open source
community experience distilled

KnockoutJS Starter

Learn how to knock out your next app in no time with KnockoutJS

Eric M. Barnard

[PACKT] PUBLISHING

KnockoutJS Starter

Learn how to knock out your next app in no time with KnockoutJS

Eric M. Barnard

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

KnockoutJS Starter

First published: November 2012

Production Reference: 1161112

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-114-1

www.packtpub.com

Credits

Author
Eric M. Barnard

Reviewer
Roy Jacobs

Acquisition Editor
Mary Nadar

Commissioning Editor
Yogesh Dalvi

Technical Editor
Vrinda Amberkar

Project Coordinator
Amigya Khurana

Proofreader
Mario Cecere

Graphics
Valentina D'Silva
Aditi Gajjar

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

Cover Image
Conidon Miranda

About the author

Eric Barnard is a Software Engineer in Champaign-Urbana, Illinois. He truly feels that a great day starts with a fresh pot of coffee and a blank JavaScript file on his computer screen.

Eric grew up on a farm in central Indiana, where he attended Purdue University. After graduating from Purdue, he sharpened his web development and startup skills as a Fellow in the Governor Robert Orr Fellowship in Indianapolis. At the time of this writing, Eric has recently got married and spends his free time attempting to keep his wife sane. He is the author of the Knockout Validation plugin and "KoGrid" a JavaScript DataGrid completely built on top of Knockout. You can find his blog at `http://www.ericbarnard.com`.

About the reviewer

Roy Jacobs is a Software Architect in Utrecht, the Netherlands. Wrangling C# and JavaScript is just as interesting as moving an icon two pixels to the left to improve the user experience.

Roy received his Bachelor's in Computer Science from the Fontys Polytechnic in Eindhoven and his Master's in Human-technology Interaction from the Technical University of Eindhoven. Apart from the technical stuff he dabbled in directing and visual effects and enjoys spending time with his girlfriend and their hamster.

He is the author of the Knockout Mapping plugin and his blog can be found at `http://www.royjacobs.org`.

Table of Contents

KnockoutJS Starter **1**

So, what is Knockout? **3**

Knockout – A JavaScript library 3

Knockout – A Model-View-ViewModel (MVVM) library 3

Installation **6**

Step 1 – What do I need? 6

Step 2 – Create a starter site 6

Step 3 – Download Knockout 8

Step 4 – Create our main application JavaScript file 8

Step 5 – Run the application 9

Summary 11

Quick start **12**

Business overview 12

Step 1 – Defining a namespace 12

Step 2 – Creating our Model 12

Step 3 – Creating a View for our Model 14

Step 4 – Creating a ViewModel to manage our Models 15

Step 5 – Working with Observable Arrays 15

Step 6 – Adding and removing Models from an Observable Array 16

Step 7 – Editing properties of a Model 18

Step 8 – Setting up a Master-Details view 19

Step 9 – Applying bindings 21

Summary 22

Top features you need to know about **23**
Subscribables 23
Observables 24
Observable Arrays 25
Computed Observables 27
Utilities 29
Data-bind statements 30
Applying bindings 31
Binding handlers 31
Summary 34
People and places you should get to know **35**
Official sites 35
Articles and tutorials 35
Community 36
Blogs 36
Twitter 36

KnockoutJS Starter

Welcome to *KnockoutJS Starter*. This book focuses on giving the reader a firm understanding of the core concepts of Knockout, such as MVVM and data binding, and works through real-life app development scenarios. All core components of Knockout's amazing library are covered in detail, and strategies are outlined for getting the best use of time when developing with Knockout.

This book contains the following sections:

- *So, what is Knockout?*– In this section you will get to know what Knockout does and how you might start to benefit from its functionality.
- *Installation* – In five easy steps, you will see how you can be ready to start developing with Knockout.
- *Quick start* – In this section you will learn to use Knockout to build an often needed business tool – an inventory management app.
- *Top features you need to know about* – This section illustrates Knockout's numerous extension points and plethora of utilities to help smooth the process of building your app.
- *People and places you should get to know* – Knockout's core development team is continually adding features and resources to help developers. You will learn, in this section, how to access Knockout's community and stay up to date in the future.

So, what is Knockout?

In this section you will get to know what Knockout does and how you might start to benefit from its functionality.

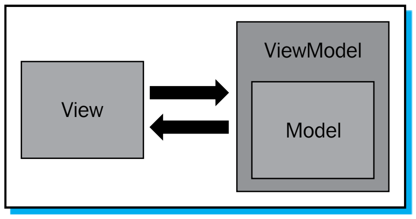
Knockout – A JavaScript library

Knockout, at its core, is a simple JavaScript file that can be included in a website or a web application to add JavaScript functionality, and provides the ability to enhance a user's experience. By default, Knockout does nothing to your website or web application until you specifically write code to utilize it. It is important to understand the difference between Knockout and many other JavaScript "frameworks" or "libraries" as some frameworks actually change how a website or web application works when included.

Knockout – A Model-View-ViewModel (MVVM) library

One of the reasons that Knockout was created was for the specific goal of enabling **Model-View-ViewModel** (**MVVM**) style development for websites and web applications. MVVM is a style of development where different object classes are designed to separate user-interface logic from business functions for testability purposes.

We strive to write testable code for many reasons (which is a topic for an entire other book), but the biggest reasons are maintainability of the code base and improved quality assurance. As I've learned to write code in a testable way, I've seen my code bases become more idiomatic and easier to maintain. If anything, building JavaScript applications with the MVVM pattern has allowed me to deliver more reliable applications in a shorter time period compared to when I was simply trying to sprinkle my HTML pages with DOM event handlers and unorganized pieces of logic.



The preceding diagram illustrates the common components and communication flow of an MVVM architecture. We can see that Models, Views, and ViewModels are the building blocks that we have to understand in order to derive benefit from the MVVM pattern. The first principle of a MVVM-style application is defining the business Models. A **Model** is an object that usually most directly represents a real-world object in the business system you are working in. It contains properties and functions that have business-like names and reactions. If you were to make a Model that represented an automobile, it would have properties such as:

- `MaxSpeed` (Number)
- `TireSize` (Number)
- `ManufacturerName` (String)

It would also have functions such as:

- `Honk`
- `DriveForward`

The second principle of a MVVM-style application is the **View**. A View is the HTML markup that describes the layout, elements (buttons, textboxes), colors, and other visual pieces of a portion of the user interface. It has no logic or code embedded in it, and it is completely declarative (all needed parts of the view are described purely in the HTML markup).

The third part of a MVVM-style application is the **ViewModel**. The ViewModel provides the connection between the View and the Model. If you were making a ViewModel for a View that was designed to display automobiles, it might have properties such as:

- `AutomobileCollection` (Array)
- `SelectedAutomobile` (Object)

And it would have functions like:

- `AddAutomobile`
- `SortAutomobiles`

The ViewModel really allows you to keep business logic in your Model objects and create the logic needed to power the user interface inside itself. The term for this is "Separation of Concerns" and is incredibly useful with large (and small) web application architectures.

The final principle of MVVM is the concept of **Binding**. Binding is the idea of connecting the properties and events of user interface elements (such as HTML elements) to functions and properties of an object such as a ViewModel. An example of a binding would be the need to connect a **AddAutomobile** button in the user interface with the ViewModel's `AddAutomobile` function, or perhaps even connecting many user interface buttons to the single `AddAutomobile` function on the ViewModel.

