

# DEVHOL203 – Curing the asynchronous blues with the Reactive Extensions for JavaScript

---

## Introduction

This Hands-on-Lab (HOL) familiarizes the reader with the Reactive Extensions for JavaScript (RxJS). By exploring the framework through a series of incremental samples, the reader gets a feel for Rx's compositional power used to write asynchronous applications, based on the concept of observable collections.

## Prerequisites

We assume the following intellectual and material prerequisites in order to complete this lab successfully:

- Active knowledge of the JavaScript programming language and familiarity with the jQuery library.
- Feeling for the concept of asynchronous programming and related complexities.
- Visual Studio 2010 and .NET 4 (prior versions can be used but the lab is built for VS2010) installation.
- Installation of Rx for JavaScript from MSDN DevLabs at <http://msdn.microsoft.com/en-us/devlabs>.

## What is Rx?

Rx can be summarized in the following sentence which can also be read on the DevLabs homepage:

*Rx is a library for composing asynchronous and event-based programs using observable collections.*

Three core properties are reflected in here, all of which will be addressed throughout this lab:

- **Asynchronous and event-based** – As reflected in the title, the bread and butter of Rx's mission statement is to simplify those programming models. Everyone knows what stuck user interfaces look like, both on the Windows platform and on the web. And with the cloud around the corner, asynchrony becomes quintessential. Low-level technologies like DOM and custom events, asynchronous patterns, AJAX, etc. are often too hard.
- **Composition** – Combining asynchronous computations today is way too hard. It involves a lot of plumbing code that has little to nothing to do with the problem being solved. In particular, the data flow of the operations involved in the problem is not clear at all, and code gets spread out throughout event handlers, asynchronous callback procedures, and whatnot.
- **Observable collections** – By looking at asynchronous computations as data sources, we can leverage the active knowledge of LINQ's programming model. That's right: your mouse is a database of mouse moves and clicks. In the world of Rx, such asynchronous data sources are composed using various combinators in the LINQ sense, allowing things like filters, projections, joins, time-based operations, etc.

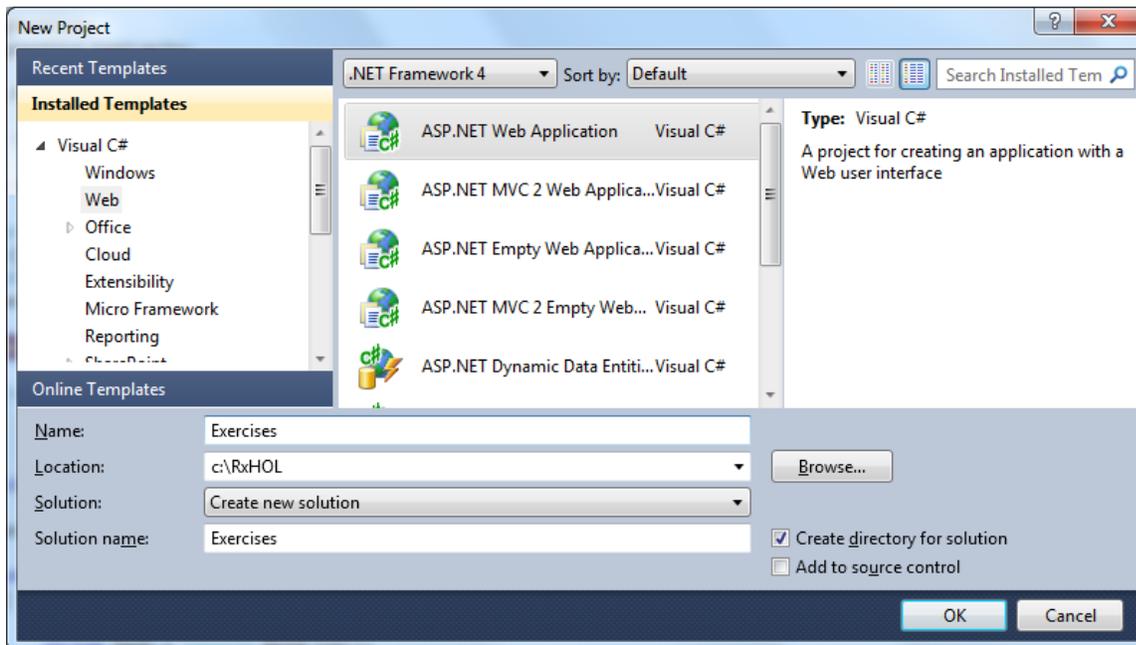
## Lab flow

In this lab, we'll explore Rx in a gradual manner. First, we'll have a look at the **core objects** of Rx which ship in as part of the Reactive Extensions for JavaScript. Once we understand those objects, we'll move on to show how the Rx libraries allow for creating **simple observable sequences** using factory methods. This allows for some basic experimentation. As we proceed, we'll introduce how to **bridge with existing asynchronous event sources** such as DOM events and the asynchronous pattern. Showing **more query operators** as we move along, we'll end up at a point where we start to **compose multiple asynchronous sources**, unleashing the true power of Rx.

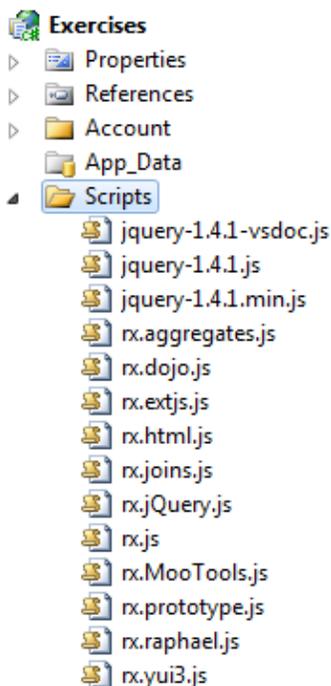
## Exercise 1 – Getting started with Rx interfaces and assemblies

**Objective:** Introducing the core Rx notions of observable collections and observers, which are reflected in several object definitions in the RxJS library. Those are mirror images of the two new .NET 4 interfaces `System.IObservable<T>` and `System.IObserver<T>`.

1. Open Visual Studio 2010 and go to File, New, Project... to create a new ASP.NET Web Application project in C#. Make sure the .NET Framework 4 target is set in the dropdown box at the top of the dialog.



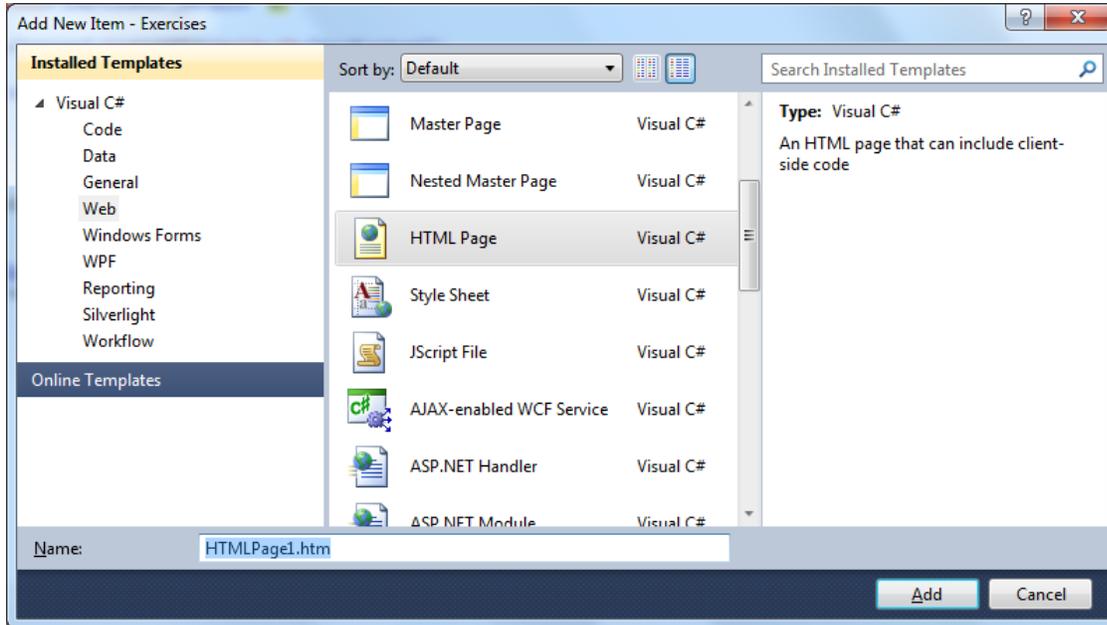
2. In order to get started with RxJS, we need to add a reference to the library which got installed on the lab machine in the `C:\Program Files\Microsoft Cloud Programmability\Reactive Extensions\v1.0.2590.0\RX_JS` folder. Right-click on the Scripts folder of your web project, and choose Add, Existing Item... to add all of the .js files in the folder mentioned here. The Project should look as follows:



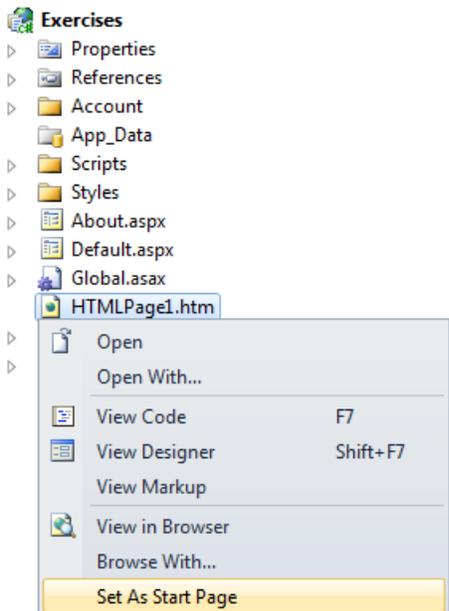


**Note:** When downloading RxJS from the DevLabs site, a single ZIP file will contain the same files as the ones that appear in this folder.

3. To start the exploration and to keep things as simple as possible, add a new HTML to the project using the Add, New Item... dialog:



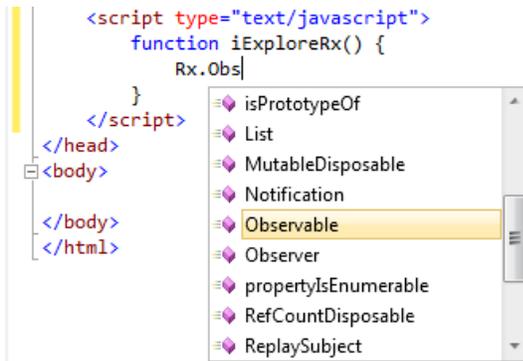
It's convenient to set this page as the start page for an easy F5 debugging experience:



4. Next, we'll import the script file into the HTML page using a script tag. The following markup is used to achieve this:

```
<head>  
  <title>Introductory Exercise</title>  
  <script type="text/javascript" src="Scripts\rx.js"></script>  
</head>
```

- Now we're ready to start exploring the RxJS library. Add another script tag with type text/javascript and add a function to it. At this point, we won't execute the script code yet but simply look at Visual Studio's IntelliSense provided for the rx.js file.



