QCon

INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

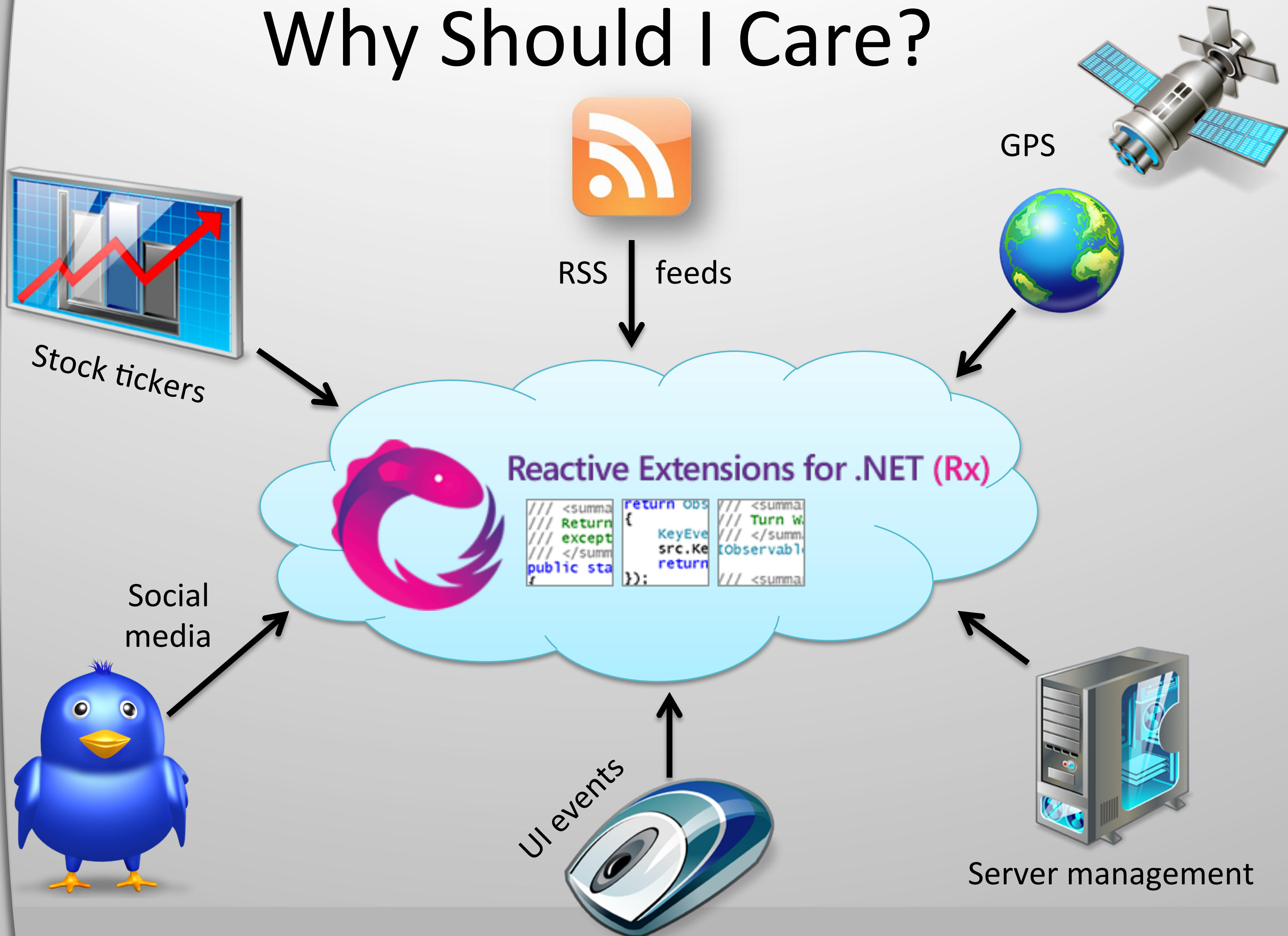
Rx: Curing your Asynchronous Programming Blues

Bart J.F. De Smet

Microsoft Corporation

bartde@microsoft.com

Why Should I Care?



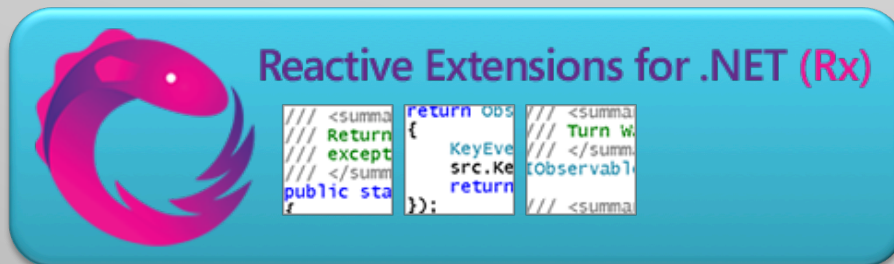
Mission Statement

Too hard today!

$(f \circ g)(x) = f(g(x))$

Rx is a library for composing asynchronous and event-based programs using observable sequences

Queries? **LINQ?**



- .NET 3.5 SP1 and 4.0
- Silverlight 3 and 4
- XNA 3.1 for XBOX and Zune
- Windows Phone 7
- JavaScript (RxJS)

Download at **MSDN Data Developer Center** or use **NuGet**

Essential Interfaces

Enumerables – a pull-based world

```
interface IEnumerable<out T>
{
    IEnumerator<T> GetEnumerator();
}

interface IEnumerator<out T> : IDisposable
{
    bool    MoveNext();
    T       Current { get; }
    void    Reset();
}
```

C# 4.0 **covariance**

You could get ***stuck***

Essential Interfaces

Enumerables – a pull-based world

Synchronous snow ~~(Not Responding)~~ **(Waiting to move next)**

Do work!

snow.exe

snow.exe is ~~not responding~~ **moving on**

If you close the program, you might lose information.