As Views in an MVVM application are almost always declarative in nature, bindings are often declared in the View markup. Knockout is no different, and heavily utilizes HTML-compliant "data-bind" declarations on HTML elements to enable its binding system.

The beauty of a solid MVVM library such as Knockout is that you can focus on developing the business logic and critical functionality that your application or website needs rather than spending critical time writing code to attach/detach event handlers and manually update textboxes when data values change.

Installation

In five easy steps, you can be ready to start developing with Knockout!

Step 1 – What do I need?

At the very least you will need the following in order to keep up with the examples in this guide:

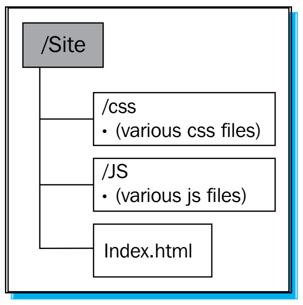
- A web browser
- A text editor
- Roughly 2 megabytes (MB) of space on the computer of your choice
- A basic web server (explained as follows)

Knockout development can be performed on most operating systems as long as you can install and use the tools listed above.

For the purpose of this guide we will be using Google Chrome as my web browser. It is free and can be installed on both, Windows and Mac operating systems. You can find it at `http://www.google.com/chrome`. My text editor of choice will actually take care of both my need for a text editor and a webserver. I will be using Microsoft's WebMatrix development tool. It is free and works on Windows operating systems. It can be downloaded from `http://www.microsoft.com/web/webmatrix/`. I will be using the IIS Express as the basic web server. It can be downloaded from `http://www.microsoft.com/en-us/download/details.aspx?id=1038`. If you have a Mac or Linux operating system, there are many great text editors and web servers out there that you can download and install for free.

Step 2 – Create a starter site

Now that we have our tools, we need to create a **workbench** for our application. I use the term "workbench" as we will be creating and editing many different files as we work through this guide. Some will be used simply for learning, while others will be part of the final application. To start, I generally recommend creating a folder in a well-known spot (such as your computer Desktop) to contain all of the files and folders that we will be using. My folder will usually end up looking something like the following diagram:



The `JS` folder will house all of our JavaScript files, and the `CSS` folder will house any CSS files that we use in our Knockout app. We can create this site structure in a number of ways, but my favorite way is to just visit `http://html5boilerplate.com/` and download their starter site. It provides a site structure (or Boilerplate) that includes many of the things you wouldn't want to try and remember how to do (such as including a `robots.txt` file for your site).

Once you've created your `Site` folder, make sure to create a `Index.html` file. For the purpose of this guide, ours will start out looking like the following HTML:

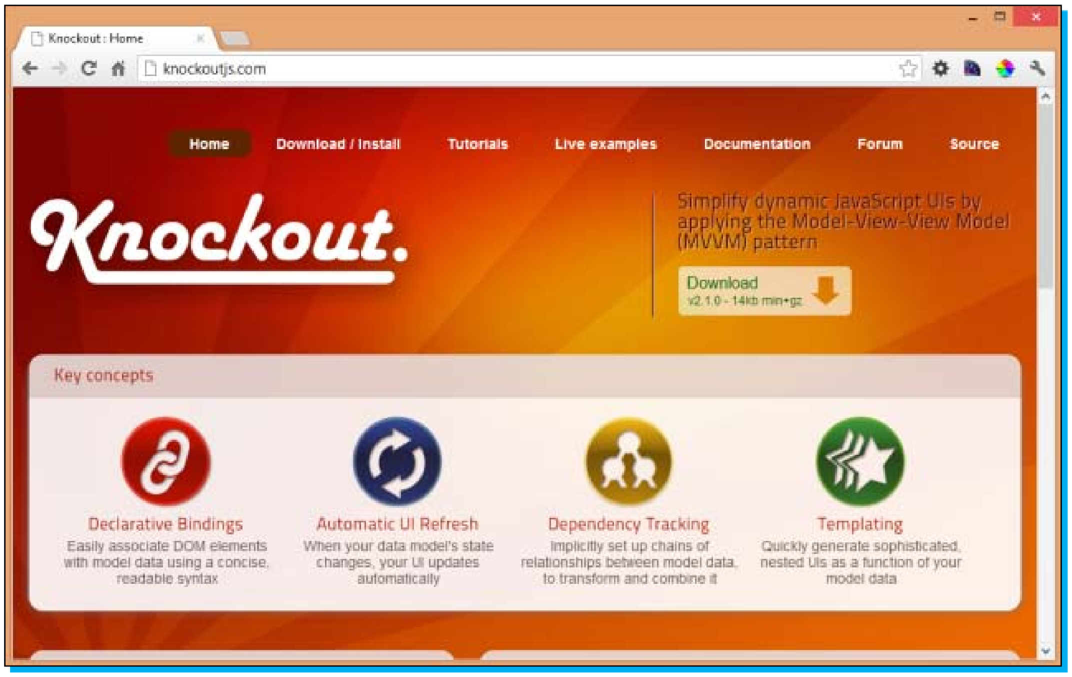
```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Knockout Starter Guide</title>
        <!-- CSS Here-->
    </head>
    <body>
        <div id="content">
            <p>Hello World!</p>
        </div>
        <!-- JavaScript Files Here -->
    </body>
</html>
```

This will be the starting point for developing our app. We have left placeholders for our JavaScript and CSS files, and we have created a "Content" area for the majority of our markup to be placed.

Step 3 – Download Knockout

Now the magic starts. In order to build a Knockout app, we obviously need to have the Knockout JavaScript library, and the best way to do that is to just download it from the Knockout website at `http://www.knockoutjs.com/`.


Knockout : Home
knockoutjs.com
Home
Download / Install
Tutorials
Live examples
Documentation
Forum
Source
Knockout.
Simplify dynamic JavaScript UIs by applying the Model-View-View Model (MVVM) pattern
Download v2.1.0 - 14kb min+gz
Key concepts
Declarative Bindings
Easily associate DOM elements with model data using a concise, readable syntax
Automatic UI Refresh
When your data model's state changes, your UI updates automatically
Dependency Tracking
Implicitly set up chains of relationships between model data, to transform and combine it
Templating
Quickly generate sophisticated, nested UIs as a function of your model data

Once you are on the Knockout website, just click on the **Download / Install** link at the top of the page, and follow the instructions on that page.

For the purposes of this guide, we will want to use the `Knockout-2.1.0.debug.js` JavaScript file, and we will want to put it in the `JS` folder that we created earlier.

Step 4 – Create our main application JavaScript file

Now we need to create a JavaScript file that will house all of the code for our application. We will create a `App.js` JavaScript file and place it in the `JS` folder of our site. To start, this file will just be an empty JavaScript as seen in the following sample:

```
// main application code here
```

Now that we have Knockout downloaded and our main application JavaScript file created, we need to include them in our `Index.html` page. When including JavaScript files, we simply add them to the HTML of our page using the traditional HTML `script` tag. The following HTML example illustrates how we use these at the bottom of the page to reference our JavaScript files.

