

# RAWDATA

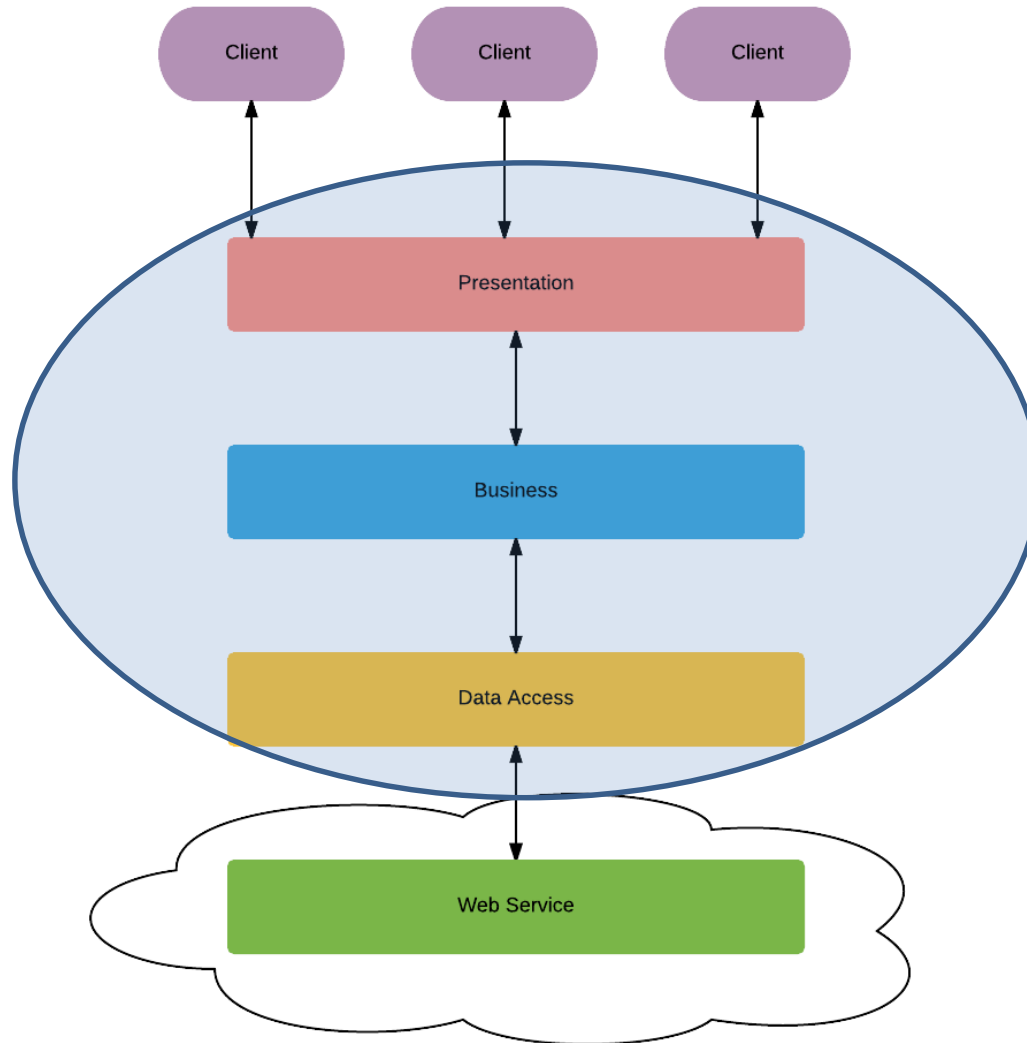
## Section 4

Troels Andreassen & Henrik Bulskov

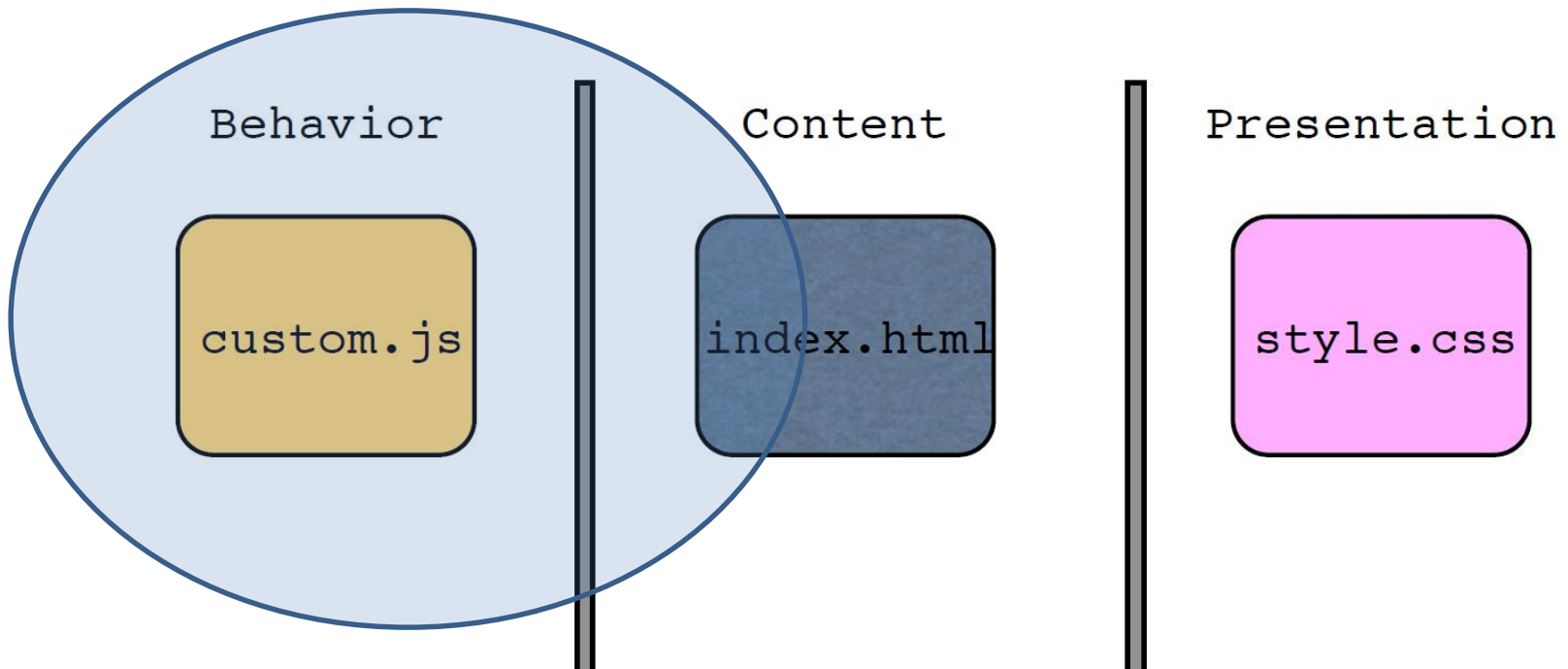
# What to do in section 4?

- JavaScript
  - Functions
  - JQuery
- **Single Page Applications**
  - Databinding
  - Modularity
- Responsiveness
  - Adaptive applications
  - Bootstrap