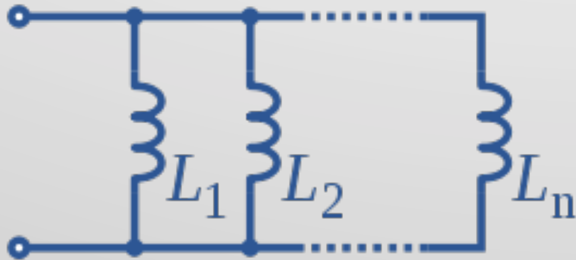
→ Close the program

→ Wait for the program to respond

Mathematical Duality

Because the Dutch are (said to be) cheap

– **Electricity**: inductor and capacitor



– **Logic**: De Morgan's Law

$\neg(A \wedge B) \equiv \neg A \vee \neg B$

$\neg(A \vee B) \equiv \neg A \wedge \neg B$

– **Programming**?

What's the dual of IEnumerable?

The recipe to dualization

http://en.wikipedia.org/wiki/**Dual_(category_theory)**

Formal definition [edit]

We define the elementary language of category theory as the two-sorted first order language with objects and morphisms as distinct sorts, together with the relations of an object being the source or target of a morphism and a symbol for composing two morphisms.

Let σ be any statement in this language. We form the dual σ^{op} as follows:

1. Interchange each occurrence of "source" in σ with "target".
2. Interchange the order of composing morphisms. That is, replace each occurrence of $g \circ f$ with $f \circ g$

Informally, these conditions state that the dual of a statement is formed by reversing arrows and compositions.

Duality is the observation that σ is true for some category *C* if and only if σ^{op} is true for C^{op}.

Making a U-turn in synchrony

Reversing arrows…
Input becomes output and vice versa

What's the dual of IEnumerable?

Step 0 – Simplicity and honesty

```
interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}

interface IEnumerator<out T> : IDisposable
{
  bool   MoveNext();
  T      Current { get; }
  void   Reset();
}
```

Properties are ***methods***

What's the dual of IEnumerable?

Step 0 – Simplicity and honesty

```
interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}

interface IEnumerator<out T> : IDisposable
{
  bool   MoveNext();
  T      GetCurrent();
  void   Reset();
}
```

~~void Reset();~~ — A ***historical*** mishap

What's the dual of IEnumerable?

Step 0 – Simplicity and honesty

```
interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}

interface IEnumerator<out T> : IDisposable
{
  bool   MoveNext();
  T      GetCurrent();
}
```

No checked exceptions in .NET / C#

What’s the dual of IEnumerable?

Step 0 – Simplicity and honesty

```
interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}

interface IEnumerator<out T> : IDisposable
{
  bool  MoveNext() throws Exception;
  T     GetCurrent();
}
```

What's the dual of IEnumerable?

Step 1 – Input versus output

```
interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}

interface IEnumerator<out T> : IDisposable
{
  bool   MoveNext() throws Exception;
  T      GetCurrent();
}
```

This is an ***output too***!

What's the dual of IEnumerable?

Step 1 – Input versus output

```
interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}

interface IEnumerator<out T> : IDisposable
{
  (bool | Exception) MoveNext();
  T      GetCurrent();
}
```

Really ***only two values***

Discriminated union type ("or")

What's the dual of IEnumerable?

Step 1 – Input versus output

```
interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}

interface IEnumerator<out T> : IDisposable
{
  (true | false | Exception) MoveNext();
  T      GetCurrent();
}
```

Got ***true***?
Really got ***T***!

What's the dual of IEnumerable?

Step 1 – Input versus output

```
interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}

interface IEnumerator<out T> : IDisposable
{
  (T | false | Exception) MoveNext();
}
```

Got ***false***?
Really got ***void***!

What's the dual of IEnumerable?

Step 1 – Input versus output

```
interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}
```

Every enumerator *is* disposable

```
interface IEnumerator<out T> : IDisposable
{
  (T | void | Exception) MoveNext();
}
```

What's the dual of IEnumerable?

Step 1 – Input versus output

```
interface IEnumerable<out T>
{
  (IEnumerator<T> & IDisposable) GetEnumerator();
}
```

Hypothetical syntax ("and")

```
interface IEnumerator<out T>
{
  (T | void | Exception) MoveNext();
}
```

What's the dual of IEnumerable?

Step 2 – Swapping input and output

```
interface IEnumerable<out T>
{
  (IEnumerator<T> & IDisposable) GetEnumerator();
}

interface IEnumerator<out T>
{
  (T | void | Exception) MoveNext();
}
```

Variance will flip!

Will ***rename*** too

This is really ***void***

What's the dual of IEnumerable?

Step 2 – Swapping input and output

```
interface IEnumerable<out T>
{
  (IEnumerator<T> & IDisposable) GetEnumerator();
}

interface IEnumeratorDual<in T>
{
  void   OnNext(T | void | Exception);
}
```

Can encode as ***three methods***

What's the dual of IEnumerable?

Step 2 – Swapping input and output

```
interface IEnumerable<out T>
{
  (IEnumerator<T> & IDisposable) GetEnumerator();
}

interface IEnumeratorDual<in T>
{
  void    OnNext(T value);
  void    OnCompleted();
  void    OnError(Exception exception);
}
```

Color of the bikeshed (*)

(*) Visit http://en.wikipedia.org/wiki/Color_of_the_bikeshed

What's the dual of IEnumerable?