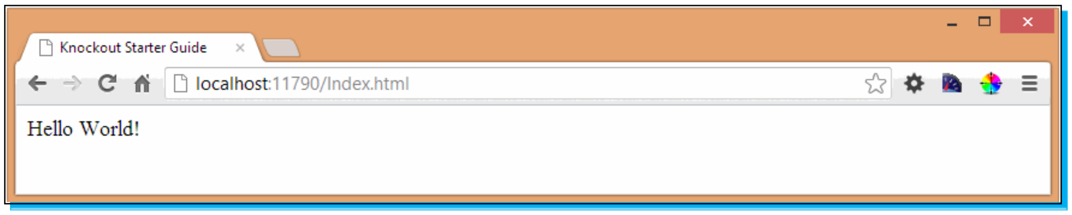
```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Knockout Starter Guide</title>
        <!-- CSS Here-->
    </head>
    <body>
        <div id="content">
            <p>Hello World!</p>
        </div>
        <!-- JavaScript Files Here-->
        <script type="text/javascript" src="/JS/knockout-2.1.0.debug.
js"></script>
        <script type="text/javascript" src="/JS/App.js"></script>
    </body>
</html>
```

We are referencing these scripts at the bottom of the page for a few reasons. The first reason is that it allows the page to appear to render faster to the end user because the entire visual markup is processed before the script references are, and thus will be displayed on the web browser's screen while the referenced JavaScript files are downloading and being processed into the browser's memory. Secondly, Knockout is a library that works with the browser's **Document Object Model** (**DOM**), and in order to do so, the browser needs to have created and rendered the content portions of the DOM before Knockout executes. Many other very good reasons exist as to why JavaScript files are referenced at the bottom of an HTML page, but that discussion is a bit out of scope for this guide. So, simply as a best practice we place our JavaScript files at the bottom of the page in order to give our users the best experience possible.

Step 5 – Run the application

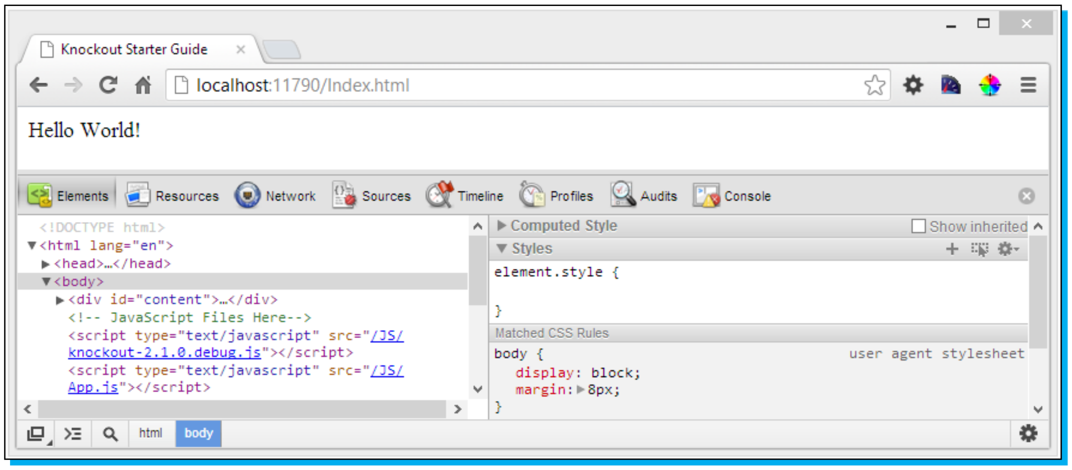
Now that we have our new site set up, let's run it and make sure everything works. As I am using WebMatrix, I can just select the `Index.html` file and click on **Run** (on my Windows computer). If you are using a different webserver, just start it with the root of the site being the site folder that we created in the *Step 2 – Create a starter site* section, and then point your web browser at the local URL your webserver is serving from.



Hello World!

You can see in the preceding screenshot that WebMatrix has selected to use port `11790` to serve my files from, and so it will likely depend on the machine that you are working with. The **Hello World!** that is currently displayed is simply the contents of our `Index.html` page. Great! We have a working JavaScript application!

In order to make sure that our JavaScript files actually loaded and did not create any errors, we need to fire up the browser's debugging tool and do a little investigation. In Chrome, I can open the debugger by clicking on the wrench icon (upper-right corner) and going to **Tools** | **Developer Tools**. The debugger will then take the bottom half of your browser window and look something like the following screenshot:



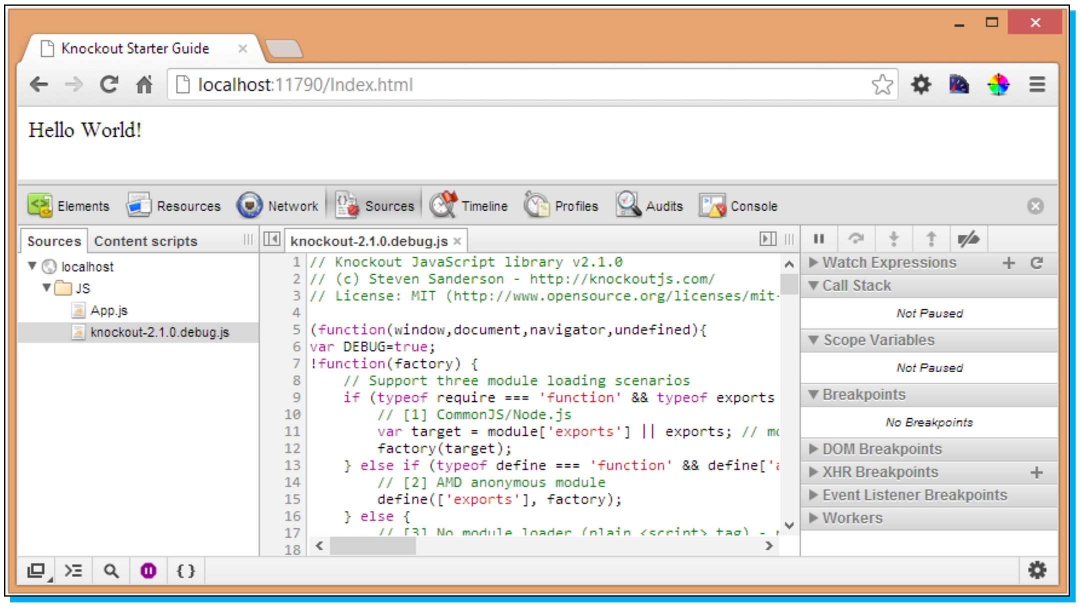
Hello World!

```
<!DOCTYPE html>
▼<html lang="en">
  ▶<head>…</head>
  ▼<body>
    ▶<div id="content">…</div>
     <!-- JavaScript Files Here-->
     <script type="text/javascript" src="/JS/
     knockout-2.1.0.debug.js"></script>
     <script type="text/javascript" src="/JS/
     App.js"></script>
```

```
element.style {

}
body {                          user agent stylesheet
   display: block;
   margin:▶8px;
}
```

If we click on the **Sources** tab of the debugger, we see what JavaScript files were loaded by the page, and the contents of those pages. In the following screenshot, we can see that both our `App.js` file and our freshly downloaded Knockout library file are, in fact, loaded correctly by the page. It is also important to note that there aren't any JavaScript errors on the page. We can see this by noting that the lower-right corner of the debugger is free of any error alerts. If we did have some errors on the page, you would see a red alert symbol with a number of errors inside of it.