# System Development



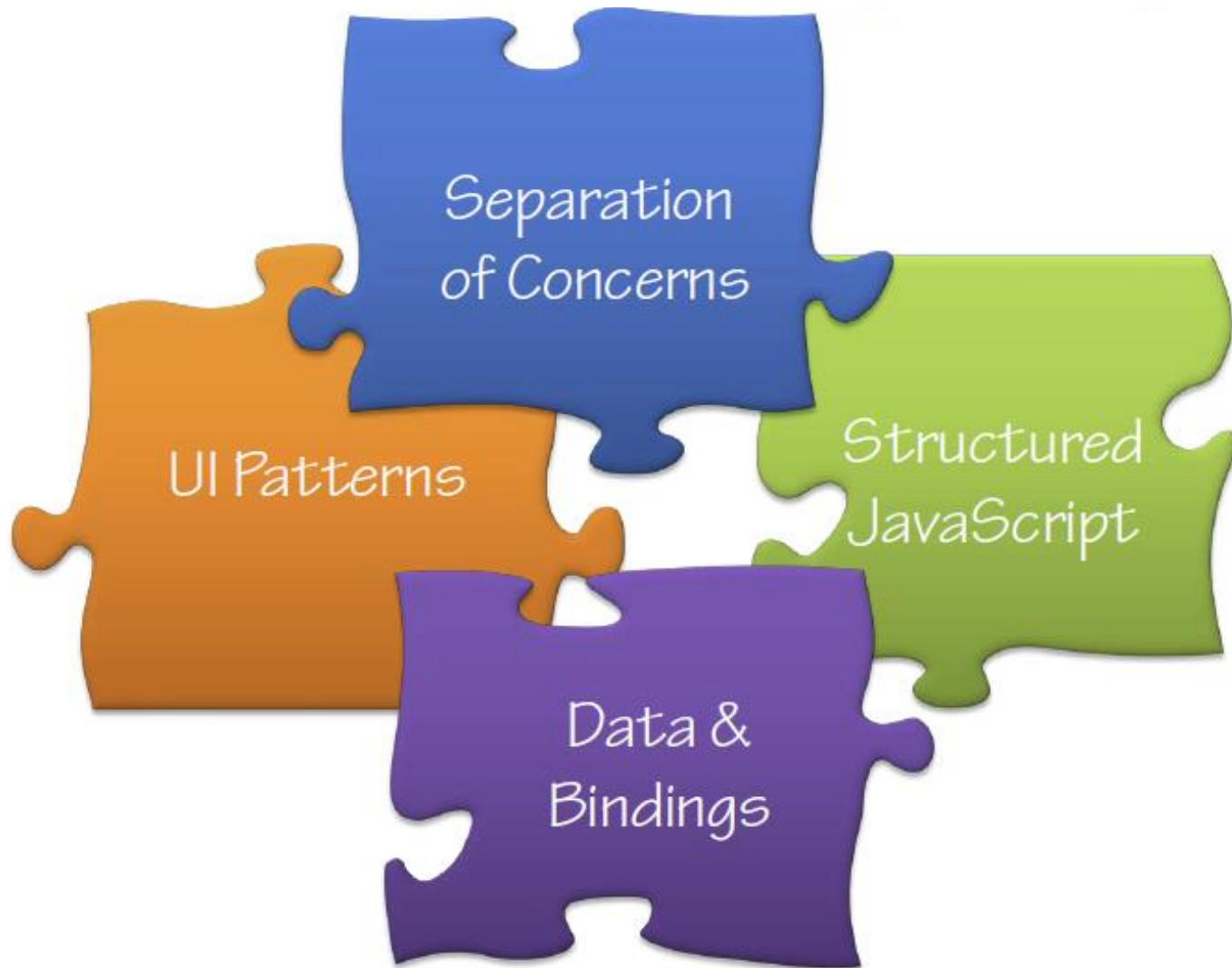
# Unobtrusive Design



# JS the important parts

- Dynamic
- Untyped
- First-class Functions
  - Function scope
- Prototype-based
- Fun, relaxing, and powerful
- To do it right – needs discipline!

# JavaScript Development



# AJAX

- Asynchronous JavaScript and XML
- A way to update a page without reloading

# jQuery Ajax

- Allows parts of a page to be updated
- Cross-Browser Support(Polyfill)
- Simple API
- GET and POST supported
- Load JSON, XML, HTML og even scripts



# jQuery Ajax Functions

- `$(selector).load()`
  - Loads HTML data from the server
- `$.get()` and `$.post()`
  - get raw data from server
- `$.getJSON()`
  - Get/Post and return JSON
- `$.ajax()`
  - Provides core functionality

# The ajax() Function

- The ajax() function provides extra control over making Ajax calls to a server
- Configure using JSON properties:
  - contentType
  - data
  - dataType
  - error
  - success
  - type (GET or POST)

