**Working with Lists and Collections**

Very often, you'll want to generate repeating blocks of UI elements, especially when displaying lists where the user can add and remove elements. Knockout lets you do that easily, using *observable arrays* and the **foreach** binding.

**Getting started**

In the next few minutes you'll build a dynamic UI for reserving seats and meals — this could be one step in an airline booking process. In the bottom-right pane, you've already got:

* **SeatReservation**, a simple JavaScript class constructor that stores a passenger name with a meal selection
* **ReservationsViewModel**, a viewmodel class that holds:
  + **availableMeals**, a JavaScript object providing meal data
  + **seats**, an array holding an initial collection of**SeatReservation** instances. Note that it's a**ko.observableArray** - not surprisingly, that's the*observable* equivalent of a regular array, which means it can automatically trigger UI updates whenever items are added or removed.

The view (top-right pane) doesn't yet display the reservation data, so let's fix that. Update the **<tbody>** element to use the **foreach** binding, so that it will render a copy of its child elements for each entry in the **seats** array:

<**tbody** data-bind="foreach: seats"></**tbody**>

... and then populate that **<tbody>** element with some markup to say that you want a table row (**<tr>**) for each entry:

<**tbody** data-bind="foreach: seats"> <**tr**> <**td** data-bind="text: name"></**td**> <**td** data-bind="text: meal().mealName"></**td**> <**td** data-bind="text: meal().price"></**td**> </**tr**> </**tbody**>

Notice that, because the **meal** property is an *observable*, it's important to invoke **meal()** as a function (to obtain its current value) before attempting to read sub-properties. In other words, write **meal().price**, *not***meal.price**.

The result? If you run the application now, you should see a simple table of seat reservation information.

**foreach** is part of a family of *control flow bindings* that includes [foreach](http://knockoutjs.com/documentation/foreach-binding.html),[if](http://knockoutjs.com/documentation/if-binding.html), [ifnot](http://knockoutjs.com/documentation/ifnot-binding.html), and [with](http://knockoutjs.com/documentation/with-binding.html). These make it possible to construct any kind of iterative, conditional, or nested UI based on your dynamic viewmodel.

Step 2

## Adding items

Following the MVVM pattern makes it very simple to work with changeable object graphs such as arrays and hierarchies. You update the underlying data, and the UI automatically updates in sync.

### Adding seat reservations

Add a button to your view, using the **click** binding to associate its clicks with a function on your viewmodel:

<**button** data-bind="click: addSeat">Reserve another seat</**button**>

... then implement that **addSeat** function, making it push an extra entry into the **seats** array:

**function** **ReservationsViewModel**() { // ... leave all the rest unchanged ... // Operations self.addSeat = **function**() { self.seats.push(**new** SeatReservation("", self.availableMeals[0])); } }

Try it - now when you click "Reserve another seat", the UI updates to match. This is because **seats** is an observable array, so adding or removing items will trigger UI updates automatically.

Note that adding a row does not involve regenerating the entire UI. For efficiency, Knockout tracks what has changed in your viewmodel, and performs a minimal set of DOM updates to match. This means you can scale up to very large or sophisticated UIs, and it can remain snappy and responsive even on low-end mobile devices.

Step 3

**Making the data editable**

You can use bindings within **foreach** blocks just the same as anywhere else, so it's pretty easy to upgrade what we've got into a full data editor. Update your view:

<**tbody** data-bind="foreach: seats"> <**tr**> <**td**><**input** data-bind="value: name" /></**td**> <**td**><**select** data-bind="options: $root.availableMeals, value: meal, optionsText: 'mealName'"></**select**></**td**> <**td** data-bind="text: meal().price"></**td**> </**tr**> </**tbody**>

This code uses two new bindings, **options** and **optionsText**, which together control both the set of available items in a dropdown list, and which object property (in this case, **mealName**) is used to represent each item on screen.

Try it - you can now select from the available meals, and doing so causes the corresponding row (only) to be refreshed to display that meal's price.

**Formatting prices**

We've got a nice object-oriented representation of our data, so we can trivially add extra properties and functions anywhere in the object graph. Let's give the **SeatReservation** class the ability to format its own price data using some custom logic.