Step 2 – Swapping input and output

```
interface IEnumerable<out T>
{
  (IEnumerator<T> & IDisposable) GetEnumerator();
}

interface IObserver<in T>
{
  void   OnNext(T value);
  void   OnCompleted();
  void   OnError(Exception exception);
}
```

Will ***leave*** this part alone

This is the ***data source***

Need to ***patch*** this one up too

What's the dual of IEnumerable?

Step 2 – Swapping input and output

```
interface IEnumerableDual<out T>
{
  IDisposable SetObserver(IObserver<T> observer);
}

interface IObserver<in T>
{
  void   OnNext(T value);
  void   OnCompleted();
  void   OnError(Exception exception);
}
```

Color of the bikeshed (*)

(*) Visit http://en.wikipedia.org/wiki/Color_of_the_bikeshed

What's the dual of IEnumerable?

Step 3 – The final result

```
interface IObservable<out T>
{
  IDisposable Subscribe(IObserver<T> observer);
}

interface IObserver<in T>
{
  void   OnNext(T value);
  void   OnCompleted();
  void   OnError(Exception exception);
}
```

Essential Interfaces

Observables – a push-based world

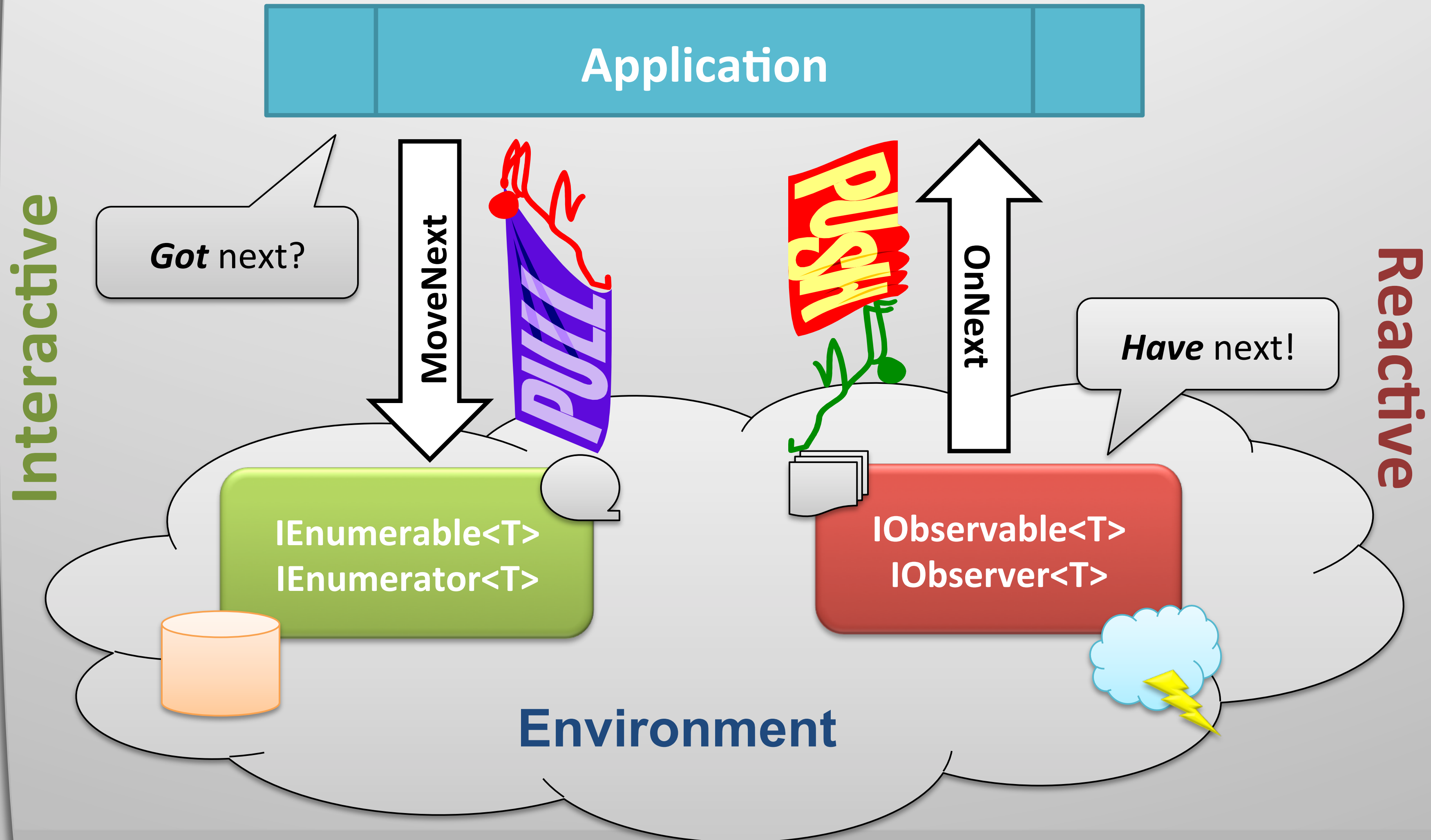
```
interface IObservable<out T>
{
  IDisposable Subscribe(IObserver<T> observer);
}

interface IObserver<in T>
{
  void    OnNext(T value);
  void    OnError(Exception ex);
  void    OnCompleted();
}
```

C# 4.0
contravariance

You could get ***flooded***

Essential Interfaces

Summary – push versus pull



Demo

Getting Your Observables

Primitive constructors

OnCompleted

Observable.Empty<int>()

new int[0]

OnNext

Observable.Return(42)

new[] { 42 }

OnError

Observable.Throw<int>(ex)

***Throwing** iterator*

Observable.Never<int>()