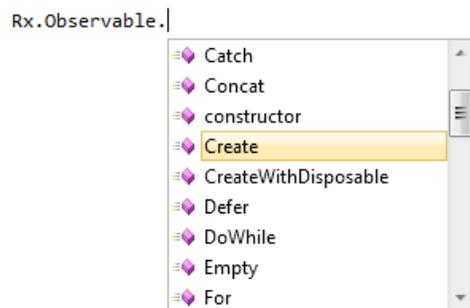
```
<script type="text/javascript">
  function iExploreRx() {
    Rx.Observable
  }
</script>
</head>
<body>
</body>
</html>
```

The screenshot shows the IntelliSense completion list for `Rx.Observable`. The list includes the following items:

- isPrototypeOf
- List
- MutableDisposable
- Notification
- Observable (highlighted)
- Observer
- propertyIsEnumerable
- RefCountDisposable
- ReplaySubject

In the screenshot above, we're typing `Rx.Observable` to see IntelliSense complete the various objects that have been added to the top-level "Rx" declaration. Of those, the key ones are `Observable` and `Observer`. Let's have a closer look now.

- First off, an observable is a collection that can notify subscribed observers about new values. This is where the essence of asynchronous programming in Rx lies in. By representing asynchronous computations as collections, a whole bunch of operators can be defined over those. We'll cover this aspect later. For now, let's have a look at what the `Observable` object has to offer:



```
Rx.Observable.|
```

The screenshot shows the IntelliSense completion list for `Rx.Observable`. The list includes the following items:

- Catch
- Concat
- constructor
- Create (highlighted)
- CreateWithDisposable
- Defer
- DoWhile
- Empty
- For

What you're seeing here are a bunch of operators that don't operate on an instance of an observable. For those who're in to C# and other managed languages, the functions shown here roughly correspond to static methods (excluding extension methods). In the next exercise, we'll explore how to create observable sequences using some of those methods that act as factories.



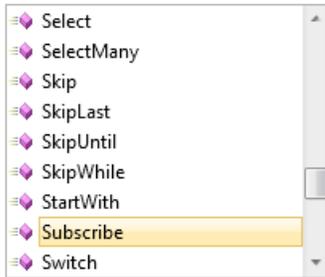
**Note:** In languages designed by people who care about types (much like one of the authors), Rx uses two interfaces to capture the notion of an observable and an observer. Those are called `IObservable` and `IObserver`, respectively. Starting with .NET 4, both interfaces are built in to `microsoft.winfx.dll`. On .NET 3.5 and in the Silverlight flavors of Rx, a separate assembly called `System.Observable.dll` has to be included in the project in order to get access to those two interfaces.

For now, just enter the following code to create one of the most basic observable sequences. All it does is letting its observers know about a single value that's become available:

```
var answer = Rx.Observable.Return(42);
```

With this observable (single-element) sequence in place, we can have a look at the equivalent of instance methods in languages like C#. Let's "dot into" the `answer` object and see what's available:

```
var answer = Rx.Observable.Return(42);
answer.Sub|
```



LINQ-savvy readers will immediately recognize some of the Standard Query Operators, but we'll skip over this aspect for now. The highlighted Subscribe function is what's important to have a solid understanding about right now. Contrast to enumerable sequences, observable sequences don't have a GetEnumerator function. Where the use of an enumerator is to *pull* data to you ("Hey, give me the next element please? Got more?"), a *push*-based mechanism is desired for observable sequences. So, instead of *getting* an enumerator, you have to *give* an observer to an observable sequence. This observer will then be used to notify you about the availability of new data, e.g. when an asynchronous web service call completes or an event was raised. Let's have a closer look at the Subscribe function now:

```
var answer = Rx.Observable.Return(42);
answer.Subscribe(|
    Subscribe(10, m0, n0)
```

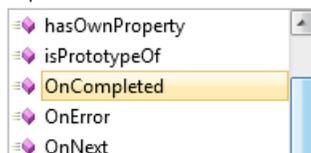
Subscribe can take up to three parameters which represent the observer. The role of an observer is to receive notifications from the observable sequence. Three such messages exist as can be witnessed by the following experiment. Let's create an Observer using the Observer.Create function:

```
var observer = Rx.Observer.Create(
    function (x) {
        document.write("The answer is " + x);
    }
);
```

While typing the Create method you will have noticed there are up to three parameters supported. In the above, we've only specified one. This particular one acts as the OnNext handler which will get called by an observable sequence when data is available. For example, if we give the observer to the "answer" sequence, a call will be made to the OnNext function shown above. Two other functions exist:

```
var observer = Rx.Observer.Create(
    function (x) {
        document.write("The answer is " + x);
    }
);
```

```
observer.On|
```



We'll explore the role of OnCompleted and OnError in the next exercise, but suffice it to say that those functions act as callbacks used by an observable sequence. In the Create call above, we've simply omitted everything but the OnNext handler. Now we can pass the observer to the observable "answer":

```
answer.Subscribe(observer);
```



**Note:** Due to size minimization of the rx.js library, parameter names don't look very meaningful. Looking at the .NET version of the corresponding interface would reveal the intent of those parameters. While IntelliSense can be provided for JavaScript libraries in Visual Studio 2008 and beyond, we haven't done so (yet). On source of additional complexity involved in this is the absence of overloads in JavaScript. In fact, we do support various overloads for a bunch of functions by means of parameter checks. How this would look in IntelliSense isn't quite clear at this point.

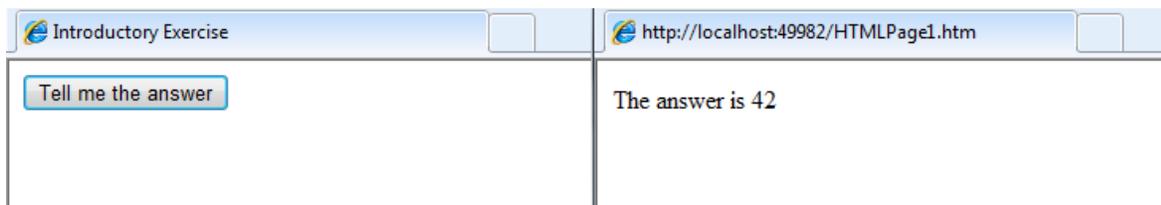
7. So far, we've written the following piece of code that we shall recap briefly. An observable sequence is an object that will produce values in an asynchronous manner. In order for it to signal the availability of such values, it uses an observer to make callbacks to. Three such callbacks exist, which are subject of the next exercise, but for now we've only specified the OnNext handler which will receive the observable's values:

```
<script type="text/javascript">
  function iExploreRx() {
    var answer = Rx.Observable.Return(42);
    var observer = Rx.Observer.Create(
      function (x) {
        document.write("The answer is " + x);
      }
    );
    answer.Subscribe(observer);
  }
</script>
```

In order to show this code in action, let's hook up the iExploreRx function to a button. Later we'll use jQuery to facilitate easier configuration of event handlers and so on:

```
<body>
  <button onclick="javascript:iExploreRx()">Tell me the answer</button>
</body>
```

Clicking the button will print "The answer is 42" on the screen, as illustrated below.



**Note:** Use of the document.write and document.writeln functions is quite invasive as it doesn't append the content to the existing document. Beyond this introductory example, we'll use more sophisticated HTML and JavaScript facilities to put responses on the screen.

The reader may recall that Subscribe didn't just take one but a possible total of three parameters. What are the remaining two for? Well, instead of having you create an observer manually using the Observer.Create function, one can pass the triplet of callback functions (or certain subsets thereof) directly to the Subscribe function:

```
var answer = Rx.Observable.Return(42);
answer.Subscribe(
  function (x) {
    document.write("The answer is " + x);
  }
);
```

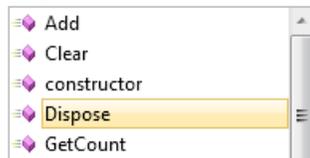
Other callback functions will be discussed in the next exercise. The reader should run the modified code to ensure the same result is produced, this time with less coding effort.

- One thing we haven't quite talked about yet is the return value of a Subscribe function call. Rx differs from other event models (like classic .NET events) in the way unsubscription (or detachment of a handler if you will) is accomplished. Instead of requiring the use of another construct to unsubscribe, the return value of Subscribe represents a handle to the subscription. In order to unsubscribe, one calls Dispose on the returned object:

```
var answer = Rx.Observable.Return(42);
var handle = answer.Subscribe(
    function (x) {
        document.write("The answer is " + x);
    }
);

// Here the OnNext function can still be called.
```

handle.Disp



For trivial sequences like the one constructed using Return, the use of Dispose isn't applicable often. However, when bridging with ongoing event streams – e.g. DOM events as we shall see later – this functionality comes in handy.

**Conclusion:** Observable and Observer objects represent a data source and a listener, respectively. In order to observe an observable sequence's notifications, one gives it an observer object using the Subscribe function, receiving a handle object that be used to unsubscribe by calling Dispose. In order to gain access to those objects and functions, one has to import the rx.js script library as discussed here.

## Exercise 2 – Creating observable sequences

**Objective:** Observable sequences are rarely provided by implementing the IObservable<T> interface directly. Instead a whole set of factory methods exist that create primitive observable sequences. Those factory methods provide a good means for initial exploration of the core notions of observable sources and observers.

- Ensure the project setup of Exercise 1 is still intact, i.e. containing the references to our JavaScript files. Also ensure the following skeleton code is put in place. Here we'll explore OnError and OnCompleted as well:

```
var source = null; // We'll explore some factory methods here
var subscription = source.Subscribe(
    function (next) {
        $("<p/>").html("OnNext: " + next).appendTo("#content");
    },
    function (exn) {
        $("<p/>").html("OnError: " + exn).appendTo("#content");
    },
    function () {
        $("<p/>").html("OnCompleted").appendTo("#content");
    });
```

- We'll start by looking at the Empty factory method:

```
var source = Rx.Observable.Empty();
```

Running this code produces the following output:

OnCompleted

In other words, the empty sequence simply signals completion to its observers by calling `OnCompleted`. This is very similar to LINQ to Object's `Enumerable.Empty` or an empty array. For those enumerable sequences, the first call to the enumerator's `MoveNext` method would return `false`, signaling completion.



**Background:** One may wonder when observable sequences start running. In particular, what's triggering the empty sequence to fire out the `OnCompleted` message to its observers? The answer differs from sequence to sequence. Most of the sequences we're looking at in this exercise are so-called *cold observables* which means they start running upon subscription. This is different from *hot observables* such as mouse move events which are flowing even before a subscription is active (there's no way to keep the mouse from moving after all...).

3. Besides the `OnCompleted` message, `OnError` is also a terminating notification, in a sense no messages can follow it. Where `Empty` is the factory method creating an observable sequence that immediately triggers completion, the `Throw` method creates an observable sequence that immediately triggers an `OnError` message to observers:

```
var source = Rx.Observable.Throw("Oops!");
```

Running this code produces the following output:

```
OnError: Oops
```



**Background:** The `OnError` message is typically used by an observable sequence (not as trivial as the one simply returned by a call to `Throw`) to signal an error state which could either originate from a failed computation or the delivery thereof. Following the semantic model of the CLR's exception mechanism, errors in Rx are always terminating and exhibit a fail-fast characteristic, surfacing errors through observer handlers. More advanced mechanisms to deal with errors exist in the form of handlers called `Catch`, `OnErrorResumeNext` and `Finally`. We won't discuss those during this HOL, but their role should be self-explanatory based on corresponding language constructs in various managed languages.

4. One final essential factory method or primitive constructor is called `Return`. Its role is to represent a single-element sequence, just like a single-cell array would be in the world of enumerable sequences. The behavior observed by subscribed observers is two messages: `OnNext` with the value and `OnCompleted` signaling the end of the sequence has been received:

```
var source = Rx.Observable.Return(42);
```

Running this code produces the following output:

```
OnNext: 42  
OnCompleted
```



**Background:** `Return` plays an essential role in the theory behind LINQ, known as *monads*. Together with an operator called `SelectMany` (which we'll learn about more later on), they form the primitive functions needed to leverage the power of monadic computation. More information can be found by searching the web using `monad` and `functional` as the keywords.