Since the formatted price will be computed based on the selected meal, we can represent it using **ko.computed** (so it will update automatically whenever the meal selection changes):

**function** **SeatReservation**(name, initialMeal) { **var** self = **this**; self.name = name; self.meal = ko.observable(initialMeal); self.formattedPrice = ko.computed(**function**() { **var** price = self.meal().price; **return** price ? "$" + price.toFixed(2) : "None"; }); }

Now you can update the view to make use of the **formattedPrice** on each **SeatReservation** instance:

<**tr**> <**td**><**input** data-bind="value: name" /></**td**> <**td**><**select** data-bind="options: $root.availableMeals, value: meal, optionsText: 'mealName'"></**select**></**td**> <**td** data-bind="text: formattedPrice"></**td**> </**tr**>

Try it out.

Step 4

**Removing items and showing a total surcharge**

Since you can add items, you should be able to remove them too, right? As you'd expect, this is merely a matter of updating the underlying data.

Update your view so that it displays a "remove" link next to each item:

<**tr**> <**td**><**input** data-bind="value: name" /></**td**> <**td**><**select** data-bind="options: $root.availableMeals, value: meal, optionsText: 'mealName'"></**select**></**td**> <**td** data-bind="text: formattedPrice"></**td**> <**td**><**a** href="#" data-bind="click: $root.removeSeat">Remove</**a**></**td**> </**tr**>

Note that the **$root.** prefix causes Knockout to look for a **removeSeat**handler on your top-level viewmodel instead of on the **SeatReservation**instance being bound --- that's a more convenient place to put**removeSeat** in this example. So, add a corresponding **removeSeat**function to your root viewmodel class, **ReservationsViewModel**:

**function** **ReservationsViewModel**() { // ... leave the rest unchanged ... // Operations self.addSeat = **function**() { /\* ... leave unchanged ... \*/ } self.removeSeat = **function**(seat) { self.seats.remove(seat) } }

Now if you run the application, you'll be able to remove items as well as add them.

**Displaying a total surcharge**

It would be nice to let the customer know what they will be paying, right? Not surprisingly, we can define the total as a *computed property*, and let the framework take care of knowing when to recalculate and refresh the display.

Add the following **ko.computed** property inside**ReservationsViewModel**:

self.totalSurcharge = ko.computed(**function**() { **var** total = 0; **for** (**var** i = 0; i < self.seats().length; i++) total += self.seats()[i].meal().price; **return** total; });

Again, notice that since **seats** and **meal** are both observables, we're invoking them as functions to read their current values (e.g.,**self.seats().length**).

Display the total surcharge by adding the following **<h3>** element to the bottom of your view:

<**h3** data-bind="visible: totalSurcharge() > 0"> Total surcharge: $<**span** data-bind="text: totalSurcharge().toFixed(2)"></**span**> </**h3**>

This snippet demonstrates two new points:

* The **visible** binding makes an element visible or invisible as your data changes (internally, it modifies the element's CSS **display**style). In this case, we choose to show the "total surcharge" information only if it's greater than zero.
* You can use **arbitrary JavaScript expressions** inside declarative bindings. Here, we used **totalSurcharge() > 0** and**totalSurcharge().toFixed(2)**. Internally, this actually defines a computed property to represent the output from that expression. It's just a very lightweight and convenient syntactical alternative.

Now if you run the code, you'll see "total surcharge" appear and disappear as appropriate, and thanks to dependency tracking, it knows when to recalculate its own value — you *don't* need to put any code in your "add" or "remove" functions to force dependencies to update manually.

Step 5

## Final niceties

Having followed the MVVM pattern and got an object-oriented representation of the UI's data and behaviors, you're in a great position to sprinkle on extra behaviors in a very natural and convenient way.

For example, if you're asked to display the total number of seats being reserved, you can implement that in just a single place, and you don't have to write any extra code to make the seat count update when you add or remove items. Just update the **<h2>** at the top of your view:

<**h2**>Your seat reservations (<**span** data-bind="text: seats().length"></**span**>)</**h2**>

Trivial.

Similarly, if you're asked to put a limit on the number of seats you can reserve, say, you can make the UI represent that by using the **enable**binding:

<**button** data-bind="click: addSeat, enable: seats().length < 5">Reserve another seat</**button**>

The button becomes disabled when the seat limit is reached. You don't have to write any code to re-enable it when the user removes some seats (cluttering up your "remove" logic), because the expression will automatically be re-evaluated by Knockout when the associated data changes.

If you'd like to learn ways of saving the updated data back to the server, see the Loading and Saving Data tutorial.