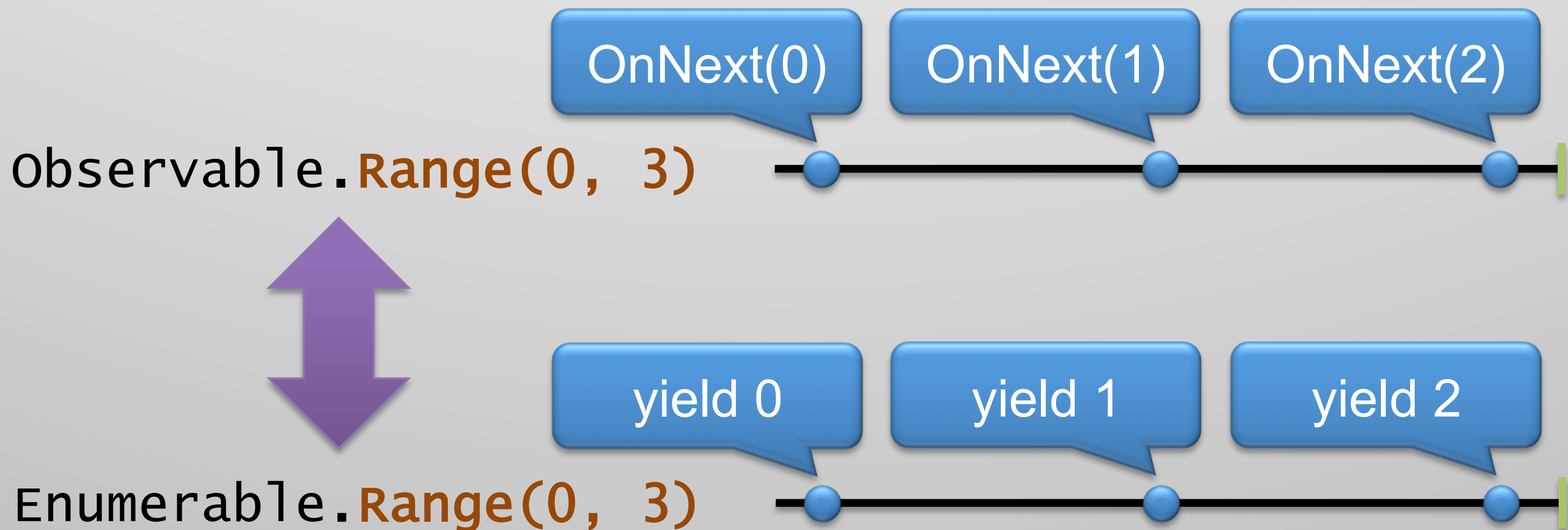
Notion of **time**

*Iterator that got **stuck***

Getting Your Observables

Observer (and enumerator) grammar

OnNext* [OnError | OnCompleted]



Getting Your Observables

Generator functions

A variant with time notion exists (***GenerateWithTime***)

```
o = Observable.Generate(
    0,
    i => i < 10,
    i => i + 1,
    i => i * i
);
```

Asynchronous

```
o.Subscribe(x => {
    Console.WriteLine(x);
});
```

Hypothetical anonymous iterator syntax in C#

```
e = new IEnumerable<int> {
    for (int i = 0;
         i < 10;
         i++)
      yield return i * i;
};
```

Synchronous

```
foreach (var x in e) {
    Console.WriteLine(x);
}
```

Getting Your Observables

Create, the most generic creation operator

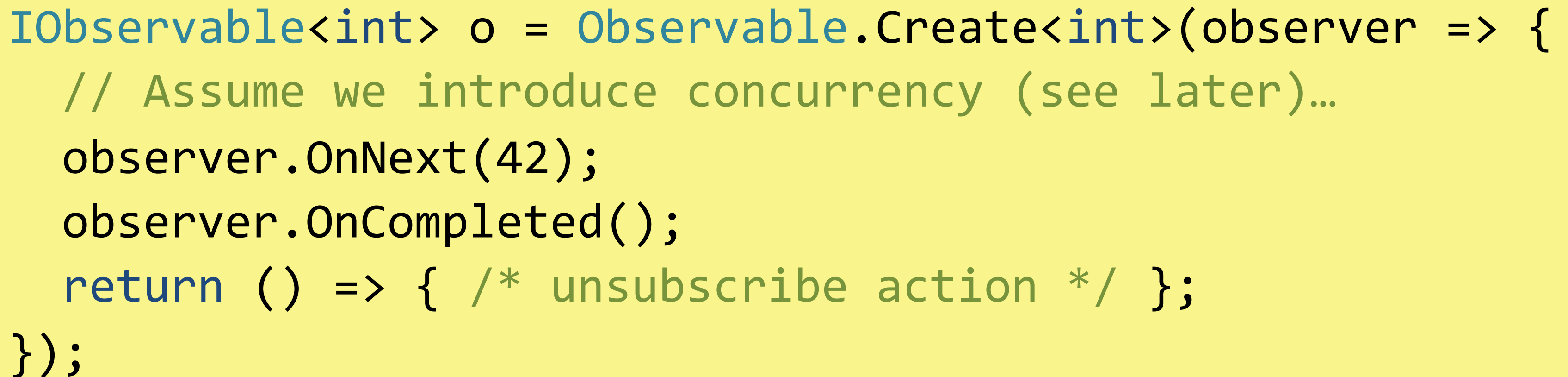
```
IObservable<int> o = Observable.Create<int>(observer => {
  // Assume we introduce concurrency (see later)…
  observer.OnNext(42);
  observer.OnCompleted();
  return () => { /* unsubscribe action */ };
});

IDisposable subscription = o.Subscribe(
  onNext:      x  => { Console.WriteLine("Next: " + x); },
  onError:     ex => { Console.WriteLine("Oops: " + ex); },
  onCompleted: () => { Console.WriteLine("Done"); }
);
```

