerElem` (in the process, cleaning away any data associated with them to avoid memory leaks), and then inserts all of the nodes from `arrayOfNodes` as its new children.

Notice that this is *not* intended to be a complete replacement to the full set of regular DOM APIs. Knockout provides only a minimal set of virtual element APIs to make it possible to perform the kinds of transformations needed when implementing control flow bindings.