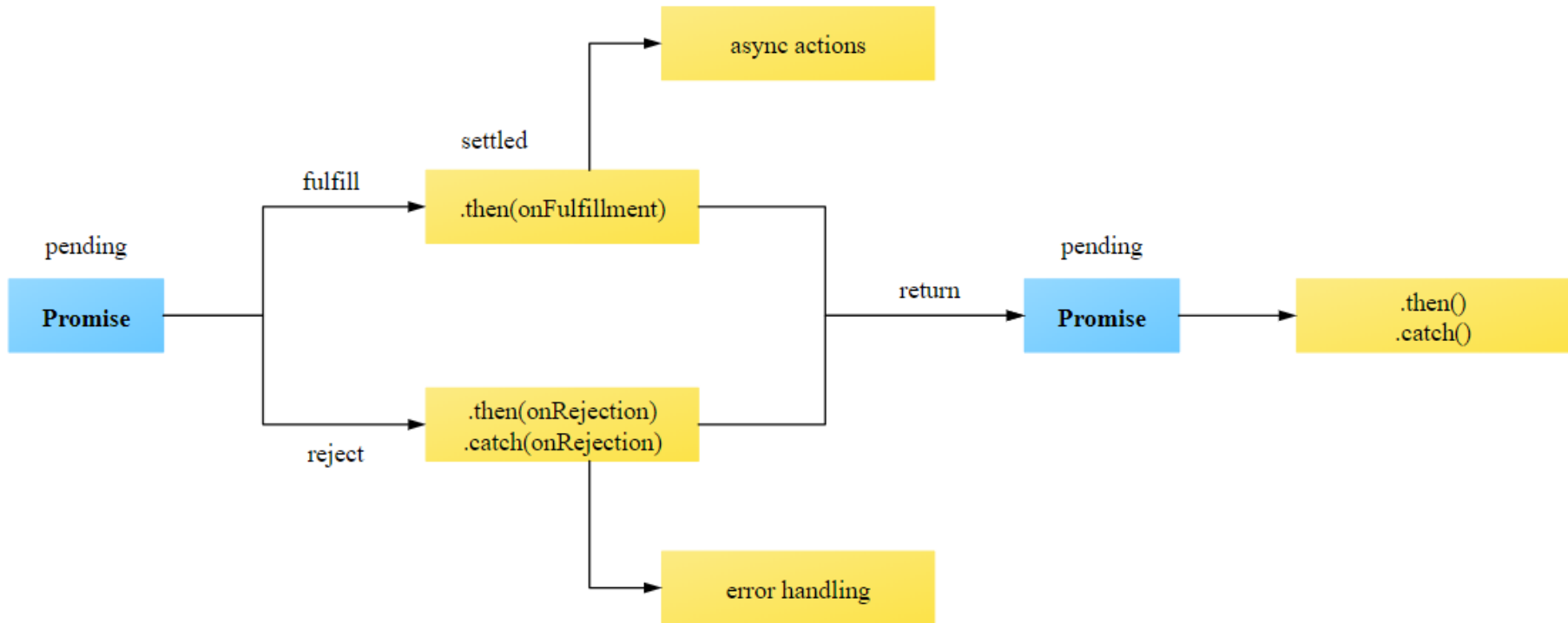
# ES6 AJAX Alternative

- `fetch()`
  - API that provides an interface for fetching resources (including across the network)
- Promises
  - A Promise is a proxy for a value not necessarily known when the promise is created

[https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)

[https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API/Using\\_Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch)

# ES6 Promises



# ES6 AJAX Examples

```
var getPosts = function (data) {  
  fetch("api/posts", { method: 'GET' })  
    .then(function (response) {  
      return response.json();  
    })  
    .then(function (json) {  
      data(json);  
    });  
};
```

```
var createPost = function (post, callback) {  
  var headers = new Headers();  
  headers.append("Content-Type", "application/json");  
  fetch("api/posts", { method: 'POST', body: JSON.stringify(post), headers })  
    .then(response => response.json())  
    .then(json => callback(json));  
}
```

# What is Knockout?

- JavaScript MVVM Framework
- MVVM – Model-View-View Model
  - Model – objects in your business domain
  - View – user interface that is visible to user
  - View Model – code representing data/operations on a UI
- Complementary to other JavaScript frameworks
  - e.g., jQuery, CoffeeScript, Prototype, etc.