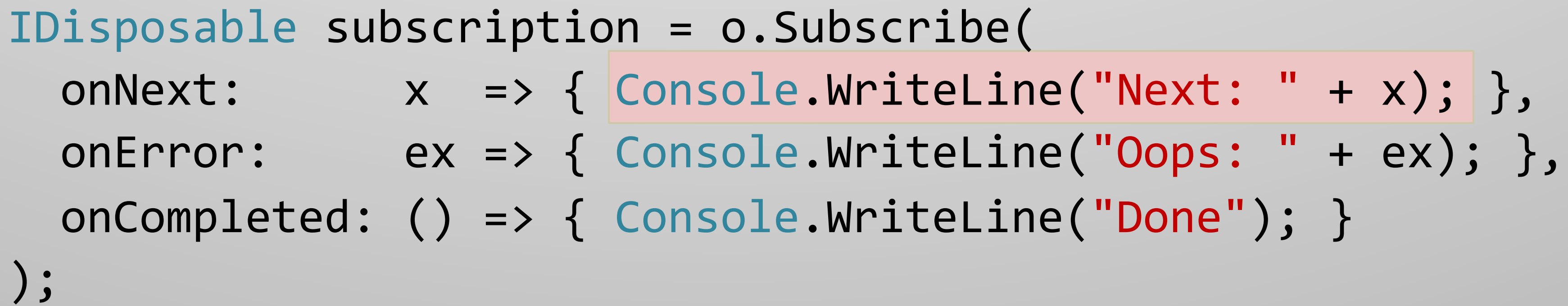
C# 4.0 named parameter syntax

*C# doesn't have **anonymous interface implementation**, so we provide various extension methods that take lambdas.*

Getting Your Observables

Create, the most generic creation operator





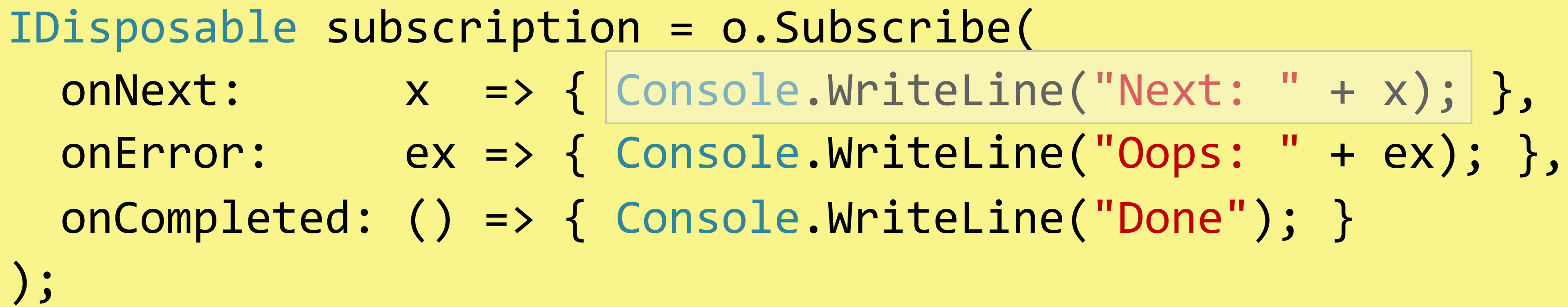

```
Thread.Sleep(30000); // Main thread is blocked…
```

F10

Getting Your Observables

Create, the most generic creation operator

```
IObservable<int> o = Observable.Create<int>(observer => {
  // Assume we introduce concurrency (see later)…
  observer.OnNext(42);
  observer.OnCompleted();
  return () => { /* unsubscribe action */ };
});
```




```
Thread.Sleep(30000); // Main thread is blocked…
```

F10

Getting Your Observables

Create, the most generic creation operator

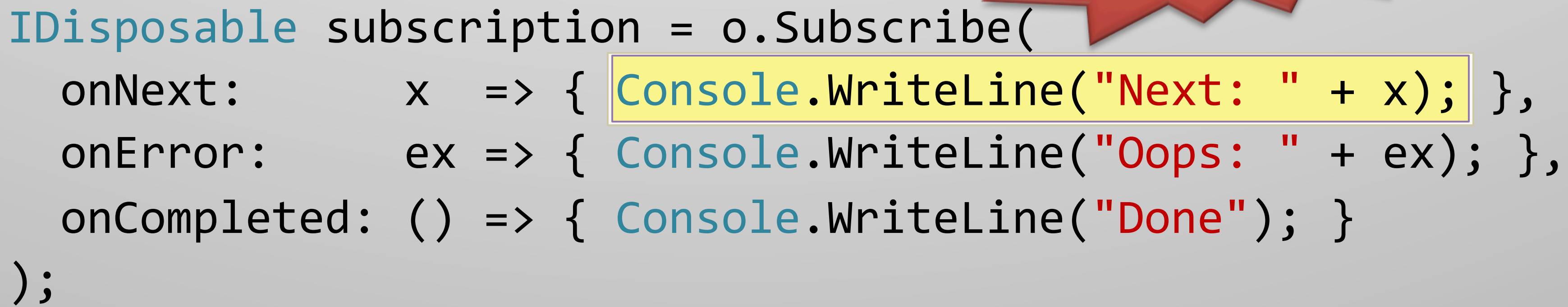
```
IObservable<int> o = Observable.Create<int>(observer => {
  // Assume we introduce concurrency (see later)…
  observer.OnNext(42);
  observer.OnCompleted();
  return () => { /* unsubscribe action */ };
});

IDisposable subscription = o.Subscribe(
  onNext:      x  => { Console.WriteLine("Next: " + x); },
  onError:     ex => { Console.WriteLine("Oops: " + ex); },
  onCompleted: () => { Console.WriteLine("Done"); }
);
```

F5

Getting Your Observables

Create, the most generic creation operator

```
IObservable<int> o = Observable.Create<int>(observer => {
  // Assume we introduce concurrency (see later)…
  observer.OnNext(42);
  observer.OnCompleted();
  return () => { /* unsubscribe action */ };
});
```



```
IDisposable subscription = o.Subscribe(
  onNext:      x  => { Console.WriteLine("Next: " + x); },
  onError:     ex => { Console.WriteLine("Oops: " + ex); },
  onCompleted: () => { Console.WriteLine("Done"); }
);
```

```
Thread.Sleep(30000); // Main thread is blocked…
```

Demo

Bridging Rx with the World

Why .NET events aren't first-class…

How to pass around?