5. At this point, we've seen the most trivial observable sequence constructors that are intimately related to an observer's triplet of methods. While those are of interest in certain cases, more meaty sequences are worth to explore as well. The `Range` operator is just one operator that generates such a sequence. Symmetric to the same operator on `Enumerable`, `Range` does return a sequence with 32-bit integer values given a starting value and a length:

```
var source = Rx.Observable.Range(5, 3);
```

Running this code produces the following output:

```
OnNext: 5  
OnNext: 6  
OnNext: 7  
OnCompleted
```



**Note:** As with all the sequences mentioned in this exercise, `Range` is a *cold observable*. To recap, this simply means that it starts producing its results to an observer upon subscription. Another property of cold observable sequence is that every subscription will cause such reevaluation. Thus, if two calls to `Subscribe` are made, both of the observers passed to those calls will receive the messages from the observable. It's not because the data observation has run to completion for one observer that other observers won't run anymore. Whether or not the produced data is the same for every observer depends on the characteristics of the sequence that's being generated. For deterministic and "purist functional" operators like `Return` and `Range`, the messages delivered to every observer will be the same. However, one could imagine other kinds of observable sequences that depend on side-effects and thus deliver different results for every observer that subscribes to them.

- To generalize the notion of sequence creation in terms of a generator function, the `Generate` constructor function was added to Rx. It closely resembles the structure of a for-loop as one would use to generate an enumerable sequence using C# iterators (cf. the "yield return" and "yield break" statements). To do so, it takes a number of delegate-typed parameters that expect function to check for termination, to iterate one step and to emit a result that becomes part of the sequence and is sent to the observer:

```
var source = Rx.Observable.Generate(  
    0, function (i) { return i < 4; }, function (i) { return i + 1; }, // Like a for loop  
    function (i) { return i * i; });
```

Running this code produces the following output:

```
OnNext: 0  
OnNext: 1  
OnNext: 4  
OnNext: 9  
OnCompleted
```



**Note:** A sister function called `GenerateWithTime` exists that waits a computed amount of time before moving on to the next iteration. We'll look at this one in just a moment.

- One operator that may seem interesting from a curiosity point of view only is called `Never`. It creates an observable sequence that will never signal any notification to a subscribed observer:

```
var source = Rx.Observable.Never();
```

Running this code shouldn't produce any output till the end of mankind.



**Background:** One reason this operator has a role is to reason about *composition* with other sequences, e.g. what would it mean to concatenate a finite and an infinite sequence? Should the result also exhibit an infinite characteristic? Those answers are essential to ensuring all of the Rx semantics make sense and are consistent throughout various operators. Besides the operator's theoretical use, there are real use cases for it as well. For example, you may use it to ensure a sequence never terminates by avoiding an `OnCompleted` handler getting triggered. Another use is to test whether an application does not hang in the presence of non-termination.

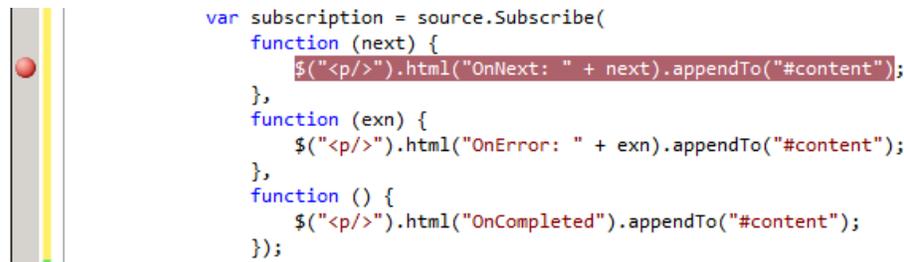
8. Finally, let's inspect the behavior of an observable sequence by looking at it through the lenses of the debugger in Visual Studio. We'll use the following fragment to do so:

```
var source = Rx.Observable.GenerateWithTime(
    0,
    function(i) { return i < 5; },
    function(i) { return i + 1; },
    function(i) { return i * i; },
    function(i) { return i * 1000; }
);

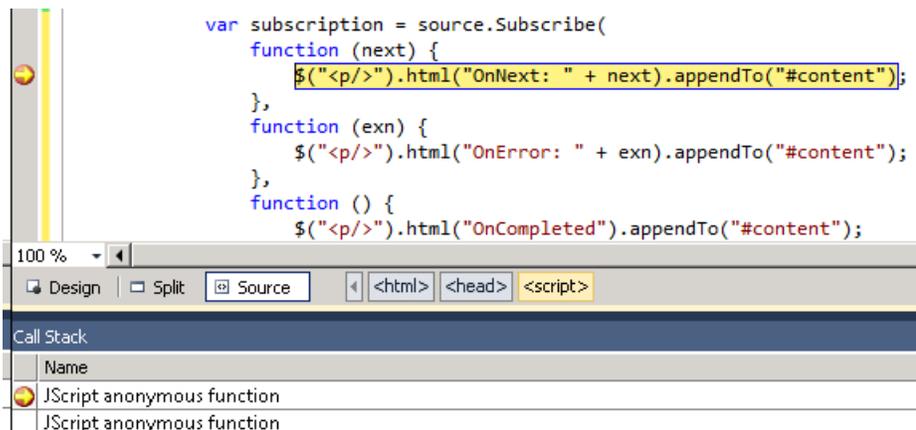
var subscription = source.Subscribe(
    function (next) {
        $("<p/>").html("OnNext: " + next).appendTo("#content");
    },
    function (exn) {
        $("<p/>").html("OnError: " + exn).appendTo("#content");
    },
    function () {
        $("<p/>").html("OnCompleted").appendTo("#content");
    });
```

This sample uses a slightly different operator called `GenerateWithTime` that allows specifying iteration time between producing results, dependent on the loop variable. In this case, 0 will be produced upon subscription, followed by 1 a second later, then 4 two seconds later, 9 three seconds later and 16 four seconds later. Notice how the notion of time – all important in an asynchronous world – is entering the picture here.

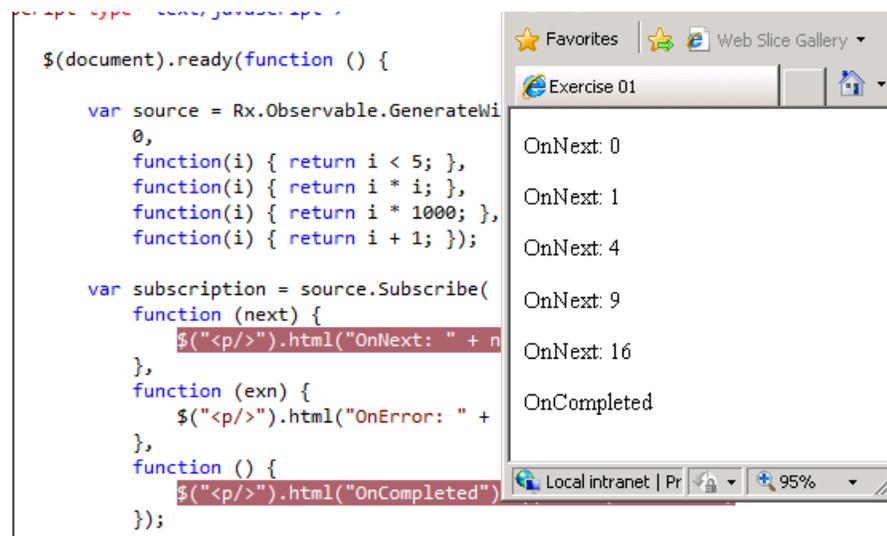
- a. Set a breakpoint on the highlighted lambda expression body using F9. Notice you need to be inside the lambda body with the cursor in the editor in order for the breakpoint to be set on the body and not the outer method call to `Subscribe`.



- b. Start running the program by pressing F5. At this time, we end up hitting our breakpoint as can be seen from the debugger:



- c. The reader should feel free to hit F5 a couple more times to see the breakpoint getting hit for every subsequent OnNext message flowing out of the observable sequence. Setting a breakpoint on the OnCompleted will show the same behavior for the GenerateWithTime sequence:



```
$(document).ready(function () {  
  
    var source = Rx.Observable.GenerateWithTime(0, 1000, function(i) { return i < 5; }, function(i) { return i * i; }, function(i) { return i * 1000; }, function(i) { return i + 1; });  
  
    var subscription = source.Subscribe(  
        function (next) {  
            $("<p/>").html("OnNext: " + next);  
        },  
        function (exn) {  
            $("<p/>").html("OnError: " + exn);  
        },  
        function () {  
            $("<p/>").html("OnCompleted");  
        });  
});
```



**Note:** In the .NET version of Rx, the place where notifications are delivered is dependent on aspects of concurrency, which can be controlled using an IScheduler object. This notion is not present in RxJS since JavaScript has a totally different – mostly non-existent – concurrency philosophy.

**Conclusion:** Creating observable sequences does not require manual implementation of the IObservable<T> interface, nor does the use of Subscribe require an IObserved<T> implementation. For the former, a series of operators exist that create sequences with zero, one or more elements. For the latter, Subscribe extension methods exist that take various combinations of OnNext, OnError and OnCompleted handlers in terms of delegates.

### Exercise 3 – Importing DOM events into Rx

**Objective:** Creating observable sequences out of nowhere using various factory “constructor” methods encountered in the previous exercise is one thing. Being able to bridge with existing sources of asynchrony in the framework is even more important. In this exercise we’ll look at the jQuery toObservable operator that allows “importing” a DOM or custom event into Rx as an observable collection. Every time the event is raised, an OnNext message will be delivered to the observable sequence.

**Background:** Rx doesn’t aim at replacing existing asynchronous programming models in JavaScript. Technologies like DOM events (using jQuery), AJAX and whatnot are perfectly suited for direct use. However, once composition enters the picture, using those low-level concepts tends to be a grueling experience dealing with resource maintenance (e.g. when to unsubscribe) and often has to reinvent the wheel (e.g. how do you “filter” an event?). Needless to say, all of this can be very error prone and distracts the developer from the real problem being solved. In this sample and the ones that follow, we’ll show where the power of Rx comes in: composition of asynchronous data sources.

1. The HTML DOM is a good sample of an API that’s full of events. Starting from our existing web project, we should still have the single HTML page lying around. In order to bridge with the DOM, we’ll be using jQuery and Rx’s wrapper for it. The following script references have to be added to the page to enable this:

```
<script type="text/javascript" src="Scripts/jquery-1.4.1.min.js"></script>  
<script type="text/javascript" src="Scripts/rx.js"></script>  
<script type="text/javascript" src="Scripts/rx.jquery.js"></script>
```

2. Using jQuery, we can write the following code to attach an event handler to the document's ready event:

```
$(document).ready(function () {  
});
```

Obviously, the above doesn't do much as we've specified a no-op function for the handler. In order to be able to contrast this kind of DOM event handling from the Rx approach using observable sequences, let's show hooking up the mousemove event within the ready event handler:

```
$(document).ready(function () {  
    $(document).mousemove(function (event) {  
        // A position tracking mechanism, updating a paragraph called "content"  
        $("    });  
});
```

To get the above to work, add a content <p> tag to the page:

```
<p id="content" />
```

While this works great, there are a number of limitations associated with classic DOM events:

- Events are hidden data sources. It requires looking at the handler's code to see this. Did you ever regard the mousemove event as a collection of points? In the world of Rx, we see events as just another concrete form of observable sequences: your mouse is a database of points!
- Events cannot be handed out, e.g. an event producing Point values cannot be handed out to a GPS service. The deeper reason for this is that events are not first-class objects. In the world of Rx, each observable sequence is represented using an object that can be passed around or stored.
- Events cannot be composed easily. For example, you can't hire a mathematician to write a generic filter operator that will filter an event's produced data based on a certain criterion. In the world of Rx, due to the first-class object nature of observables, we can provide generic operators like Where.
- Events require manual handler maintenance which requires you to remember the function that was handed to it. It's like keeping your hands on the money you paid for your newspaper subscription in order to be able to unsubscribe. In the world of Rx, you get a handle to unsubscribe using Dispose.