Hello World!

```
// Knockout JavaScript library v2.1.0
// (c) Steven Sanderson - http://knockoutjs.com/
// License: MIT (http://www.opensource.org/licenses/mit-

(function(window,document,navigator,undefined){
var DEBUG=true;
!function(factory) {
    // Support three module loading scenarios
    if (typeof require === 'function' && typeof exports
        // [1] CommonJS/Node.js
        var target = module['exports'] || exports; // mo
        factory(target);
    } else if (typeof define === 'function' && define['a
        // [2] AMD anonymous module
        define(['exports'], factory);
    } else {
        // [3] No module loader (plain <script> tag) - 
```

If you are not using Chrome, you can simply search your browser's documentation online to determine how it handles and displays JavaScript files and errors in its own debugger (or developer tools).

Summary

You now have a functioning workbench for doing Knockout development. Congratulations! In this section we have learned how to:

- Find, download, and install the basic tools needed to perform Knockout (and general HTML/JavaScript) development
- Create a basic website from an empty file directory
- Download and install external JavaScript libraries, such as Knockout, into a website
- Create our JavaScript files and install them into a website
- Run a website locally on our machine for development and debugging purposes

Quick start

Sometimes the best way to learn a library is to use it to solve real-world problems. In this section we will learn to use Knockout to build an often needed business tool—an Inventory Management app.

Business overview

Before we can begin building our app, we need to have a firm understanding of the business needs that our application will solve. The app needs to be able to do the following things:

- Track the **Stock Keeping Unit** (**SKU**) number and **Description** of each product that the company sells
- Assign a price, cost, and on-hand quantity to each product
- Create new products when the company decides to sell them
- Remove old products when the company decides to stop selling them

Step 1 – Defining a namespace

Now, before we begin to actually develop our app, it is important that we figure out how we want to organize and separate our application code and logic from the rest of the browser's objects and native functions. You might wonder why we care to do this in such a small example application, but it is important to show JavaScript architecture best-practices whenever possible. Using namespaces, even in a small app, ensures that if your application grows after if it is first written (which many code-bases will), that it can remain easily maintainable and distinguishable from third-party code (such as JavaScript plugins that you might use down the road).

We will define our namespace in the `App.js` file that we created previously. You can see in the following code snippet, how we go about defining a namespace:

```
// Define the namespace
window.myApp = {};
```

Step 2 – Creating our Model

Our first step is to create a Model that will represent our company's `Product` object. We can do this by adding a file called `Product.js` to our `JS` folder. The contents of this file will look like the following:

```
(function (myApp) {

    // Product Constructor Function
    function Product() {
        var self = this;
```

```
            // "SKU" property
            self.sku = ko.observable("");

            // "Description" property
            self.description = ko.observable("");

            // "Price" property
            self.price = ko.observable(0.00);

            // "Cost" property
            self.cost = ko.observable(0.00);

            // "Quantity" property
            self.quantity = ko.observable(0);
        }

        // add to our namespace
        myApp.Product = Product;

    } (window.myApp));
```

In the preceding code snippet, we have defined a JavaScript function that will serve as the constructor function for a `Product` class. As you can see, we have wrapped the definition of our `Product` class in what is called an **Immediately-Invoked Function Expression** (**IIFE**). We use this pattern for the following reasons:

- It allows the concept of JavaScript scope to prevent us from polluting the global namespace (where objects such as `window` and `document` hangout). This way when we are debugging, it doesn't look like our `Product` constructor is a native function like `window`.
- It allows us to define `private` functions that aren't accessible by other code. If we had defined the `Product` constructor, but never added it as a property to our `myApp` namespace, then no code outside the IIFE wrapper would have access to our `Product` constructor. This is ideal for creating functions that handle complex logic, but in a way that prevents other objects from trying to utilize or overwrite that logic.

Inside the constructor function, each property is assigned to the `self` object. The `self` object is just a variable that points back to the newly created `Product` instance. The `this` keyword in JavaScript is very powerful, but sometimes confusing when it is used inside other functions, as it can represent a number of objects (such as the global namespace, the function-calling object). So to prevent confusion, we create a variable `self`, so that we can be sure it always points to the correct instance of our object.

Finally, each property is set to an instance of a Knockout **Observable**. Observable is simply Knockout's way of creating properties that can raise events when they change (which is a core concept of Knockout that we will go in depth into later). By passing an initial value into this function, we are returning a function object that wraps our initial value. We can get and set a property by simply invoking this wrapper function. The following code does a great job of showing how we can use our new `Product` constructor and its properties:

```
// Usage

// create an instance of the Product class
var productA = new myApp.Product();

// "set" the 'sku' property
productA.sku('12345')

// "get" the 'sku' property value
var skuNumber = productA.sku();
```

Step 3 – Creating a View for our Model

Now that we have defined our `Product` class, we need to create a way to let a user visually see the `Product` on the screen. We will do this by creating an HTML `view` for the product. We will start with a very simple HTML layout meant to show the values of a `Product` instance:

```
<div id="productView">
  <p>
    SKU: <span data-bind="text: sku"></span>
  </p>
  <p>
    Description: <span data-bind="text: description"></span>
  </p>
  <p>
    Cost: <span data-bind="text: cost"></span>
  </p>
  <p>
    Price: <span data-bind="text: price"></span>
  </p>
  <p>
    Quantity: <span data-bind="text: quantity"></span>
  </p>
</div>
```

Here we are simply just showing the values of our product using Knockout's `text` binding. The `text` binding sets the `innerText` property of an HTML element (usually a `span` element) to the string value of whatever object you are binding it to.

Step 4 – Creating a ViewModel to manage our Models

At this point, we know that we will need to create logic around adding and deleting products to our master product list. We also know that we will need some sort of an array to use as our master product list. So, at this point we need to introduce a class that will allow us to manage all of these functions, arrays, and objects that we will want to bind to the user-interface. The class we need (as described before) is the ViewModel!

As we are building this app a little bit at a time, we will only give our ViewModel one property—a `selectedProduct`. This property will represent the single product that we want to view the details of on the screen. We add file to our `JS` folder called `ProductsViewModel.js` and then add the following contents to that file:

```
// Products ViewModel
(function (myApp) {

    // constructor function
    function ProductsViewModel() {
        var self = this;

        // the product that we want to view/edit
        self.selectedProduct = ko.observable();
    }

    // add our ViewModel to the public namespace
    myApp.ProductsViewModel = ProductsViewModel;

} (window.myApp));
```

Step 5 – Working with Observable Arrays

As we mentioned in our business needs, our company sells many different products, and so we need to keep a list of these products. In JavaScript, the data structure used for maintaining lists of objects or values is an Array. Knockout goes ahead and takes this one step farther by providing us with an object called **Observable Array**. I will explain more on this object in the next section, but this type of Array raises events when it changes (similar to `Observable` that we described before). The events that the `ObservableArray` raises, allows Knockout to keep our user-interface up-to-date whenever `ObservableArray` changes.

Knockout's `ObservableArrays` have all the standard native functions that normal JavaScript arrays contain (push, pop, slice, splice), so they are very natural to work with if you have worked with JavaScript Arrays before.