Hidden data source

```
form1.MouseMove += (sender, args) => {

   if (args.Location.X == args.Location.Y)
      // I'd like to raise another event
};
```

Lack of composition

```
form1.MouseMove -= /* what goes here? */
```

Resource maintenance?

http://en.wikipedia.... First-class object - Wikipedi...

Wikipedia
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

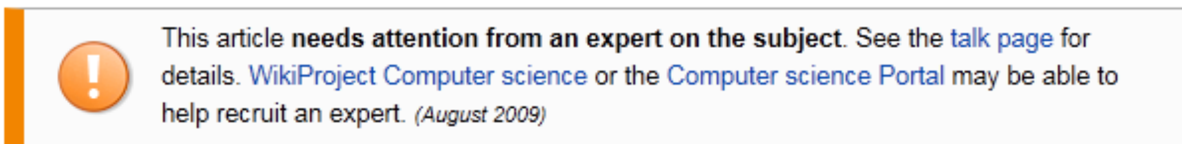
Print/export

Languages
Česky
Deutsch
Français
日本語
Polski
Русский

Article Discussion Read Edit View history Search

First-class object

From Wikipedia, the free encyclopedia


This article **needs attention from an expert on the subject**. See the talk page for details. WikiProject Computer science or the Computer science Portal may be able to help recruit an expert. *(August 2009)*

In computing, a **first-class object** (also **value**, **entity**, and **citizen**), in the context of a particular programming language, is an entity that can be passed as a parameter, returned from a subroutine, or assigned into a variable.[1] In computer science the term reification is used when referring to the process (technique, mechanism) of making something a first-class object.[2]

The term was coined by Christopher Strachey in the context of "functions as first-class citizens" in the mid-1960s.[3]

Contents [hide]

1 Definition
2 Examples
3 Notes
4 See also

Definition [edit]

An object is first-class when it:[4][5]

- can be stored in variables and data structures
- can be passed as a parameter to a subroutine
- can be returned as the result of a subroutine
- can be constructed at runtime
- has intrinsic identity (independent of any given name)

The term "object" is used loosely here, not necessarily referring to objects in object-oriented programming. The simplest scalar data types, such as integer and floating-point numbers, are nearly always first-class.

Bridging Rx with the World

...but observables sequences are!

Objects can be passed

Source of Point values

```
IObservable<Point> mouseMoves =
      Observable.FromEvent(frm, "MouseMove");
var filtered = mouseMoves
                .Where(pos => pos.X == pos.Y);
```

Can define operators

```
var subscription = filtered.Subscribe(…);
subscription.Dispose();
```

Resource maintenance!

Bridging Rx with the World

Asynchronous methods are a pain…

```
FileStream fs = File.OpenRead("data.txt");
byte[] bs = new byte[1024];
fs.BeginRead(bs, 0, bs.Length,
    new AsyncCallback(iar => {
        int bytesRead = fs.EndRead(iar);
        // Do something with bs[0..bytesRead-1]
    }),
    null
);
```

Exceptions?

Hidden data source

Really a method pair

Cancel?

Lack of composition

State?

Synchronous completion?

Bridging Rx with the World

…but observables are cuter!

```
FileStream fs = File.OpenRead("data.txt");
Func<byte[], int, int, IObservable<int>> read =

    Observable.FromAsyncPattern<byte[], int, int,
                                int>(
        fs.BeginRead, fs.EndRead);

byte[] bs = new byte[1024];
read(bs, 0, bs.Length).Subscribe(bytesRead => {
    // Do something with bs[0..bytesRead-1]
});
```

Tip: a nicer wrapper can easily be made using various **operators**

Bridging Rx with the World

The grand message

- We **don't replace** existing asynchrony:
  - .NET events have their use
  - the async method pattern is fine too
  - tasks are great at representing single-value computations
  - other sources like SSIS, PowerShell, StreamInsight, WMI, etc.
- but we…
  - **unify** those worlds
  - introduce **compositionality**
  - provide **generic operators**
- hence we…
  - **build bridges!**

Bridging Rx with the World

Terminology: hot versus cold observables

- Cold observables

```
var xs = Observable.Return(42);
```

Triggered by subscription

```
xs.Subscribe(Console.WriteLine); // Prints 42
xs.Subscribe(Console.WriteLine); // Prints 42 again
```

- Hot observables

```
var mme = Observable.FromEvent<MouseEventArgs>
                  (from, "MouseMove");
```

Mouse events going *before* subscription

```
mme.Subscribe(Console.WriteLine);
```

Demo

Composition and Querying

Concurrency and synchronization

- What does **asynchronous** mean?
  - Greek:
    - a-syn = *not with* (independent from each other)
    - chronos = *time*
  - Two or more parties work at their own pace
    - Need to **introduce concurrency**
- Notion of **IScheduler**

Parameterization of operators

```
var xs = Observable.Return(42, Scheduler.ThreadPool);
xs.Subscribe(Console.WriteLine);
```