3. Now let's see how things look when using Rx. To import an event into Rx using jQuery, we use the toObservable operator, which we tell the jQuery Event object that will be raised by the event being bridged:

```
$(document).ready(function () {  
    $(document).toObservable("mousemove").Subscribe(function (event) {  
        // A position tracking mechanism, updating a paragraph called "content"  
        $("    });  
});
```

We'll now explain this fragment in depth to drive home the points we made before:

- The toObservable operator turns the given event – specified as a string – in an observable sequence with a jQuery Event object.
- When calling Subscribe, a handler is attached to the underlying event. For every time the event gets raised, the jQuery Event object is sent to all of the observable's subscribers.
- Inside our OnNext handler, we can get the mouse position via the pageX and pageY properties. It goes without saying that those properties could be wrapped in a JSON object representing a point.
- If desired, clean-up of the event handler can be taken care of by calling Dispose on the object returned by the toObservable operator.

4. To master the technique a bit further, let's have a look at another DOM event. First add an input element to our HTML page.

```
<input id="textbox" type="text" size="100" /><br />
```

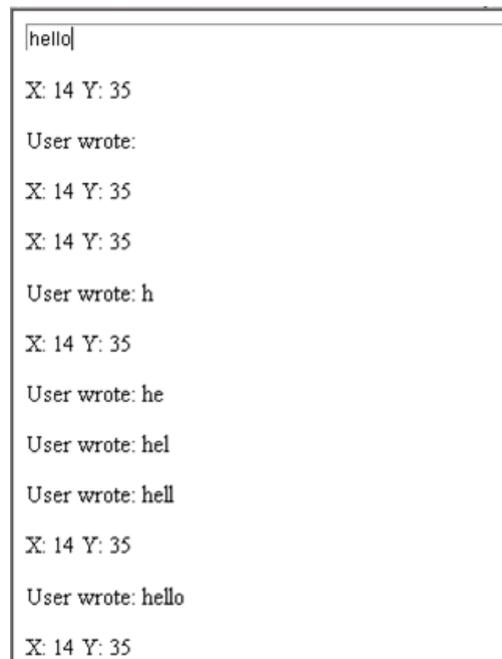
Restructure the code to look as follows in order to have both the DOM document's mousemove and the input's keyup event hooked up. We'll write output to the content element as a form of logging. In Exercise 5 we'll learn about a specialized operator (called Do) that can be used for this purpose.

```
$(document).ready(function () {  
    $(document).toObservable("mousemove").Subscribe(function (event) {  
        $("<p/>").text("X: " + event.pageX + " Y: " + event.pageY).appendTo("#content");  
    });  
  
    $("#textbox").toObservable("keyup").Subscribe(function (event) {  
        $("<p/>").text("User wrote: " + $(event.target).val()).appendTo("#content");  
    });  
});
```

At this point it may seem we haven't gained too much yet. Things are just "different". What really matters though is that we've put us in the world of Observable object, over which a lot of operators are defined that we'll talk about in a moment.

For one thing, notice all the hoops one has to go through in order to get the text out of a keyup event: get the target and call a val() function. As mentioned before, classic DOM events don't explicitly exhibit a data-oriented nature. This particular event is a great example of this observation: from the event handler of a keyup event one doesn't immediately get the text after the change has occurred, even though that's what 99% of uses of the event are about (the other 1% may be justified by "state invalidation handling", e.g. to enable "Do you want to save changes?" behavior). Using operators like Select we can simplify this code, as we'll see in the next exercise.

5. A fragment of sample output is shown below:



```
|hello|  
X: 14 Y: 35  
User wrote:  
X: 14 Y: 35  
X: 14 Y: 35  
User wrote: h  
X: 14 Y: 35  
User wrote: he  
User wrote: hel  
User wrote: hell  
X: 14 Y: 35  
User wrote: hello  
X: 14 Y: 35
```

**Conclusion:** DOM events are just one form of asynchronous data sources. In order to use them as observable collections, Rx provides the jQuery operator toObservable. In return one gets Event objects containing the jQuery event information.

## Exercise 4 – A first look at some Standard Query Operators

**Objective:** Looking at observable sequences as asynchronous data sources is what enables them to be queried, just like any other data source. Who says querying in the context of C# programming nowadays, immediately thinks LINQ. In this exercise we'll show how to use the LINQ syntax to write queries over observable collections.

1. Continuing with the previous exercise's code, let's have a look back at the code we wrote to handle various UI-related events brought into Rx using the jQuery toObservable operator:

```
$(document).toObservable("mousemove").Subscribe(function (event) {
    $("<p/>").text("X: " + event.pageX + " Y: " + event.pageY).appendTo("#content");
});

$("#textbox").toObservable("keyup").Subscribe(function (event) {
    $("<p/>").text("User wrote: " + $(event.target).val()).appendTo("#content");
});
```

Recall the moves and input collections contain jQuery Event objects. Quite often we're not interested in all of the information an Event object captures and want to "shake off" redundant stuff.

2. In a classic jQuery event world – and in any other programming model for asynchronous data sources before the advent of Rx for that matter – such data-intensive operations often led to imperative code. For events, many of you will likely have written code like this:

```
function handleMouseMove(event) {
    if (event.pageX === event.pageY) {
        // Only respond to events for mouse moves over the first bisector of the page.
    }
}

function handleKeyUp(event) {
    var text = $(event.target).val();
    // And now we can forget about the rest of the event object's data...
}
```

What we've really done here is mimicking data-intensive "sequence operators" in an imperative way. The first sample shows a *filter* using an if-statement; the second one embodies a *projection* using another local variable. In doing so, we've lost an important property though. The parts omitted by green comments no longer directly operate on an event but are lost in a sea of imperative code. In other words, it's not possible to filter an event and get another event back.

3. In the brave new Rx world, we can do better than this. Since observable sequences have gotten a first-class status by representing them as Observable *objects*, we can provide operators over them by providing a set of *functions*. Let's have a look at how we'd revamp our imperative event handler code using those query operators over observable sequences. First, let's turn the event-based input sequences into what we wish they'd look like:

```
var moves = $(document).toObservable("mousemove")
    .Select(function(event) {
        return { pageX : event.pageX, pageY : event.pageY };
    });

var input = $("#textbox").toObservable("keyup")
    .Select(function(event) {
        return $(event.target).val();
    });

var movesSubscription = moves.Subscribe(function (pos) {
    $("<p/>").text("X: " + pos.pageX + " Y: " + pos.pageY).appendTo("#content");
});
```

```
var inputSubscription = input.Subscribe(function (text) {
    $("<p/>").text("User wrote: " + text).appendTo("#content");
});
```

Using the Select function, we project away the Event object in favor of a JavaScript object with points and a string, respectively. As a result both the moves and input sequences now are observable sequences of a meaningful data type that just captures what we need.



**Background:** The ability to represent asynchronous data sources as first-class objects is what enables operators like this to be defined. Being able to produce an observable sequence that operates based on input of one or more sequences is not just interesting from a data point of view. Equally important is the ability to control the lifetime of subscriptions. Consider someone subscribes to, say, the input sequence. What really happens here is that the Select operator's result sequence gets a request to subscribe an observer. On its turn, it propagates this request to its source, which is a sequence produced by toObservable. Ultimately, the event-wrapping source sequence hooks up an event handler. Disposing a subscription is propagated in a similar manner.

- With the projections in place to reduce the noise on input sequences, we can now easily filter the mouse moves to those over the first bisector (where x and y coordinates are equal). How do you perform a filter in LINQ? Use the Where operator:

```
var overFirstBisector = moves
    .Where(function(pos) {
        return pos.pageX === pos.pageY;
    });

var movesSubscription = overFirstBisector.Subscribe(function (pos) {
    $("<p/>").text("Mouse at: "+pos.pageX+", "+pos.pageY).appendTo("#content");
});
```

The type for both moves and overFirstBisector will be an object with a pageX and pageY property.

- A sample of the output is shown below. All of the emitted mouse move messages satisfy the filter constraint we specified in the query:

```

|Hello|
User wrote:
User wrote: H
User wrote: H
User wrote: He
User wrote: Hel
User wrote: Hell
User wrote: Hello
Mouse at: 5,5
Mouse at: 4,4
```

**Conclusion:** The first-class nature of observable sequences as Observable objects is what enables us to provide operators (sometimes referred to as combinators) to be defined over them. The majority of those operators produce another observable sequence. This allows continuous “chaining” of operators to manipulate an asynchronous data source's emitted results till the application's requirements are met. Others will be discussed further on.

## Exercise 5 – More query operators to tame the user’s input

**Objective:** Observable sequences are often not well-behaved for the intended usage. We may get data presented in one form but really want it in another shape. For this, simple projection can be used as shown in the previous exercise. But there are far more cases of ill-behaved data sources. For example, duplicate values may come out. But there’s more beyond the perspective of observable sequences as “just data sources”. In particular, asynchronous data sources have an intrinsic notion of timing. What if a source goes too fast for consumers to deal with their data? We’ll learn how to deal with those situations in this exercise.

From this exercise on, we’ll be floating on a common theme of the typical “dictionary suggest” sample for asynchronous programming. The idea is to let the user type a term in an input element and show all the words starting with the term, by consulting a web service. To keep the UI from freezing, we got to do this kind of stuff in an asynchronous manner. Rx is a great fit for this kind of composition. But first things first, let’s see how the input element is behaving.

1. In what follows, we won’t need the mousemove event anymore, so let’s stick with just a single input element and its (projected) keyup event:

```
var input = $("#textbox").toObservable("keyup")
    .Select(function (event) {
        return $(event.target).val();
    });

var inputSubscription = input.Subscribe(function (text) {
    $("<p/>").text("User wrote: " + text).appendTo("#content");
});
```

2. Now, let’s carry out a few experiments. Load the web page and type “reactive” (without the quotes) in the input box. Obviously you should see no less than eight events being handled through the observer. However, notice what happens if you select a single letter in the word and overwrite it by the same letter:

r e a c t | i v e → (SHIFT-LEFT ARROW) r e a c t i v e → (type t) r e a c t | i v e

The screenshot below shows the corresponding output. What’s this duplicate message at the end about? Didn’t we ask for keyup events? Yes, but it turns out the DOM does not keep track of the last value entered by the user and may raise false positives based on internal state changes.

```
|react
User wrote:
User wrote: r
User wrote: re
User wrote: rea
User wrote: reac
User wrote: react
User wrote: react
```

Notice the same “issue” appears when pasting the same text over the entire selection of the input (e.g. CTRL-A, CTRL-C, CTRL-V to exhibit the quirk).

3. Assume for a moment we take the user input and feed it to a dictionary suggest web service (as we will later on) which charges the user or application vendor for every request made to the service. Do you really want to pay twice the price to lookup “reactive” because of some weird behavior in the DOM? Likely not.