# Key Knockout Concepts



# Knockout in 3 Steps

Declarative  
Binding

```
<input data-bind="value: firstName" />
```

Create an  
Observable

```
var myViewModel = {  
  firstName: ko.observable("John")  
};
```

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

Bind the ViewModel  
to the View

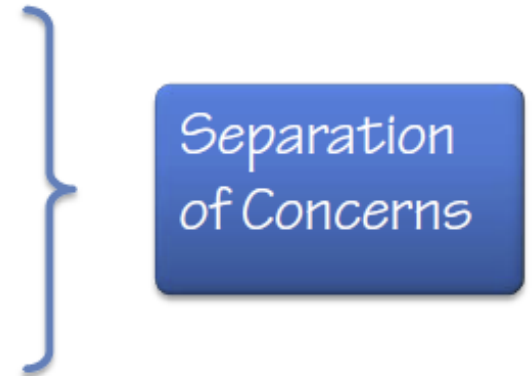


# Your First Knockout

- Resources:
  - <http://knockoutjs.com/>
  - <http://blog.stevensanderson.com/>
  - <http://www.knockmeout.net/>
  - <http://stackoverflow.com/questions/tagged/knockout.js>
  - <https://groups.google.com/forum/#!forum/knockoutjs>

# Separation, Organization, Data Binding

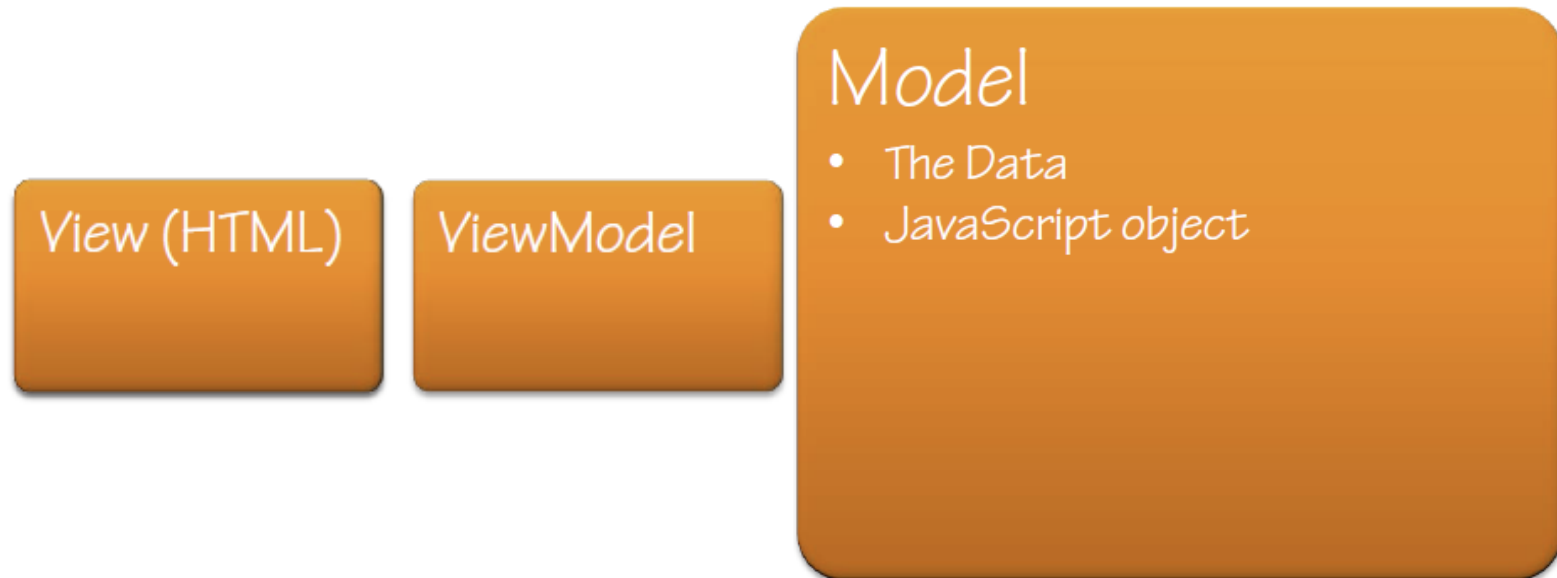
- MVVM
  - Foremost, a separation pattern
  - Model – View – ViewModel
  - Not technology specific
  - Works well with data binding