Will run on the source's scheduler

Composition and Querying

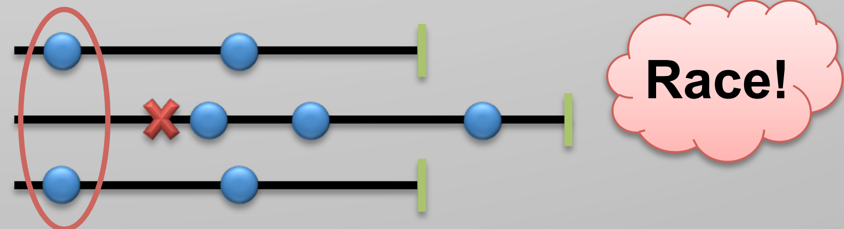
Concurrency and synchronization

- Does **duality** apply?
  - Convert between both worlds

```
// Introduces concurrency to enumerate and signal…
var xs = Enumerable.Range(0, 10).ToObservable();

// Removes concurrency by observing and yielding…
var ys = Observable.Range(0, 10).ToEnumerable();
```

- "Time-centric" **reactive** operators:

```
source1
source2
source1.Amb(source2)
```

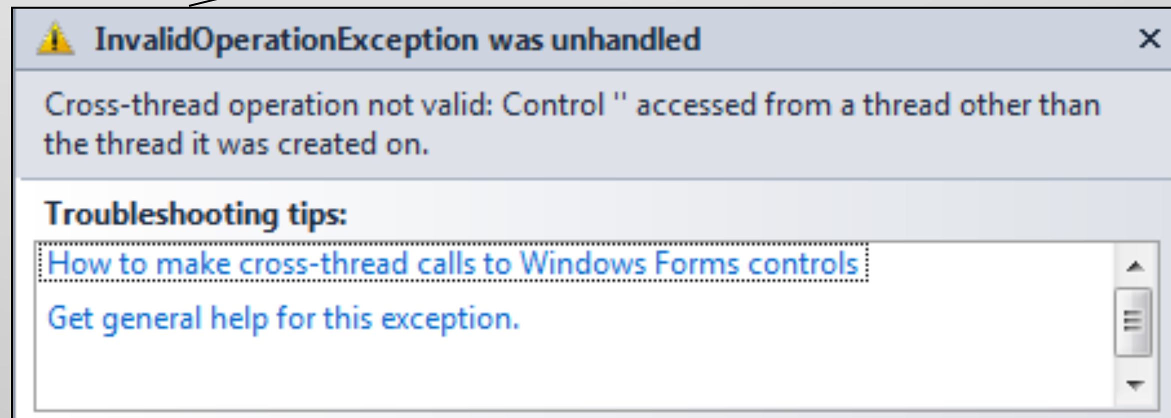


Composition and Querying

Concurrency and synchronization

- How to **synchronize**?

```
var xs = Observable.Return(42, Scheduler.ThreadPool);
xs.Subscribe(x => lbl.Text = "Answer = " + x);
```



IScheduler interface
- WPF dispatcher
- WinForms control
- SynchronizationContext

- Compositionality to the rescue!

```
xs.ObserveOn(frm)
  .Subscribe(x => lbl.Text = "Answer = " + x);
```