So how would you solve the issue in an Rx-free world? Well, you’d keep some state somewhere to keep track of the last value seen and only propagate the input through in case it differs from the previous input. All of this clutters the code significantly with things like a private field, an if-statement and additional assignment in the event handler, etc. But worse, all the logic goes in an event handler which lacks composition: at no point we have an asynchronous data source, free of duplicates, we can put our hands on (e.g. to hand it to another component in the system).

In Rx, thanks to the power of composition, we get away with a single operator call that does all the comparison and state maintenance on our behalf. This operator is called `DistinctUntilChanged`:

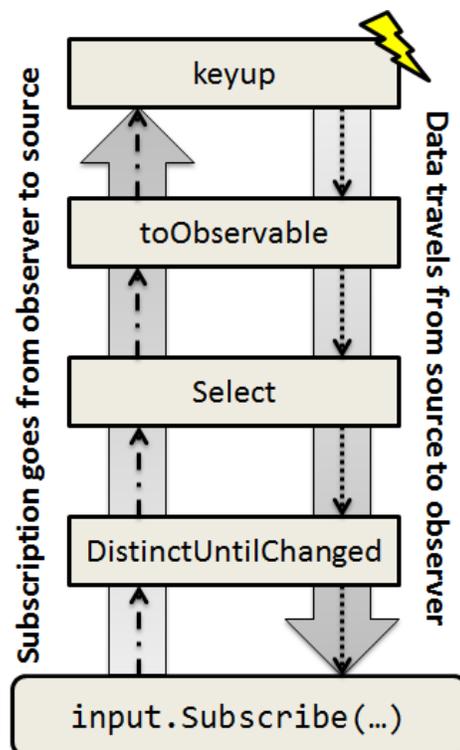
```
var input = $("#textbox").toObservable("keyup")
    .Select(function (event) { return $(event.target).val();})
    .DistinctUntilChanged();

var inputSubscription = input.Subscribe(function (text) {
    $("<p/>").text("User wrote: " + text).appendTo("#content");
});
```

With this fix in place, runs of identical values will only cause the first such value to be propagated. If the value received from the source is different the very first value or different from the previous one, it goes out at that very moment. If it’s the same as the previous value, it’s muted and attached observers don’t get to see it.



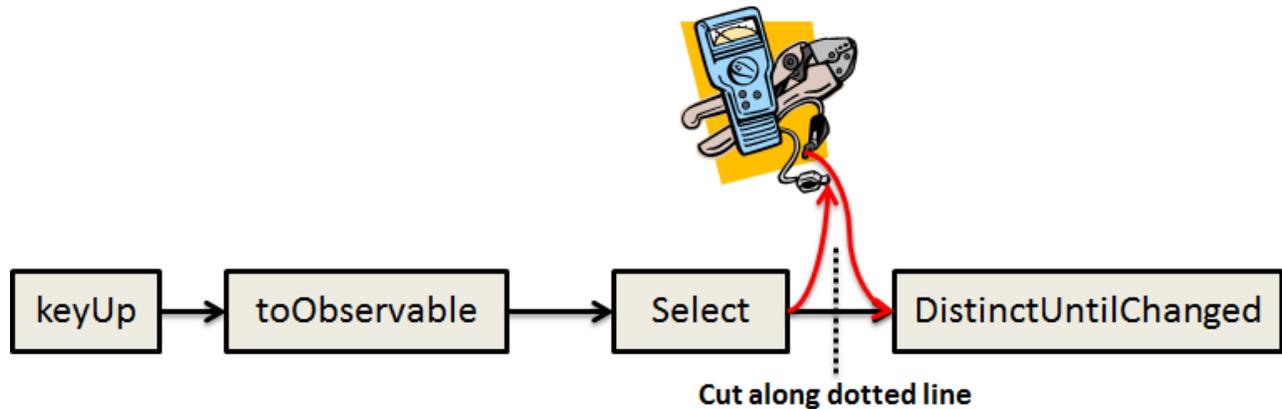
**Background:** It’s essential to understand how data flows through a “network” of interconnected observable sequences. In the sample above, there are three sequences in the mix. First, there’s `toObservable` that listens on a DOM event and emits its values to subscribed observers. Next, the `Select` operator takes care of carrying out a projection by receiving values, transforming them and sending them out. Finally, `DistinctUntilChanged` receives output from `Select`, filters out consecutive duplicates and propagates the results to its observer. The figure below illustrates how a subscription is set up and how data flows through the operators.





**Note:** Notice the casing of operators in RxJS. While typically functions in JavaScript are spelled with a lower case letter to start with, RxJS doesn't follow this convention. There are a couple of reasons for this. Some operators, namely the ones for imperative-style asynchronous computation like `If` and `Where`, conflict with keywords of the language. Another reason is consistency with Rx.NET, making porting code back and forth easier.

Each operator is a little black box that knows how to propagate subscriptions to its source sequence(s), as well as how to take the data it receives and transform it to send it along (if desired). All of this works nice till the point you see some data coming out in the observer and wonder where it comes from. To figure that out, a handy operator is called `Do`, which allows to log the data that's flowing through a "wire":



This is the Rx form of "printf debugging". The `Do` operator has several overloads that are similar to `Subscribe`'s. Either you give it an observer to monitor the data that's been propagated down through the operator or you can give it a set of handlers for `OnNext`, `OnError` and `OnCompleted`.



**Note:** The `Do` operator is also a great way to learn Rx. Simply observe the flow of data using it, for example by logging notifications to a paragraph element in HTML. Due to the absence of LINQ syntax in JavaScript, it's very easy to insert a `Do` operator between two consecutive operator applications, which are realized using chains of method calls as we've seen above. In C# with Rx.NET, use of `Do` can be a little more intrusive when query expression keywords are used since one needs to switch back to (extension) method calling syntax in order to wire up a `Do` operator (e.g. between "where" and "select"). This said, the author would love to see LINQ-alike syntax in languages other than C# and Visual Basic.

Below is a sample of the operator's use to see `DistinctUntilChanged` filtering out the duplicate values it receives:

```

var input = $("#textbox").toObservable("keyup")
    .Select(function (event) { return $(event.target).val(); })
    .Do(function(text) {
        $("<p/>").text("Before DistinctUntilChanged: " + text).appendTo("#content");
    })
    .DistinctUntilChanged();

var inputSubscription = input.Subscribe(function (text) {
    $("<p/>").text("User wrote: " + text).appendTo("#content");
});
    
```

Before the user's input is sent on to `DistinctUntilChanged`, we log it to the content element. The final result received by `Subscribe` – having flown through the `DistinctUntilChanged` operator – is also logged to the same element. Later we'll feed this data into web service calls.

With this in place, the output looks as follows. Here we produced the quirky duplicate input a few times and yet the "User wrote" message only appears for the first such input.

```

react
Before DistinctUntilChanged: r
User wrote: r
Before DistinctUntilChanged: re
User wrote: re
Before DistinctUntilChanged: rea
User wrote: rea
Before DistinctUntilChanged: reac
User wrote: reac
Before DistinctUntilChanged: react
User wrote: react
Before DistinctUntilChanged: react
Before DistinctUntilChanged: react

```

- Back to our running sample, there's yet another problem with the user's input. Since we'll ultimately feed it to a web service (which may be, as we said before, "pay for play" on a per-request basis), it's unlikely we want to send requests for every substring the user wrote while entering input. Stated otherwise, we need to protect the web service against fast typists.

Rx has an operator that can be used to "calm down" an observable sequence, called Throttle:

```

var input = $("#textbox").toObservable("keyup")
    .Select(function (event) { return $(event.target).val(); })
    .Throttle(1000)
    .DistinctUntilChanged();

var inputSubscription = input.Subscribe(function (text) {
    $("").text("User wrote: " + text).appendTo("#content");
});

```

The way this works is a timer is used to let an incoming message age for the specified duration, after which it can be propagated further on. If during this timeframe another message comes in, the original message gets dropped on the floor and substituted for the new one that effectively also resets the timer. For our sample, if the user types "reactive" without hiccups (i.e. no two consecutive changes are further apart than 1 second), no intermediate substrings will be propagated. When the user stops typing (after hitting 'e', causing the last changed event higher up), it takes one second before the input "reactive" is propagated down. Later one we'll feed this entire sequence to a web service which now cannot be called excessively due to a typist gone loose.

To illustrate the operator's effect, let's use the Do operator in conjunction with two specialized projection operators called Timestamp and RemoveTimestamp. The former takes an Observable and turns it into an Observable which yields an object with a Value and Timestamp property, where the latter does the opposite. Those operators simply add or remove a timestamp at the point a message is received. This allows us to visualize timing information:

```

var input = $("#textbox").toObservable("keyup")
    .Select(function (event) { return $(event.target).val(); })

```

```

.Timestamp()
.Do(function(inp) {
    var text = "I: " + inp.Timestamp + "-" + inp.Value;
    $("<p/>").text(text).appendTo("#content");
})
.RemoveTimestamp()
.Throttle(1000)
.Timestamp()
.Do(function(inp) {
    var text = "T: " + inp.Timestamp + "-" + inp.Value;
    $("<p/>").text(text).appendTo("#content");
})
.RemoveTimestamp()

.DistinctUntilChanged();

```



**Note:** We need to remove and reapply the timestamp around the Throttle operator call. If we wouldn't do so, Throttle would simply propagate the original timestamped value. This technique allows us to see the real delta between entering Throttle and leaving it, which should be about 1 second (give or take a few milliseconds). As we've used the same three operators twice, a good exercise is to extract the pattern into a specialized operator. Creating your own operators shouldn't be hard as you can see:

```

Rx.Observable.prototype.logTimestampedValues = function(onNext) {
    return this.Timestamp().Do(onNext).RemoveTimestamp();
};

```

Below is the output for the sample of typing "reactive" with a mild hiccup after "re" and after "reac", both of which were in the sub-second range, hence not causing propagation to beyond the Throttle operator. However, when the user stopped typing it took 996ms before the "reactive" string was observed in the Do operator after the Throttle operator (timers and timing values are subject to some deviation as usual).

```

react

```

```

I: 1278970589276-r

```

```

I: 1278970589482-re

```

```

I: 1278970589570-rea

```

```

I: 1278970589842-reac

```

```

T: 1278970590851-reac

```

```

User wrote: reac

```



**Note:** It's strongly encouraged to brainstorm for a moment how you'd write a Throttle operator on classic DOM events taking all complexities of timers, subscriptions, resource management, etc. into account. One of the core properties of Rx is that it allows reusable operators to be written, operating on a wide range of asynchronous data sources. This improves signal-to-noise ratio of user code significantly. In this particular sample, just two operators had to be added in order to tame the input sequence both for its data and for its timing behavior.