In order to create a master list of products for our company's application, we need to give our ViewModel a new property—`productCollection`.

```
// the product that we want to view/edit
        self.selectedProduct = ko.observable();

    // the product collection
        self.productCollection = ko.observableArray([]);
```

Step 6 – Adding and removing Models from an Observable Array

Now that we have a master list of products for our company, we need to implement the logic for adding and removing products from the master list.

The logic for adding products will stay relatively simple. It would be easy to add some validation and other checks in this process, but I want to ensure it is as straightforward as possible right now.

```
// creates a new product and sets it up
// for editing
self.addNewProduct = function () {
  // create a new instance of a Product
  var p = new myApp.Product();

  // set the selected Product to our new instance
  self.selectedProduct(p);
};

// logic that is called whenever a user is done editing
// a product or done adding a product
self.doneEditingProduct = function () {
  // get a reference to our currently selected product
  var p = self.selectedProduct();

  // ignore if it is null
  if (!p) {
    return;
  }

  // check to see that the product
  // doesn't already exist in our list
  if (self.productCollection.indexOf(p) > -1) {
    return;
  }
```

```
    // add the product to the collection
    self.productCollection.push(p);

    // clear out the selected product
    self.selectedProduct(null);
  };
```

In this code example, we plan for the user to invoke the `addNewProduct` function, which will populate our `selectedProduct` property with a new product ready for editing. When the user is done editing the product, we plan for them to invoke the `doneEditingProduct` function, which will take care of checking for null values, and then add it to our `productCollection`.

The logic for removing a product is pretty simple. We simply check for null values and then pull it out of the collection, as shown in the following code:

```
  // logic that removes the selected product
  // from the collection
  self.removeProduct = function () {
    // get a reference to our currently selected product
    var p = self.selectedProduct();

    // ignore if it is null
    if (!p) {
      return;
    }

    // empty the selectedProduct
    self.selectedProduct(null);

    // simply remove the item from the collection
    return self.productCollection.remove(p);
  };
```

Finally we need to provide some buttons in the user-interface so that our users can actually invoke this logic. We will add some buttons, and then bind the click event of those buttons to their corresponding functions on the ViewModel as in the following example:

```
  <div id="content">
    <div id="productView" data-bind="with: selectedProduct">
      <p>
        SKU: <span data-bind="text: sku"></span>
      </p>
      <p>
        Description: <span data-bind="text: description"></span>
      </p>
      <p>
        Cost: <span data-bind="text: cost"></span>
      </p>
      <p>
        Price: <span data-bind="text: price"></span>
      </p>
      <p>
        Quantity: <span data-bind="text: quantity"></span>
      </p>
    </div>
    <div id="buttonContainer">
      <button type="button" data-bind="click: addNewProduct">Add</
  button>
      <button type="button" data-bind="click: removeProduct">Remove</
  button>
      <button type="button" data-bind="click: doneEditingProduct">Done</
  button>
    </div>
  </div>
```

Step 7 – Editing properties of a Model

At this point we have no way of actually editing the properties of each product in our master list. So we need to change our product view to utilize two-way bindings. Knockout's `value` binding allows this, but we can only use this on `input` elements. So let's change the HTML of our product view to be a little more like regular HTML forms:

```
<div id="productView">
  <form>
    <fieldset>
      <legend>Product Details</legend>
      <label>SKU:
        <input type="text" data-bind="value: sku"/>
      </label>
      <br/>
      <label>Description:
        <input type="text" data-bind="value: description"/>
      </label>
      <br/>
      <label>Cost:
        <input type="text" data-bind="value: cost"/>
      </label>
      <br/>
      <label>Price:
        <input type="text" data-bind="value: price"/>
```

```
          </label>
          <br/>
          <label>Quantity:
            <input type="text" data-bind="value: quantity"/>
          </label>
        </fieldset>
      </form>
    </div>
```

Our new form-based view now allows editing of selected product. I will mention at this point, that you will definitely want to utilize some form of input validation to ensure that `Cost` and `Price` are valid currency amounts, and that `Quantity` is a valid integer. However, that is a bit out of scope of this guide and there are several great libraries out on the web for providing that type of functionality.

Step 8 – Setting up a Master-Details view

Finally, we now have the logic and HTML pieces for creating a very nice user-interface for managing our company's inventory. Let's go ahead and put some finishing touches to our app to create a nice Master-Details experience for our user.

First, let's make sure that our product view is correctly bound to our selected product, and that the product view is only visible when we have a valid product instance selected. Knockout provides a "with" binding that allows us to do exactly that. We will discuss these bindings in more detail later, but the "with" binding allows us to both perform a null check on the `selectedProduct` observable and change the binding context of the html from the `ProductsViewModel` to the `selectedProduct` (so that we can directly reference those properties in our data-bind statements).

As the **Remove** and **Done** buttons only make sense to be visible when we have a product selected, we will add a `visible` binding to those buttons that checks whether the `selectedProduct` property has a value. We can also do the exact opposite with the **Add** button. One way of accomplishing this is shown in the following code:

```
<div id="buttonContainer">
  <button type="button"
    data-bind="click: addNewProduct,
         visible: (selectedProduct() ? false : true)">Add</button>
  <button type="button"
    data-bind="click: removeProduct,
         visible: (selectedProduct() ? true : false)">Remove</button>
  <button type="button"
    data-bind="click: doneEditingProduct,
         visible: (selectedProduct() ? true : false)">Done</button>
</div>
```

Finally, we need to provide our users with a way to view the master list of products that they are managing. Often we see data grids, tables, unordered lists, or other widgets designed to display lists of objects. Knockout is powerful enough, though, that we can actually just use raw HTML to provide a view into our `productCollection`.

We are going to implement a basic list-view using a select element. Knockout provides a `options` binding that allows us to bind an Observable Array to a select element. We will also set up a second Observable in our `ProductsViewModel` to hold our new list-view's selected item. In order to do that, we can simply use a `value` binding to bind the selected item of our select element to this new property of our `ProductsViewModel`. The following code shows the added property, `listViewSelectedItem`, along with a `subscription` that passes along any changes in the `listViewSelectedItem` property to our main `selectedProduct` property:

```
// the product that we want to view/edit
self.selectedProduct = ko.observable();

// the product collection
self.productCollection = ko.observableArray([]);

// product list view selected item
self.listViewSelectedItem = ko.observable(null);

// push any changes in the list view to our
// main selectedProduct
self.listViewSelectedItem.subscribe(function (product) {
  if (product) {
    self.selectedProduct(product);
  }
});
```

Our list view is then implemented in HTML with the following code:

```
<div id="productListView">
  <select id="productList" size="10"
      style="min-width:  120px;"
      data-bind="options: productCollection,
             value: listViewSelectedItem,
             optionsText: 'sku'">
  </select>
</div>
```

You can see in the preceding code that we can specify a `optionsText` binding that specifies which property of each element in our Observable Array we want to appear in the select element's options. Initially I've set this property to just be the `Sku` property of our `Product` class—but what if I wanted to see both the `Sku` property and the `Description` property? We can accomplish this with a Computed Observable. Again, we will go into deeper details about

Computed Observables, but for now we will simply add a Computed Observable that returns the `Sku` property and the `Description` property of the `Product` class:

```
// "Quantity" property
self.quantity = ko.observable(0);

// Computed Observables

// simply combines the Sku and Description properties
self.skuAndDescription = ko.computed(function () {
  var sku = self.sku() || "";
  var description = self.description() || "";

  return sku + ": " + description;
});
```

After adding the `skuAndDescription` computed property to our `Product` class, we can update our product list-view HTML. We just simply change the `optionsText` binding to now point to the `skuAndDescription` property instead of the `Sku` property.