# MVVM Components



# MVVM Components



# MVVM Components

## View

- The web page, the HTML
- User friendly presentation of information

ViewModel

Model  
(JSON)

# MVVM Components

## ViewModel

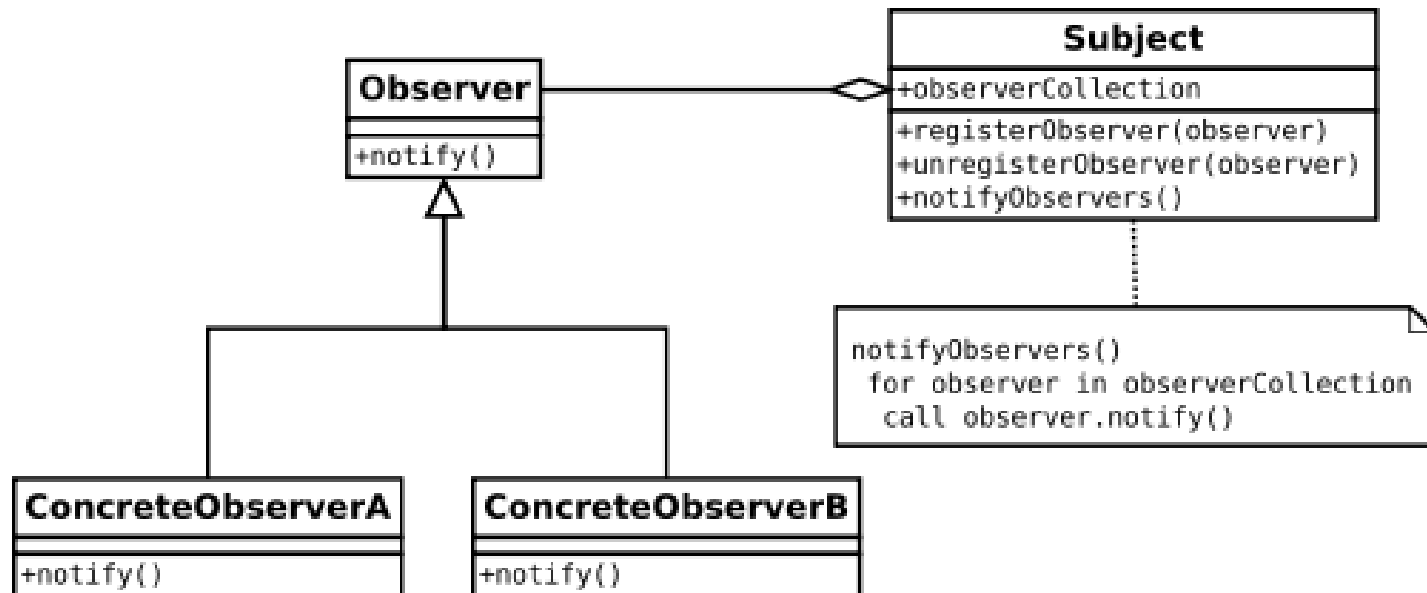
- Behavior and Data for the View
- Contains Properties, Methods & the Model

```
viewModel = {  
  id: ko.observable("123"),  
  salePrice: ko.observable(1995),  
  rating: ko.observable(4),  
  isInStock: ko.observable(true),  
  guitarModel: {  
    code: ko.observable("314ce"),  
    name: ko.observable("Taylor 314 ce")  
  },  
  showDetails: function () {  
    /* method goes here */  
  }  
};
```

View

Model  
(JSON)

# Observer Pattern



# Observables

- JavaScript functions
  - Not all browsers support JavaScript getters/setters
- Internally KO's bindings observe the observables

```
// read a value
var name = viewModel.name();

// write a value
viewModel.name("Peter");
```



# 3 Types of Observables

- Observable
  - Used for view model properties
- Observable array
  - Used for collections
- Computed observable
  - Encapsulate one or more other observables

# Computed observable

- Encapsulate one or more observables
- Need to manage this pointer

```
var viewModel = {  
  firstName: ko.observable("Peter"),  
  lastName: ko.observable("Smith")  
};  
  
viewModel.fullName = ko.computed(function() {  
  return this.firstName() + " " + this.lastName();  
}, viewModel);
```