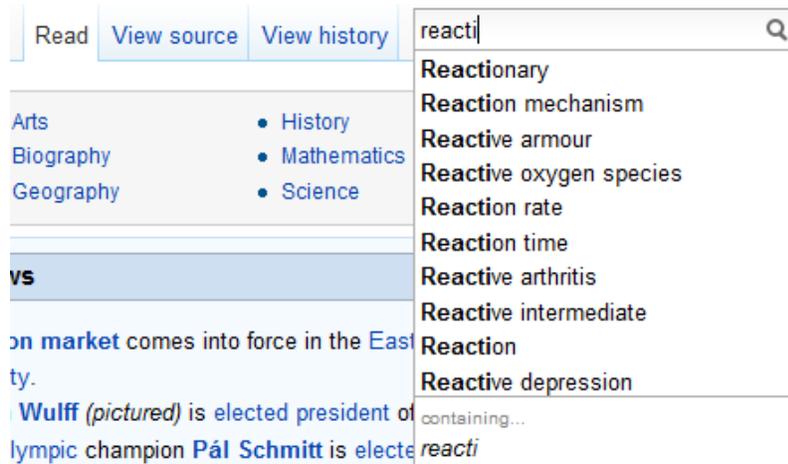
**Conclusion:** Thanks to the first-class nature of observable sequences we were able to apply operators to the input element data sources to tame it. We learned how to filter out consecutive duplicate values and how to calm down an event stream using the Throttle operator. We also introduced debugging techniques using Do and Timestamp in this exercise. Manned with a well-behaved asynchronous input sequence from an input element, we're now ready to walk up to the dictionary suggest web service, ask for word suggestions and present them to the user. It goes without saying that Rx will once more be the protagonist in this composition play. But before we do so, let's talk about synchronization.

## Exercise 6 – Bridging the asynchronous method pattern with Rx

**Objective:** In exercise 3, we learned how to bring DOM events to Rx by means of the toObservable jQuery operator. Other asynchronous data sources exist in JavaScript, as testified by the plethora of asynchronous method patterns such as AJAX APIs. We'll now explore how to expose such asynchronous data sources as observables.

1. In our running sample, we're building up a simple Wikipedia autocomplete application. Upon the user entering a search term, the application will fire off a call to a web service to get word suggestions back from Wikipedia. Since we don't want to block the UI, we'll want to keep the communication with the dictionary service asynchronous too.

For this example, we'll be using the English version of Wikipedia (<http://en.wikipedia.org/>) and the associated MediaWiki API (<http://en.wikipedia.org/w/api.php>).



For our experiment we will use the opensearch API which allows us to send search terms and retrieve the results in a JSON object array. To test what the data will look like, we can enter the following URL into our browser (<http://en.wikipedia.org/w/api.php?action=opensearch&search=react&format=json>) which will search for the term react and return the results in JSON format. The data should look as follows:

```
1  [
2    "react",
3    [
4      "Reactionary",
5      "Reaction mechanism",
6      "Reactive armour",
7      "Reactive oxygen species",
8      "Reactant",
9      "Reaction rate",
10     "REACT",
11     "Reaction time",
12     "Reactive arthritis",
13     "Reactive intermediate"
14   ]
15 ]
```

This response basically consists of an array of string values, corresponding to what the user wrote. Ultimately we want to wire up the request to the web service using the input element as the source. Before we go there, let's go through the motions of exposing the web service as an observable collection.

- Keeping our existing code with the input element, we'll first focus on bridging with the webAPI. To perform those experiments, comment out your current code to have a clean document.ready area. We'll focus on using the jQuery AJAX API and in particular, the \$.ajax function. Below is an excerpt of the samples in the ajax function's documentation, which can be found at <http://api.jquery.com/jQuery.ajax>:

```
$.ajax({
  type: "POST",
  url: "some.php",
  data: "name=John&location=Boston",
  success: function(msg){
    alert( "Data Saved: " + msg );
  }
});
```

In this sample, a POST request will be made to the given URL, passing it some data. Upon success, the specified function will be called passing in the received data in the msg parameter. Besides a success function, an error handler can be specified too.



**Note:** Looking at this API, the reader should immediately see the asynchronous nature of AJAX reflected in the function's parameters. While the call is being made, the ajax function doesn't block you. However, when data becomes available, it's passed to the *continuation* function. One should also start to see the relationship with Rx, where those continuation functions are wrapped in the concept of an observer.

Since the request to the opensearch service goes cross-domain, we'll need to utilize the JSONP data type on the ajax function. JSONP stands for "JSON with padding" and injects a piece of script into the HTML DOM used to evaluate the returned JSON data. More information on JSONP can be found on the jQuery website. All we'll have to do here is to specify a dataType parameter on the call to the ajax function, as shown in the next step.

- Now that we have a basic understanding of the ajax function, let's go through how we would call the API passing in a search term of react. On a successful call to the API, we will empty an unordered list and then add line items for each entry in the resulting JSON array. If there is a failure, we can then handle that with the error function which passes us the XMLHttpRequest, the text status and the error thrown. In this instance, we'll simply put the error into its own paragraph.

```
$.ajax(
  { url: "http://en.wikipedia.org/w/api.php",
    dataType: "jsonp",
    data: { action: "opensearch",
            search: "react",
            format: "json"
          },
    success: function (data, textStatus, xhr) {
      $("#results").empty();
      $.each(data[1], function (_, result) {
        $("#results").append("<li>" + result + "</li>");
      });
    },
    error: function (xhr, textStatus, errorThrown) {
      $("#error").text(errorThrown);
    }
  }
);
```

To make this work, add the following two elements to the page:

```
<ul id="results" />
```

```
<p id="error" />
```

For completeness, here are the results you should expect to get back:

- Reactionary
- Reaction mechanism
- Reactive armour
- Reactive oxygen species
- Reactant
- Reaction rate
- REACT
- Reaction time
- Reactive arthritis
- Reactive intermediate

4. Converting the above fragment to Rx isn't very hard using the Rx provided `$.ajaxAsObservable` function. This function acts as a tiny wrapper around `$.ajax` function using success and error functions that call into the observer's `OnNext` and `OnCompleted` for the success case, and `OnError` for the failure case:

```
$.ajaxAsObservable(  
  { url: "http://en.wikipedia.org/w/api.php",  
    dataType: "jsonp",  
    data: { action: "opensearch",  
          search: "react",  
          format: "json"  
        }  
  }  
);
```

Now we can use the `Subscribe` function to receive data from the web service call for search term "react". This result will be presented as a JSON array, just like we saw in the manual call in step 1. Since we don't want to hardcode the search term, we'll wrap the above in a function to parameterize input:

```
function searchWikipedia(term) {  
  return $.ajaxAsObservable(  
    { url: "http://en.wikipedia.org/w/api.php",  
      dataType: "jsonp",  
      data: { action: "opensearch",  
            search: term,  
            format: "json"  
          }  
    }  
  )  
  .Select(function (d) { return d.data[1]; });  
}
```

Using the `Select` operator we extract the answers from the result. To see this bit of array traversal, have a look back at the figure in step 1 to see where the string array with results is sitting.

5. With this function in place, we can now substitute the code from step 3 with the following Rx-based code. This should continue to produce the same results as shown in step 3:

```
var searchObservable = searchWikipedia("react");  
var searchSubscription = searchObservable.Subscribe(  
  function (results) {  
    $("#results").empty();  
    $.each(results, function (_, result) {  
      $("#results").append("<li>" + result + "</li>");  
    });  
  });
```

6. Since web services can easily fail, we should say a word or two on error handling. We don't really care about the specifics of possible errors but it should be common wisdom that in the world of distributed and asynchronous programming errors are not that exceptional. Rx is particularly good at dealing with errors due to the separate observer's `OnError` channel to signal those. If we were to change our sample as shown below, the error would be handled by the `OnError` function that's part of the observer:

```
var searchObservable = searchWikipedia("react");

var searchSubscription = searchObservable.Subscribe(
    function (results) {
        $("#results").empty();
        $.each(results, function (_, result) {
            $("#results").append("<li>" + result + "</li>");
        });
    },
    function (exn) {
        $("#error").text(error);
    }
);
```



**Note:** Rx has exception handling operators such as `Catch`, `Finally`, `OnErrorResumeNext` and `Retry` which allow taking a compositional approach to error handling. We won't elaborate on the rich exception handling operators present in Rx and will keep things simple by using an `OnError` handler passed to `Subscribe`.

7. One general issue with distributed programming we should call out is the potential for out-of-order arrival of responses. In the next exercise, we'll compose the user input from the bridged HTML DOM input element with web service calls, so multiple requests may be running at the same time.

For example, if the user types "react", waits one second (for `Throttle` to forward the string to its observers), then proceeds with typing "reactive" and waits another second, both web service calls will be in flight. The response to the second call may arrive before the call to the first one does. This is not too far-fetched even for this simple sample: as there'll be more words starting with "react" than with "reactive", the first request will be more network-intensive than the second one.

We can mimic this situation quite easily by starting a couple of web service requests for "incremental strings" and observe the order answers come back in:

```
var input = "reactive";

var makeRequest = function (len) {
    var req = input.substring(0, len);
    searchWikipedia(req).Subscribe(
        function (results) {
            $("#results").append("<li>" + req + " --> " + results.length + "</li>");
        },
        function (exn) {
            $("#error").text(error);
        }
    );
};

for (var len = 3; len <= input.length; len++) {
    makeRequest(len);
}
```

If you run the fragment above a couple of times, you should be (un)lucky enough to hit an out-of-order arrival situation. While requests for "rea", "react", "react", "react", "reactiv" and "reactive" are started in that order, results may come back in a different order as shown below:

- `rea --> 10`
- `reac --> 10`
- `reactive --> 10`
- `react --> 10`
- `reacti --> 10`
- `reactiv --> 10`

In the next exercise, we'll learn how to avoid this bad thing from happening.

**Conclusion:** Wrapping the omnipresent but hard to use AJAX functions in an observable sequence is easy with the `ajaxAsObservable` function provided by Rx. Specifying pretty much the same parameters as for the jQuery `ajax` function, it returns an observable sequence object that triggers an AJAX call upon subscription. Using simple function abstraction, we've created a means to represent an asynchronous web service call. Finally, we pinpointed a couple of asynchronous computing caveats such as errors and timing issues. Both of those will be tackled in the next exercise.

## Exercise 7 – SelectMany: the Zen of composition

**Objective:** With a tamed input sequence and a bridge function for an asynchronous web service call, we're ready to glue together both pieces into a single application. For every term entered by the user, we want to reach out to the service to obtain word suggestions. In doing so, we'll have to make sure to deal with potential out-of-order arrival of results as exposed in the previous exercise.

1. Before we get started with the grand composition of user input with web service calls, let's recap what we have so far. In the fragment below, the simple HTML-based UI definition for our application is shown:

```
<body>
  <input id="searchInput" size="100" type="text" />
  <ul id="results" />
  <p id="error" />
</body>
```

In order to make things more concrete, we've renamed the input control to "searchInput". Furthermore, recall we've imported the following three libraries in the `<head>` section of the HTML page:

```
<head>
  <title>Wikipedia Lookup</title>
  <script type="text/javascript" src="Scripts/jquery-1.4.1.min.js"></script>
  <script type="text/javascript" src="Scripts/rx.js"></script>
  <script type="text/javascript" src="Scripts/rx.jquery.js"></script>
```

In the previous exercise, we built a function wrapping the Wikipedia service. This code is defined in another `<script>` block:

```
<script type="text/javascript">
  function searchWikipedia(term) {
    return $.ajaxAsObservable(
      { url: "http://en.wikipedia.org/w/api.php",
        dataType: "json",
        data: { action: "opensearch",
              search: term,
              format: "json"
            }
      }
    )
    .Select(function (d) { return d.data[1]; });
  }
}
```

Finally, we defined a way to tame the user input using Rx, resulting in a throttled observable sequence bound to the “searchInput” element. This was hooked up in the document’s ready event from the same <script> block as shown above:

```
$(document).ready(function () {  
    var terms = $("#searchInput").toObservable("keyup")  
        .Select(function (event) { return $(event.target).val(); })  
        .Throttle(250);  
  
    // Time to compose stuff here...  
});
```

In here, we’ve omitted subscriptions to either of those sources since we’re now going to compose both.