Step 9 – Applying bindings

In order for our app to actually run, we need to kick-off the Knockout binding process. We need to make sure to wait and invoke this process after all of our scripts have been loaded, and after our ViewModel has been initialized. My suggested way is to set up this logic in the `App.js` file like the following:

```
// Define the namespace
window.myApp = {};

(function(myApp){

    // constructor functio for App
    function App(){

        // core logic to run when all
        // dependencies are loaded
        this.run = function(){

            // create an instance of our ViewModel
            var vm = new myApp.ProductsViewModel();

            // tell Knockout to process our bindings
            ko.applyBindings(vm);
        }
    }

    // make sure its public
    myApp.App = App;

}(window.myApp));
```

After we have created my initialization logic in my `App.js` file, we will then create an instance of the app, and call the `run` function at the very bottom of the page, as shown in the following code snippet:

```
    <script type="text/javascript">
      var app = new myApp.App();

      app.run();
    </script>
```

For the purpose of this tutorial, I am placing this call at the very bottom of the page. We do have other options, though, as to where we put this code. Another suggested option would be inside of jQuery `ready` callback (a popular function in the jQuery library).

Summary

After completing the previous steps, you have successfully created the Knockout JS app! Congratulations! In this section we covered the following Knockout concepts and objects:

- Observables
- Observable Arrays
- Computed Observables
- The `text`, `value`, `options`, `visible`, `click`, and `with` bindings
- The ViewModel architecture

We quickly went through many of the core concepts of Knockout, and if you have questions—that's OK—we will cover them in the following section.

Top features you need to to know about

One of the best parts of Knockout is its extensibility. Knockout has numerous extension points and contains a plethora of utilities for building your app. Many developers have built great Knockout sites without any other JavaScript libraries (even jQuery) than the core Knockout library.

Subscribables

When creating our Inventory Management app, it was easy to see that a Knockout Observable is one of the core objects that Knockout was built around. However, under the covers of Observables, Observable Arrays, and Computed Observables is **Subscribable**. A **Subscribable** is simply an object that has three methods and an array of **Subscriptions**. The three methods are:

- `subscribe`: This adds a Subscription callback for a certain topic to be invoked when subscriptions are "notified". The default topic is "changed".
- `notifySubscribers`: This calls all subscriptions for a certain topic and passes one argument to all the Subscription's callbacks.
- `extend`: This applies an Extender to the Subscribable object.

The following code snippet shows an example of how the first two methods allow Subscribables to perform basic Pub/Sub functionality:

```
var test = ko.observable();

// create a subscription for the "test-event"
test.subscribe(function (val) {
    console.log(val);
}, test, "test-event");

test.notifySubscribers("Hello World", "test-event");
```

One of the coolest abilities of a Knockout Subscribable is its ability to "mix-in" to any JavaScript object. The following code example shows a great way that you can leverage the Knockout `Subscribable` in some of your current code:

```
 // Dummy Subscribable
function PubSub(){

    // inherit Subscribable
    ko.subscribable.call(this);
}

// create an instance of our Subscribable
var pubsub = new PubSub();
```

```
// make a subscription
var subscription = pubsub.subscribe(function (val) {
    console.log(val);
}, pubsub, 'test-topic');

pubsub.notifySubscribers("hello world", "test-topic");
// console: "hello world"

// clean up things
subscription.dispose();
```

Whenever we call the `subscribe` function of a `Subscribable`, we are returned a `Subscription`. Often developers will ignore the `Subscription` that they are returned from these calls, but it is important to understand that a `Subscription` allows you to properly dispose of your `Subscription's` callback and also your app's other objects that are referenced by that callback. Just simply calling `dispose` on your `Subscription` takes care of this, and you can ensure that your app doesn't create memory leaks!

Now that you understand one of Knockout's absolute core building blocks, the `Subscribable`, we can learn how Knockout extends its usage into other building blocks.

Observables

The Knockout Observable is one of the first objects that leverages the Subscribable functionality, and is one of the most simple, yet powerful pieces of the Knockout library. The following shows the very basic idea of implementation of Observable:

```
 // Very Simple Knockout Observable Implementation

// ko.observable is actually a function factory
ko.observable = function (initialValue) {

    // private variable to hold the Observable's value
    var _latestValue = initialValue;

    // the actual "Observable" function
    function observable() {

        // one or more args, so it's a Write
        if (arguments.length > 0) {

            // set the private variable
            _latestValue = arguments[0];

            // tell any subscribers that things have changed
```

```
                observable["notifySubscribers"](_latestValue);

                return this; // Permits chained assignments
            }
            else { // no args, so it's a Read

                // just hand back the private variable's value
                return _latestValue;
            }
        }

        // inherit from Subscribable
        ko.subscribable.call(observable);

        // return the freshly created Observable function
        return observable;
    };
```

Again, this is very basic example implementation of the Knockout Observable (the actual implementation has much more robust logic, equality checking, and dependency detection). You can see from its implementation that the `ko.observable` function is essentially a function factory. When you call this factory method, it generates a function (simply called `observable`) that provides a basic "get/set" API. If you invoke the returned function without an argument, then it returns the `_initialValue`, but if you invoke the function with an argument, it sets the `_initialValue` to that argument and notifies all subscribers.

Observables are useful for just about anything you can imagine when building your application as they are **event-driven**. We can build an application architecture that only worries about reacting to events (such as a button click, or an input element's change event), rather than a procedural, single entry-point type of code base.

Observable Arrays

I briefly mentioned Observable Arrays earlier, and I promised that I would dig deeper into this concept. While the previous example implementation of the Observable is relatively simple, the Observable Array is even simpler, as shown in the following example implementation:

```
    // Very Simple Knockout Observable Array Implementation

    // function factory for observable arrays
    ko.observableArray = function (initialValues) {

        // make sure we have an array
        initialValues = initialValues || [];
        // create a Knockout Observable around our Array
        var result = ko.observable(initialValues);

        // add our Observable Array member functions
        // like "push", "pop", and so forth
        ko.utils.extend(result, ko.observableArray['fn']);

        // hand back the Observable we've created
        return result;
    };
```

As you can see, it is really just an Observable, except that the value of Observable is an Array. Functions have also been added to the Observable Array that match methods of a native Array. The Knockout authors have done this in order to allow us as developers to be able to work with an Observable Array in the same way we can work with normal Arrays.

A little known extensibility point is the `fn` property of the Observable Array constructing function. You can add your own functions to this special property, and they will be included on all Observable Arrays that your application uses. The following implementation shows how we can easily add a function that filters items in the array:

```
    ko.observableArray.fn['filter'] = function (filterFunc) {
        // get the array
        var underlyingArray = this();
        var result = [];

        for (var i = 0; i < underlyingArray.length; i++) {
            var value = underlyingArray[i];

            // execute filter logic
            if (filterFunc(value)) {
                result.push(value);
            }
        }
        return result;
    };

    var list = ko.observableArray([1, 2, 3]);

    // filter down the list to only odd numbers
    var odds = list.filter(function (item) {
        return (item % 2 === 1);
    });

    console.log(odds); // [1, 3]
```

The use of a `fn` property on constructor functions is a common pattern that you see in many libraries (including jQuery), and Knockout is no different. The `fn` property also exists on the Observable, so you can extend its functionality the same way.