# Computed observable

- Demo

# Observable Arrays

- Use with collections
- Detect changes to collection – add/remove
- Use Knockout array methods
  - Cross browser
  - Dependency Tracking
  - Clean Syntax

# Observable Array Methods

<code>list.indexOf("value")</code>	Returns zero-based index of item
<code>list.slice(2, 4)</code>	Returns items between start/end index
<code>list.push("value")</code>	Adds new item to end
<code>list.pop()</code>	Removes last item
<code>list.unshift("value")</code>	Inserts item at beginning
<code>list.shift()</code>	Removes first item
<code>list.reverse()</code>	Reverses order
<code>list.sort()</code>	Sorts the items
<code>list.remove(item)</code>	Removes specified item
<code>list.removeAll()</code>	Removes all items

# Observable Arrays

- Demo

# Knockout Bindings

- Built-in Bindings
  - Text and Appearance
  - Forms
  - Control Flow
  - Templates
- Custom Bindings

# Subscribing to Changes

- Register to be notified when changes occur
- Similar to writing code in a property setter in .NET
- Useful when you need to take action when a property changes

```
// Whenever the selectedMake changes, reset the selectedModel  
  
viewmodel.selectedMake.subscribe(function () {  
    viewmodel.selectedModel(undefined);  
}, viewmodel);
```



# Built-in Bindings – Text and Appearance

visible	Toggle visible/invisible of DOM element
text	Text value of DOM element
html	Raw HTML of DOM element
css	CSS class(es) of DOM element
style	Raw style attribute of DOM element
attr	Any arbitrary attribute of DOM element

# Built-in Bindings – Forms

click	Handler invoked when DOM element clicked
event	Handler invoked for arbitrary event on DOM element
submit	Handler invoked when form submitted
enable	DOM element enabled if true
disable	DOM element disabled if true
value	Value of DOM element
textInput	Same as value but do immediate live updates
checked	Attached to checkbox and radio button
options	Collection of elements in dropdown or multi-select
selectedOptions	Currently selected item(s) of dropdown or multi-select
	34

# Built-in Bindings – Control Flow

if	Executes if condition is true
ifnot	Executes if condition is false
foreach	Executes for each item in collection
with	Executes for object specified (child models)

# Custom Bindings

- Do not limit yourself to built-in bindings!
- Custom Bindings: the most useful extensibility point of Knockout

```
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.  
  }  
};
```

# Templates and Control of Flow

- Named Templates
- Control of Flow
- Binding Contexts
- Inline Templates
- Dynamically Choosing a Template
- Template Binding Helpers
- Containerless Bindings

# Templates

- Re-use code
- Encapsulate responsibility for a specific rendering
- Knockout supports many popular templating engines
  - jQuery Templates
  - Underscore
- Knockout has native templates

# Named Templates in <script> tags

- Encapsulate a template for re-use

```
<div data-bind="template: {name: 'personTpl'}"></div>

<script type="text/html" id="personTpl">
  <span data-bind="text: firstName"></span>
  <span data-bind="text: lastName"></span>
  <button data-bind="click: selectPerson">Add</button>
</script>
```

# Control Flow

if	Executes if condition is true
ifnot	Executes if condition is false
foreach	Executes for each item in collection
with	Executes for object specified (child models)



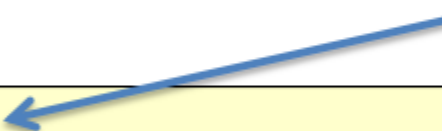
# Conditional Control of Flow

```
<p data-bind="if: lines().length > 0">  
  <span>Total value:</span>  
  <span data-bind="text: grandTotal()"></span>  
</p>
```

# Change Context “with”

```
<div>  
  <div data-bind="text: model().brand"></div>  
  <div data-bind="text: model().name"></div>  
</div>
```

Change the context  
with “with”



```
<div data-bind="with: model">  
  <div data-bind="text: brand"></div>  
  <div data-bind="text: name"></div>  
</div>
```

# Binding Contexts

- Sometimes in templates you want to change data binding scope (Data Context)
  - \$data
  - \$parent
  - \$parents
  - \$root

```
<button data-bind="click: $parent.addItem">Add</button>
```