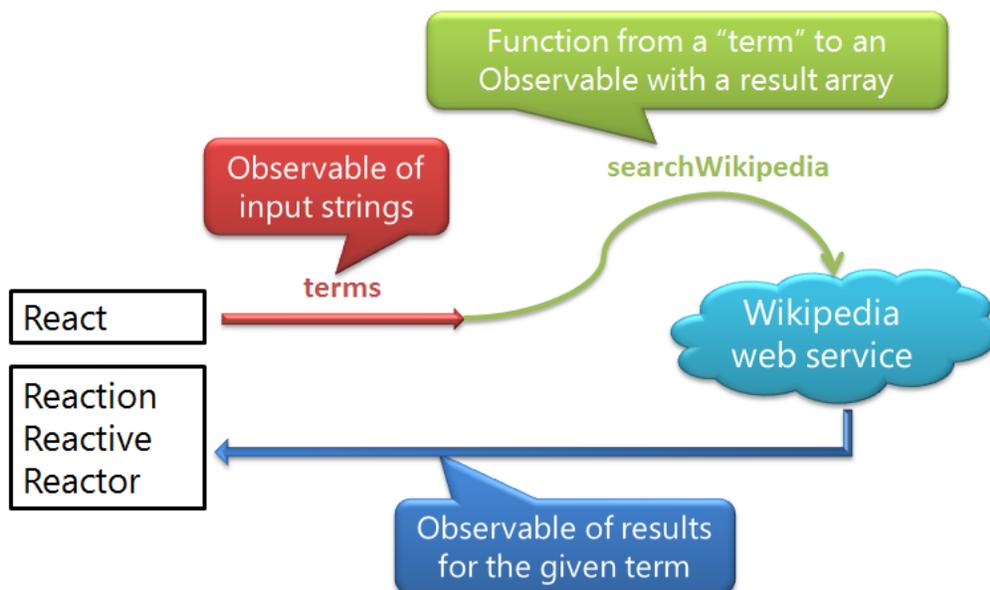
- At this point we got two things: an observable sequence of input strings and a function that takes a string and produces an observable sequence containing an array of corresponding words from Wikipedia. How do we glue those two together? The answer to this question lies in the incredibly powerful SelectMany operator. Showing how it works can best be done by using Rx for .NET where the method signature has rich types:

`Observable.SelectMany()`

```
▲ 2 of 5 ▼ (extension) IObservable<TResult> Observable.SelectMany<TSource,TResult>(this IObservable<TSource> source, Func<TSource,IObservable<TResult>> selector)  
Projects each value of an observable sequence to an observable sequence and flattens the resulting observable sequences into one observable sequence.
```

Ignore the generic nature of the types, and even the types themselves in the context of RxJS. What matters is that we got all the ingredients needed for composition fed to SelectMany. In our case, the source argument will produce terms entered by the user. Our web service wrapper function acts as the second argument, mapping those input terms onto an observable sequence containing the Wikipedia results. What the SelectMany operator can do using those inputs is *bind* them together.

Most readers will be familiar with this operator in a possibly unconscious manner. Every time you query some database traversing a relationship between tables, you’re really dealing with SelectMany. For example, assume you want to get all the suppliers across all the products in a store. Starting from a sequence of Product objects and a way to map a Product onto a Supplier (e.g. a function retrieving a Product’s SuppliedBy property), the operator can give us a flattened list of Supplier objects across all Product objects. The following figure illustrates this binding operation for our “Wikipedia complete” application:



- Let's turn SelectMany into motion now. Given the "terms" observable sequence and the mapping function, we can go ahead and add the following code to the document ready event handler:

```
var searchObservable =
    terms
    .SelectMany(function (term) { return searchWikipedia(term); });
```

For every term entered by the user, a call will be made to searchWikipedia. The results of that call will end up in the resulting "searchObservable" sequence that will be bound to the UI in the next step. This is all we need to do to compose the two asynchronous computations, the result still being asynchronous by itself as well. Herein lays the power of Rx: retaining the asynchronous nature of computations in the presence of rich composition operators (or "combinators").



**Note:** This code can be simplified a bit since searchWikipedia is a function by itself already. Hence, there's no need to wrap the function in yet another function. However, lots of people tend to write code this way since they forget about the first class citizenship of functions in languages like JavaScript. Reducing the clutter for passing a function is rooted in the principle of *eta reduction* in lambda calculus:

```
var searchObservable =
    terms
    .SelectMany(searchWikipedia);
```



**Note:** In Rx for .NET, one can use C#'s query expression syntax where the use of SelectMany is triggered by the presence of multiple from clauses. This gets translated into one of the SelectMany overloads:

```
var res = from term in terms
          from words in searchWikipedia(term)
          select words;
```



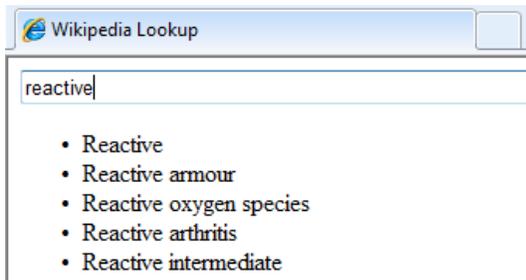
**Background:** SelectMany is one of the most powerful operators of Rx. Its power is rooted in the underlying theory of monads, leveraged by LINQ in general. While monads may sound like a scary disease, they really are a rather simple concept. In the world of monads, the SelectMany operation is called *bind*. Its purpose in life is to take an object "in the monad", apply a function to it to end up with another object "in the monad". It's basically some kind of function application on steroids, threading a concern throughout the computation.

- The next step is to bind the results to the UI. In order to receive the results, we got to subscribe the resulting observable sequence. No matter how complex the composition is we're talking about, the result is still lazy. In this case, the SelectMany use has returned an observable sequence with Wikipedia entries in arrays. All of this won't do anything till we make a call to Subscribe. From that point on, throttled user input will trigger web service calls whose results are sent to the observer passed to Subscribe:

```
searchObservable.Subscribe(
    function (results) {
        $("#results").empty();
        $.each(results, function (_, result) {
            $("#results").append("<li>" + result + "</li>");
        });
    },
    function (exn) {
        $("#error").text(error);
    }
);
```

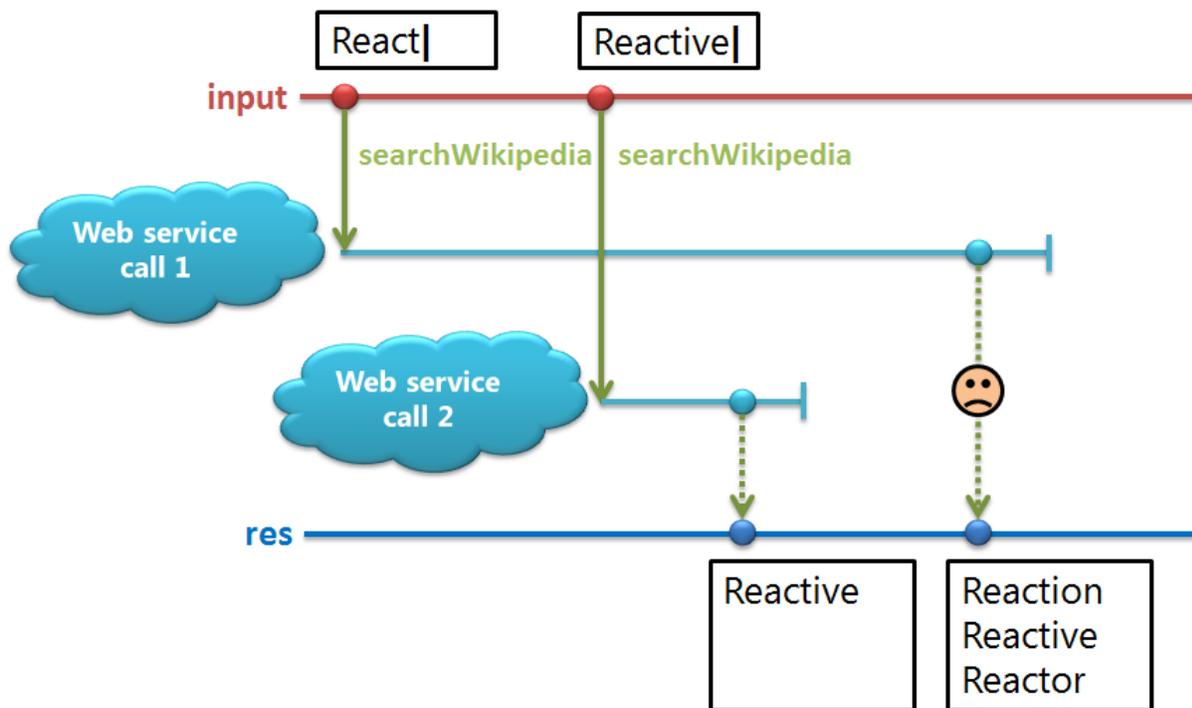
In this piece of code, we receive the words in the OnNext handler. After clearing the results bullet list, we use the jQuery each iteration function to populate the list with the values received. In case an error results from calling the service, it will be propagated to the searchObservable's OnError channel which we subscribe to as well to notify the user through the error element.

The screenshot below shows a fragment of the output:



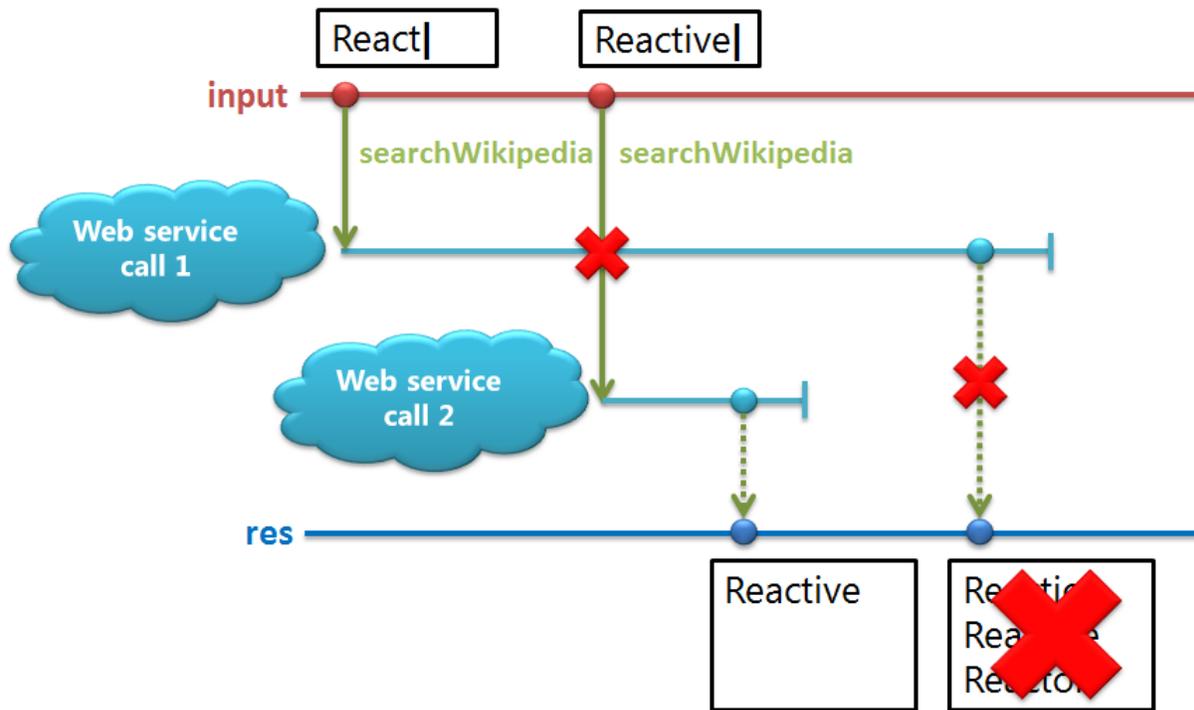
5. One problem is lurking around the corner, waiting to come and get us: out-of-order arrival. To understand why this can happen at all in the context of our composition using `SelectMany`, we need to elaborate on the working of the operator. Whenever the `SelectMany` operator receives an input on its source, it evaluates the selector function to obtain an observable sequence that will provide data for the operator's output. Essentially it flattens all of the observable sequences obtained in this manner into one flat resulting sequence.