Computed Observables

Computed Observables are arguably the most powerful feature of Knockout. You may have noticed that creating a ViewModel with nothing but Observables and Observable Arrays is a huge step up from plain JavaScript, but it would be really nice to have other properties on your ViewModel that simply depended upon other Observables and Observable Arrays and updated themselves when their dependencies change.

Enter the Computed Observable (or in older versions of Knockout, the `dependentObservable`). A Computed Observable looks to be very similar to a regular Observable; however instead of holding a value, it contains a function that is used to evaluate the return value of the Computed Observable. The evaluation function is only executed when the Observables that it depends upon change. The following example shows a very basic example of how this works.

```
var a = ko.observable(1);
var b = ko.observable(2);

var sum = ko.computed(function () {
    var total = a() + b();

    // let's log every time this runs
    console.log(total);
    return total;
});
// console: 3

a(2); // console: 4
b(3); // console: 5
b(3); // (nothing logged)
```

Computed Observables have an extremely interesting and ingenious implementation, which you can dig into on your own. However, there are a few important things to note.

First, Computed Observables build up a list of dependencies (or Subscriptions to be precise) when the evaluation function executes. If you want a certain Observable to be registered as a subscription to your Computed Observable, then you need to ensure that the Observable is always called in your evaluation function. The following example shows how you can set up a changing dependency set for a Computed Observable:

```
var dep1 = ko.observable(1);
var dep2 = ko.observable(2);
var skipDep1 = false;

var comp = ko.computed(function () {
    dep2(); // register dep2 as dependency

    if(!skipDep1) {
        dep1(); // register dep1 as dependency
    }

    console.log('evaluated');
});
// console: evaluated

dep1(99); // console: evaluated
skipDep1 = true;
dep2(98); // console: evaluated
dep1(97); // (nothing logged)
```

Generally, it is a bad idea to do something like the previous example. It is extremely hard to debug, and is not intuitive for other developers to realize what is going on. Instead, the suggested pattern is to retrieve the value of each of your dependencies at the beginning of the evaluation function, and then perform any conditional logic on private scoped variables after that.

Secondly, and this applies to Observables in general, a subscription callback will only be called if the Observable determines that the before and after values are different. Each Observable has a property called `equalityComparer` which determines if the before and after values of an Observable are in fact different. If you notice in the first example, the very last call to set the value of `b` did nothing. This is because `b` was already `3` and had no reason to notify its subscriptions. The default `equalityComparer` of each Observable is a function that only compares JavaScript primitives. So to make that simple, if a Computed Observable's dependencies are all objects, complex or not, the evaluator function will be re-executed every time one of its dependencies change. The following example provides an illustration of this important concept:

```
var depPrimitive = ko.observable(1);
var depObj = ko.observable({ val: 1 });

var comp = ko.computed(function () {
    // register dependencies
    var prim = depPrimitive();
    var obj = depObj();

    console.log("evaluated");
});
```

```
// console: evaluated


depPrimitive(1); // (nothing logged)
var previous = depObj();
depObj(previous); // console: evaluated
```

It is important to understand this because many novice Knockout developers hurt themselves by overusing Computed Observables and accidentally building long event chains that ultimately lead to slow-performing applications.

Lastly, Computed Observables can have both an evaluation function for "setting" and "getting" their values. This is incredibly useful, because it allows us to have private Observable fields on our ViewModel that are protected through publicly exposed "get/set" functions and we also get the power of a normal Observable. In the following code you can see an example of this useful concept:

```
var _val = ko.observable(1);

var vm = {
    val: ko.computed({
        read: function () {
            return _val();
        },
        write: function (newVal) {
            _val(newVal);
        }
    })
};

vm.val(2);
console.log(_val()); // console: 2

_val(3);
console.log(vm.val()); // console: 3
```

Utilities

Knockout is full of many useful utilities that you can use in your application development. You can find these by exploring the `ko.utils` namespace, but some of my favorite utilities are as follows:

- `extend`: This function "mashes" two objects together. Essentially all of the properties and functions of the second argument to this function call are added to the first argument.

- `unwrapObservable`: This function takes an object property and intelligently returns its value, figuring out whether it is a Knockout Observable or just a simple property. It is extremely useful if you have to write functions that might take an object, a primitive, or an Observable as an argument and you need it to figure that out properly at runtime.
- **All of the Array utilities**: Knockout has several Array-manipulation functions that allow you to do filtering, mapping, and removing items. I usually add these utilities to the special `ko.observableArray.fn` property when I first start a project.

The following shows some example usage of these utilities:

```
// extend usage
var a = { val: 1 },
    b = { val: 2 };

ko.utils.extend(a, b);

console.log(a.val); // console: 2

// unwrapObservable usage
var c = ko.observable(99),
    d = 98;

console.log(ko.utils.unwrapObservable(c)); // console: 99
console.log(ko.utils.unwrapObservable(d)); // console: 98

// array "map" utility function usage
var arr = [100, 101];

var mapped = ko.utils.arrayMap(arr, function (item) {
    return item + 50;
})

console.log(arr); // console: [ 150, 151 ]
```

Data-bind statements

So far, we've focused on quite a bit of the JavaScript components of the Knockout library. However, Knockout was designed to make binding JavaScript objects and HTML extremely easy. The API that is given to us to use is through the HTML5 compliant `data-bind` statement.

From the examples in our previous app, you may think that all the `data-bind` statements are simply matching an HTML element's attribute to a property of our ViewModel. In actuality, though, the `data-bind` syntax supports quite a bit more. We can actually implement small bits of JavaScript in these statements, and it will be evaluated correctly at runtime. The `data-bind` statement also allows declaring a comma-separated list of binding declarations. The following implementation shows some of the allowed syntaxes for `data-bind` statements on HTML elements:

```
<span data-bind="text: myText"></span>

<div data-bind="visible: isVisible, with: someProp"></div>

<input data-bind="value: someVal,
                  css: {
                    'error': !someVal.isValid(),
                    'success': someVal.isValid()
                  }"/>
```

Applying bindings

The `applyBindings` function is where all of the Knockout magic gets kicked off. Many examples show the `applyBindings` function being called with a ViewModel object being passed in as the only argument, but you can also specify a DOM node as the second argument. When passing a DOM node as the second argument, Knockout only binds your ViewModel to that node and its children.

Most basic applications are fine with only having one ViewModel, and just simply calling `applyBindings` with that single ViewModel. However, I've built several complex applications where we used multiple ViewModels for one page. It is important to consider the advantages of having a separate ViewModel in your application for, alerts, settings, and current user information. In some cases, you can also gain performance benefits by limiting the number of nodes that Knockout has to walk in order to finish its binding process. If you are only enhancing a small portion of your HTML page with Knockout, don't call `applyBindings` blindly across the entire DOM.

Binding handlers

I've mentioned Knockout's amazing extensibility points, and few are better than Knockout's **binding handler** objects. Every Knockout binding that you see in a `data-bind` statement is implemented as a binding handler, and Knockout allows us to define our own binding handlers so that we can implement, or override, our own custom functionality.