# Inline Templates with Control of Flow

- If not reusing it, there is no need to name a template
- Control of flow elements create an implicit template

```
<tbody data-bind="foreach: lines">
  <tr>
    <td style="width: 100px;">
      <input data-bind="value: qty"/>
    </td>
    ...
  </tr>
</tbody>
```

Template is  
created  
anonymously  
and implicitly

# Knockout's Native Template Engine

- Templates inside DOM elements
  - `<script>`
  - Other DOM elements like `<div>`
- Anonymous / Inline templates
  - Templates without a name
  - Shortcuts to Anonymous template binding
    - `if`
    - `ifnot`
    - `with`
    - `foreach`

# if and ifnot

```
<div data-bind="template: {if: isSelected}">  
  <span data-bind="text:name"></span>  
</div>
```

```
<div data-bind="if: isSelected">  
  <span data-bind="text:name"></span>  
</div>
```

“if” shortcut

```
<div data-bind="template: {ifnot: isSelected}">  
  <span data-bind="text:name"></span>  
</div>
```

```
<div data-bind="ifnot: isSelected">  
  <span data-bind="text:name"></span>  
</div>
```

“ifnot” shortcut

# foreach and with

```
<div data-bind="template: {foreach: products}">  
  <span data-bind="text:name"></span>  
</div>
```

“foreach”  
shortcut

```
<div data-bind="foreach: products">  
  <span data-bind="text:name"></span>  
</div>
```

```
<div data-bind="template:  
  {if: selectedProduct, data: selectedProduct}">  
  <span data-bind="text:name"></span>  
</div>
```

“with”  
shortcut

```
<div data-bind="with: selectedProduct">  
  <span data-bind="text:name"></span>  
</div>
```

# Dynamically Change Templates

- Swap between multiple templates
- Bind the name of the template

```
<div data-bind="template: {name: templateChoice}">
  <script type="text/html" id="tmplSummary">
    ...
  </script>
  <script type="text/html" id="tmplDetails">
    ...
  </script>
```

```
my.vm.templateChoice = function () {
  return showDetails() ? "tmplDetails" : "tmplSummary";
};
```



# Control of Flow to Toggle Templates

- if and ifnot bindings

```
<div data-bind="ifnot: showDetails()">
    ...
</div>
<div data-bind="if: showDetails()">
    ...
</div>
```

```
my.vm.showDetails = ko.observable(false);
```

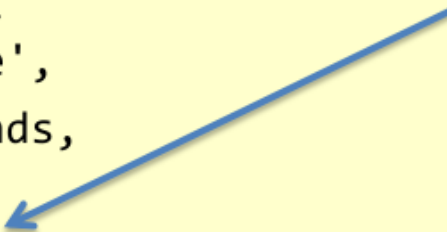
# Template Bindings

- name
  - Id of an element that contains the template
- foreach
  - Renders the template in foreach mode
  - (once for each item) foreach
- data
  - Object to supply as data for the template.
  - If omitted, uses foreach context or the current context data
- afterRender
  - Callback invoked after DOM elements are rendered
- afterAdd
  - Callback invoked after DOM elements are added
- beforeRemove
  - Callback invoked before DOM elements are removed

# Template Binding Helpers

```
<ul data-bind="template: {  
  name: 'friendsTemplate',  
  foreach: model().Friends,  
  beforeRemove: showAni,  
  afterAdd: hideAni}">  
</ul>
```

Callback to a method  
in the viewmodel



# Containerless Control of Flow Bindings

- Comment syntax
  - Unlike traditional Javascript template in `<script>`
- Use a template, without having a template!
- Comment based control flow syntax
  - if
  - ifnot
  - foreach
  - with
  - template

# Containerless Examples

```
<!-- ko with: selectedPerson -->  
  <span data-bind="text: name"></span>  
  <input data-bind="value: salary"></input>  
<!-- /ko -->
```

Reduces Unneeded  
Elements

```
<ul>  
  <li class="category">Acoustic Guitars<li>  
    <!-- ko foreach:acousticProducts -->  
    <li>  
      <span data-bind="text: shortDesc"></span>  
    </li>  
    <!-- /ko -->  
</ul>
```

Moves binding logic  
outside of elements