For example, if the user types "react" and idles out for at least one second, `SelectMany` receives the string from the `Throttle` operator preceding it. Execution of the selector function - `searchWikipedia` - causes a web service call to be started. While this call is in flight, the user may enter "reactive" which may end up triggering another web service call in a similar manner (one second throttle delay, application of the selector function). At that point, two parallel calls are happening, which could provide results out-of-order compared to the input. The figure below illustrates the issue that can arise due to this behavior:



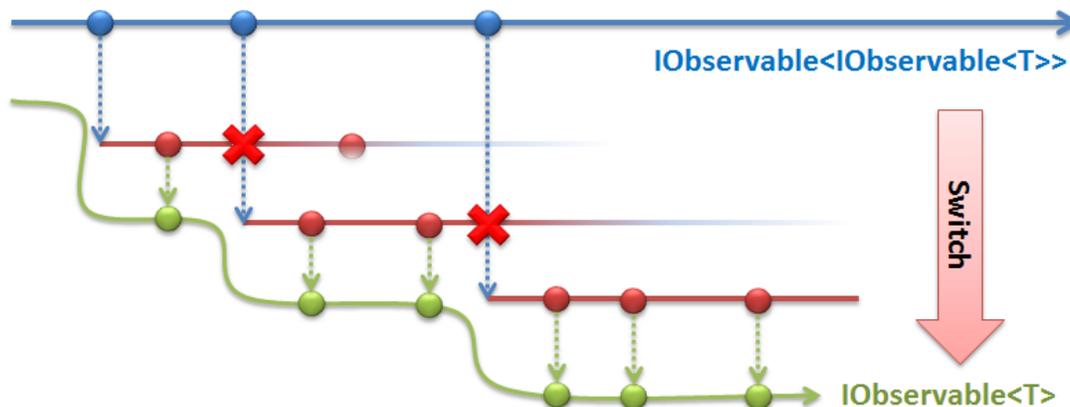
Since throttling adds a 250 millisecond delay, you have to be a bit (un)lucky to hit this issue. However, if it remains unfixed, it'd likely come and get you the first time you demo the application to your boss. Since we don't want this to happen to us, we need to cancel out existing web service requests as soon as the user enters a new term, indicating there's no further interest in the previous search term.

What we want to achieve is illustrated in the figure on the next page. The essence of the fix is "crossing out" or "muting" in-flight requests when a new one is received. This is illustrated as the red cross on the line for the first web service call.



**Note:** At some point in the design of Rx, there was a special SelectMany operator with cancellation behavior: whenever an object was received on the source, the previous (if any) inner observable was unsubscribed before calling the selector function to get a new inner observable. As you may expect, this led to numerous debates which of the two behaviors was desired. Having two different operator flavors for SelectMany was not a good thing for various reasons. For one thing, only one flavor could be tied to the C# and Visual Basic LINQ syntax. The ultimate solution was to decouple the notion of cancellation from the concept of “monadic bind”. As a result, new cancellation operators arose, which do have their use in a lot of other scenarios too. A win-win situation!

- The realization of this cancellation behavior can be achieved using a single operator called Switch whose behavior is illustrated in the diagram below:



Given a sequence of sequences (yes, that’s not a typo) it hops from one sequence to another as they come in. When the topmost observable “outer” sequence produces a new observable “inner” sequence, an existing inner sequence subscription is disposed and the newly received sequence is subscribed to. Results produced by the current inner sequence are propagated to the Switch operator’s output.

Use of this operator in our scenario proceeds as follows. Instead of using SelectMany, we’ll map user input on web service requests using a simple Select operator use. Since every web service request returns an observable sequence with an array of results as the payload the result of this projection is an “observable of observables”. If

we were to have types, we'd have an `IObservable<IObservable<string[]>>` here. Applying Switch over this nested sequence causes the behavior described above. For every request submitted by the user, the web service is contacted. If a new request is made, Switch cancels out the existing one's subscription and hops to the new one:

```
var searchObservable =
    terms
    .Select(searchWikipedia);
    .Switch();
```

With this fix in place, out-of-order arrival should not be able to come and get us.



**Note:** Alternative fixes to this typical asynchronous programming problem exist. In the HOL for the .NET version of Rx, we present the use of the TakeUntil operator that acts as a valve on an observable sequence. With this different approach, we take results from the current web service call *until* another user request is made.

**Conclusion:** Composition of multiple asynchronous data sources is one of the main strengths of Rx. In this exercise we looked at the SelectMany operator that allows “binding” one source to another. More specially, we turned user entries of terms (originating from DOM events) – into asynchronous web service calls (brought to Rx using ajaxAsObservable). To deal with errors and out-of-order arrival only a minimal portion of the code had to be tweaked. While we didn't mention operators other than SelectMany and Switch that deal with multiple sources, suffice it to say that a whole bunch of those exist awaiting your further exploration.

## Exercise 8 – Testability and mocking made easy

**Objective:** Testing asynchronous code is a hard problem. Representing asynchronous data sources as first-class objects implementing a common “interface”, Rx is in a unique position to help to simplify this task as well. We'll learn how easy it is to mock observable sequences to create solid tests.

1. Each observable sequence can be regarded as a testable unit. In our dictionary suggest sample, two essential sequences are being used. One is the user input sequence; the other is its composition with the web service. Since all of those are first-class objects, we can simply swap them out for a sequence “mock”.

One particularly useful operator in RxJS is FromArray which takes a JavaScript array. It's a pull-to-push adapter that enumerates (pulls) the given array and exposes it as an observable sequence that notifies (pushes) its observers of the array's elements. As an example, replace the input sequence for an array-based mock observable as shown below:

```
//var terms = $("#searchInput").toObservable("keyup")
//    .Select(function (event) { return $(event.target).val(); })
//    .Throttle(250);
var input = Rx.Observable.FromArray(["reac", "reactive", "bing"]);
```

Now when we try to run the application, our web service will be fed “reac”, “reactive” and “bing” virtually at the same time since there's no delay between the elements in the input. If the Switch operator does its out-of-order preventing job correctly, only the results for the “bing” request should appear on the screen.

It's a worthy experiment to take out the Switch operator from the previous exercise and observe responses coming back. With a bit of luck, you'll see the issue cropping up due to the multiple requests being fired right after one another.

2. Thanks to the various time-based operators, we can provide more realistic input. One thing we could do to mock user input in a more faithful way is to use time-based operators to mimic typing. GenerateWithTime is a suitable candidate for our next experiment. Let's generate a sequence of incremental substrings for a given term, with random delays between them.

```

var input = "reactive";
var terms = Rx.Observable.GenerateWithTime(
    1,
    function (len) { return len <= input.length; },
    function (len) { return len + 1; },
    function (len) { return input.substring(0, len); },
    function (_) { return Math.random() * 1000; }
)
.Do(function (term) {
    $("#searchInput").val(term);
});

```

Here we use a random number generator to simulate typing speed variations (such that Throttle will sometimes let a substring through). Before we throttle the sequence, we use the side-effecting Do operator to put the substring in the input element to really simulate the user typing in a visual manner.

3. Similarly, we could mock the web service by replacing the searchWikipedia function. Obviously one will have to come up with some output to go with the input term, maybe from a local dictionary or based on some dummy output generation. Below is a sample web service mock:

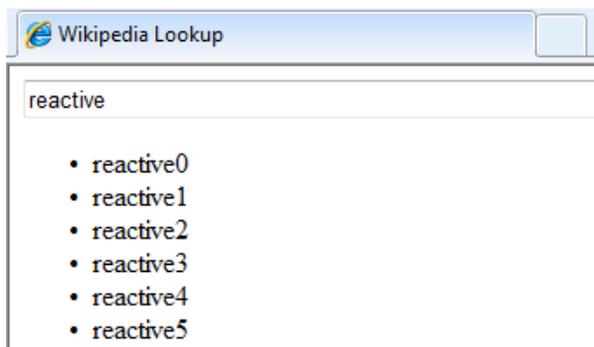
```

function searchWikipedia(term) {
    // return $.ajaxAsObservable(
    //     { url: "http://en.wikipedia.org/w/api.php",
    //       dataType: "jsonp",
    //       data: { action: "opensearch",
    //             search: term,
    //             format: "json",
    //             limit: 100
    //           }
    //     })
    //     .Select(function (d) { return d.data[1]; });
    var result = new Array();
    for (var i = 0; i < Math.random() * 50; i++)
        result[i] = term + i;

    return Rx.Observable.Return(result).Delay(Math.random() * 10000);
}

```

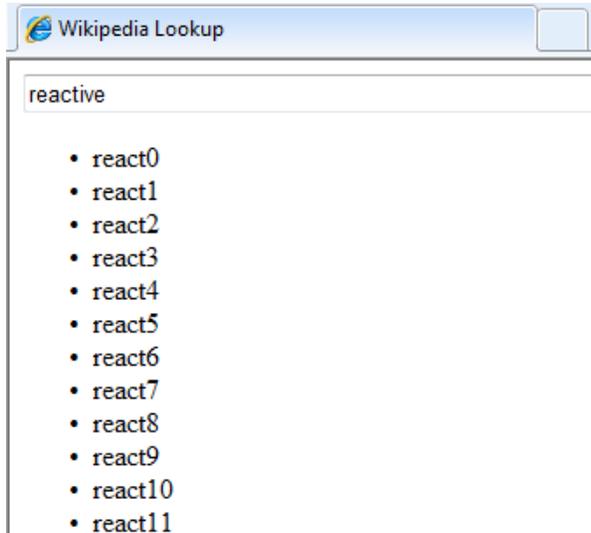
In this code, we first generate an array of 0 to 50 results simply based on the service “term” input with a number concatenated. For example, given “rea”, we may get { “rea0”, “rea1”, “rea2” } back. Since the service contract is to return an observable of such an array, we use Rx.Observable.Return to create a single-element observable sequence with the generated array. Finally, we use the Delay operator that’s defined for observable sequences to add a random 1-to-10 second delay in sending back the response.



Running the sample again without the out-of-order prevention, it should be plain easy to hit the out-of-order arrival issue we have described in much detail before:

```
var searchObservable =
  terms
  .SelectMany(searchWikipedia);
  // .Switch();
```

The result is indeed devastating. Clearly, the output shown below isn't right. When seeing the simulated typing fly by, you may see results for "reactive" coming back and then getting overwritten by the responses of some prior call.



Assume such an issue has been uncovered in your code; it's incredibly simple to create a test case for it using mocks like those.

**Conclusion:** The first class object nature of observable sequences makes it easy to replace them, contrast to various asynchronous technologies like DOM events. This allows for smooth testing of asynchronous programs using mock input sequences, e.g. based on arrays turned into observable sequences with FromArray.