With MVVM-style application development, we have two types of bindings—**one-way bindings** and **two-way bindings**. One-way bindings are simply read-only bindings, and are for pushing a ViewModel's property updates through to the DOM. You can probably guess that two-way bindings are bindings that take that one step further and allow DOM changes to be pushed back through to a ViewModel's property. Knockout allows us to create both of these types of bindings. The following shows a very basic template for a binding handler:

```
ko.bindingHandlers['myHandler'] = {

    init: function(element, valueAccessor, allBindingsAccessor,
viewModel, bindingContext){

    },

    update: function(element, valueAccessor, allBindingsAccessor,
viewModel, bindingContext){

    }
};
```

As we can see, we are provided two hooks for implementing our own logic—`init` and `update` functions that have the same signature. The arguments for those functions are as follows:

- `element`: The HTML element that the `data-bind` statement is declared on.
- `valueAccessor`: A function that returns the ViewModel value to which the binding is set. If the binding is set to an Observable property, then the Observable is returned (and we'll need to unwrap within our logic).
- `allBindingsAccessor`: Similar to the `valueAccessor`, but it returns an object that contains all the bindings and their respective bound values.
- `viewModel`: The ViewModel or root object that was passed into the `applyBindings` statement.
- `bindingContext`: A special object that has specific properties that represent the data context of the current binding. The binding context has a `$data` property that represents the currently bound context (which most often would be same as the `ViewModel` argument). It also has `$parent` and `$parents` properties that represent any data contexts that may be bound to elements higher in the DOM tree. We usually only use the parent properties if we have used the `with` binding in our application.

You might be wondering, which functions we should implement our custom logic in since they both look to do the same thing. The `init` function will be called only once when `applyBindings` function has been called. Knockout walks the DOM, finds `data-bind` statements, processes them, and calls the `init` method on each binding handler that is needed. The `update` function is then called immediately after the `init` function is called, and then also whenever the value it is bound to changes (if the value is a `Subscribable`).

My rule-of-thumb for binding handlers is that I register all of my event handlers (change, blur, focus) in the `init` function, then I perform my HTML manipulation in the `update` function.

The following code snippet shows a common one-way binding that I add to my projects. It simply reverses the `Boolean` logic of the `visible` binding so that I can handle situations where a ViewModel property makes more sense to be developed as `vm.isHidden();` instead of `vm.isVisible();`.

```
// invisible -> the inverse of 'visible'
ko.bindingHandlers['invisible'] = {
    update: function (element, valueAccessor) {
        var newValueAccessor = function () {
            // just return the opposite of the visible flag!
            return !ko.utils.unwrapObservable(valueAccessor());
        };
        return ko.bindingHandlers.visible.update(element, 
newValueAccessor);
    }
};
```

The following code snippet shows a two-way binding that I use for ensuring I have a valid number. It manually handles the updates from the UI and pushes those to the ViewModel, and then also updates the UI when the ViewModel changes.

```
// simple number parsing
function parseNumber(strVal){
    return parseInt(strVal, 10);
}

// very basic two-way binding handler
ko.bindingHandlers['number'] = {
    init: function (element, valueAccessor, allBindingsAccessor) {

        //handle the input changing
        ko.utils.registerEventHandler(element, "change", function () {
            var observable = valueAccessor();
            var number = parseNumber(element.value);

            if (number !== NaN) {
                observable(element.value);
            }
        });
    },
    update: function (element, valueAccessor) {
        var value = ko.utils.unwrapObservable(valueAccessor());
```

```
            var number = parseNumber(value);

            if (number !== NaN) {
                element.setAttribute("value", number);
            }
        }
    };
```

Summary

All in all you can see that Knockout is a very mature and thorough JavaScript library.
In this section you have learned:

- The core objects of Knockout and their usage
- Knockout utilities
- How to create and manage an app with multiple ViewModels
- Custom Binding Handlers

Knockout has a lot of goodies under the hood. I've included many snippets of Knockout's actual source code in this section so that you won't be afraid to look through the source code and learn more about how you can take advantage of its APIs.

People and places you should get to know

If you need help with KnockoutJS, here are some people and places which will prove useful.

Official sites

- **Homepage**: `http://www.knockoutjs.com/`
- **Manual and documentation**: `http://www.knockoutjs.com/documentation/introduction.html`
- **Source code**: `https://www.github.com/SteveSanderson/knockout`

Articles and tutorials

The top five Knockout resources are:

- `http://learn.knockoutjs.com/`: This is a great tutorial and playground that the author of Knockout (Steve Sanderson) has created to help folks get familiar with Knockout.
- `http://channel9.msdn.com/Events/MIX/MIX11/FRM08`: The original Channel 9 video that created much of the fame around Knockout. It is still very applicable to today's needs!
- `http://www.knockmeout.net/2011/08/simplifying-and-cleaning-up-views-in.html`: A great article about how to apply common sense and best practices to beginner level Knockout code.
- `http://www.pluralsight.com/training/Courses/TableOfContents/spa`: This is an online tutorial that costs just a few dollars, but is an amazing walkthrough of building real-life single-page applications in Knockout. It is also full of great tips on how to organize and maintain a large JavaScript codebase.
- `http://knockoutjs.com/documentation/introduction.html`: Yes, the Knockout documentation page. It is one of the best done documentation sites I've been able to use, and it does an incredible job of explaining every single API and function that Knockout exposes. I visit this page at least three times a week.

Community

- **Official mailing list**: `http://groups.google.com/group/knockoutjs`
- **GitHub**: `https://www.github.com/SteveSanderson/knockout`
- **Stack overflow tag**: `http://www.stackoverflow.com/questions/tagged/knockout.js`

Blogs

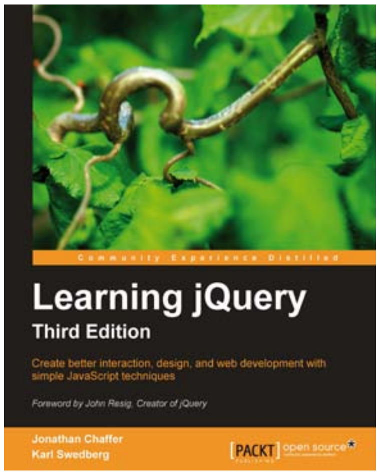
Blogs to watch:

- **Steve Sanderson**: `http://blog.stevensanderson.com/`
- **Ryan Niemeyer**: `http://www.knockmeout.net/`

Twitter

Knockout leaders on Twitter:

- Steve Sanderson: `@stevensanderson`
- John Papa: `@john_papa`
- Ryan Niemeyer: `@rpneimeyer`
- Me: `@ericmbarnard`
- For more open source information, follow Packt at `http://twitter.com/#!/packtopensource`


Community Experience Distilled
Learning jQuery
Third Edition
Create better interaction, design, and web development with simple JavaScript techniques
Foreword by John Resig, Creator of jQuery
Jonathan Chaffer
Karl Swedberg
[PACKT] open source

Quick answers to common problems
Ext JS 4 Web Application Development Cookbook
Over 110 easy-to-follow recipes backed up with real-life examples, walking you through basic Ext JS features to advanced application design using Sencha's Ext JS
Stuart Ashworth
Andrew Duncan
[PACKT] open source