Composition and Querying

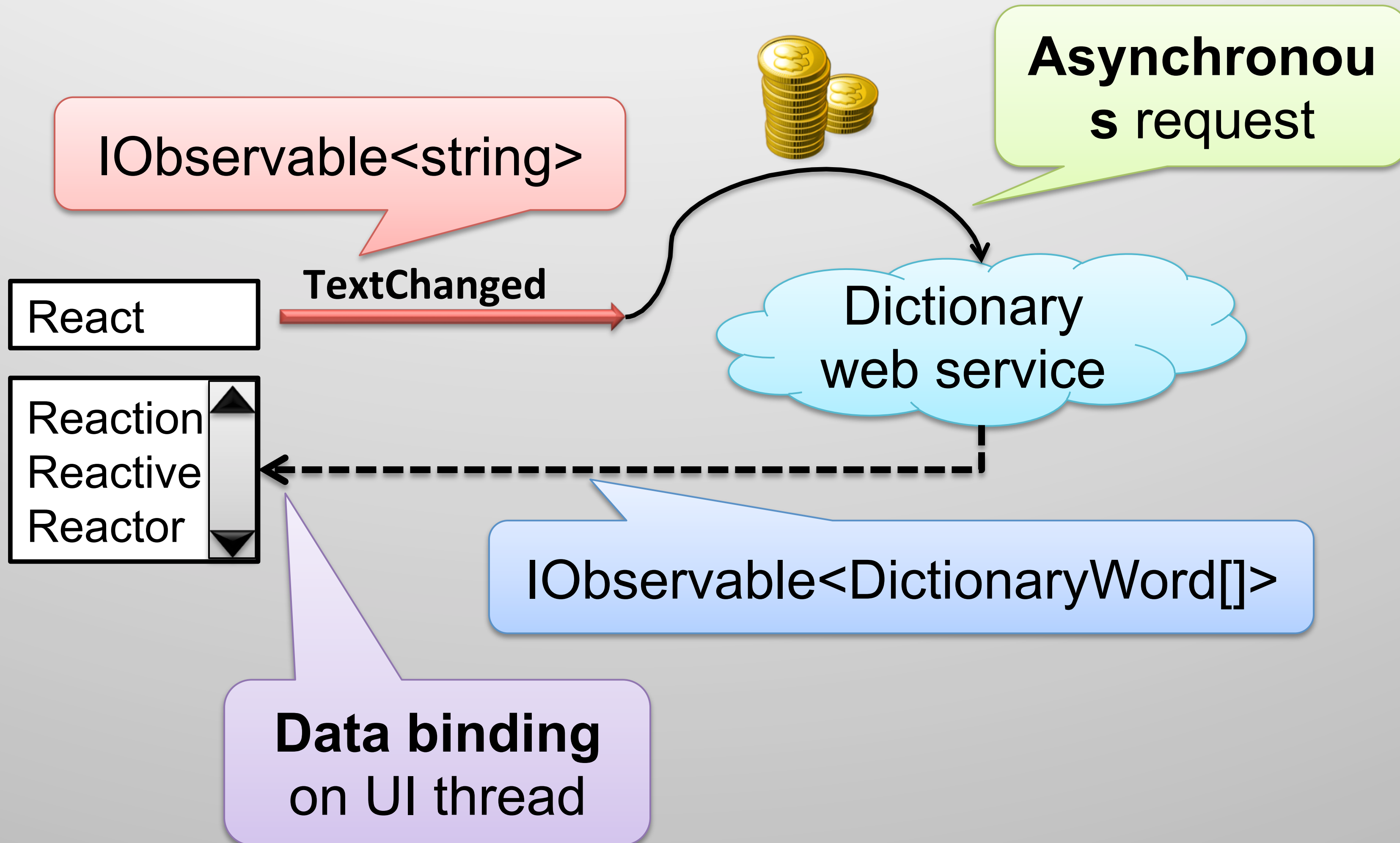
Standard Query Operators

- Observables are sources of data
  - Data is sent to you (**push based**)
  - Extra (optional) **notion of time**
- Hence we can query over them

```
// Producing an IObservable<Point> using Select
var mme = from mm in Observable.FromEvent<MouseEventArgs>(
                                 form, "MouseMove")
          select mm.EventArgs.Location;

// Filtering for the first bisector using Where
var res = from mm in mme
          where mm.X == mm.Y
          select mm;
```

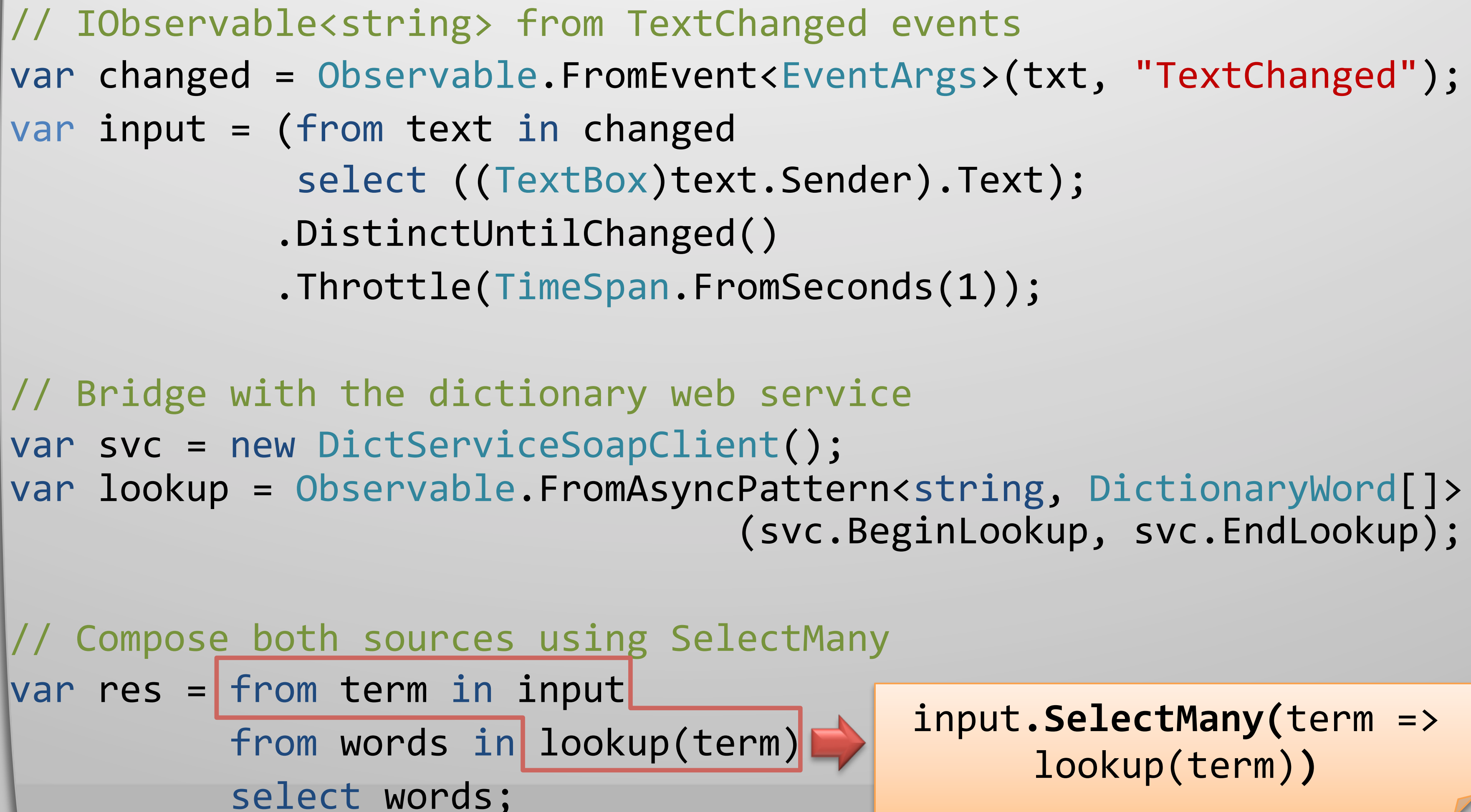
Composition and Querying

Putting the pieces together



Composition and Querying

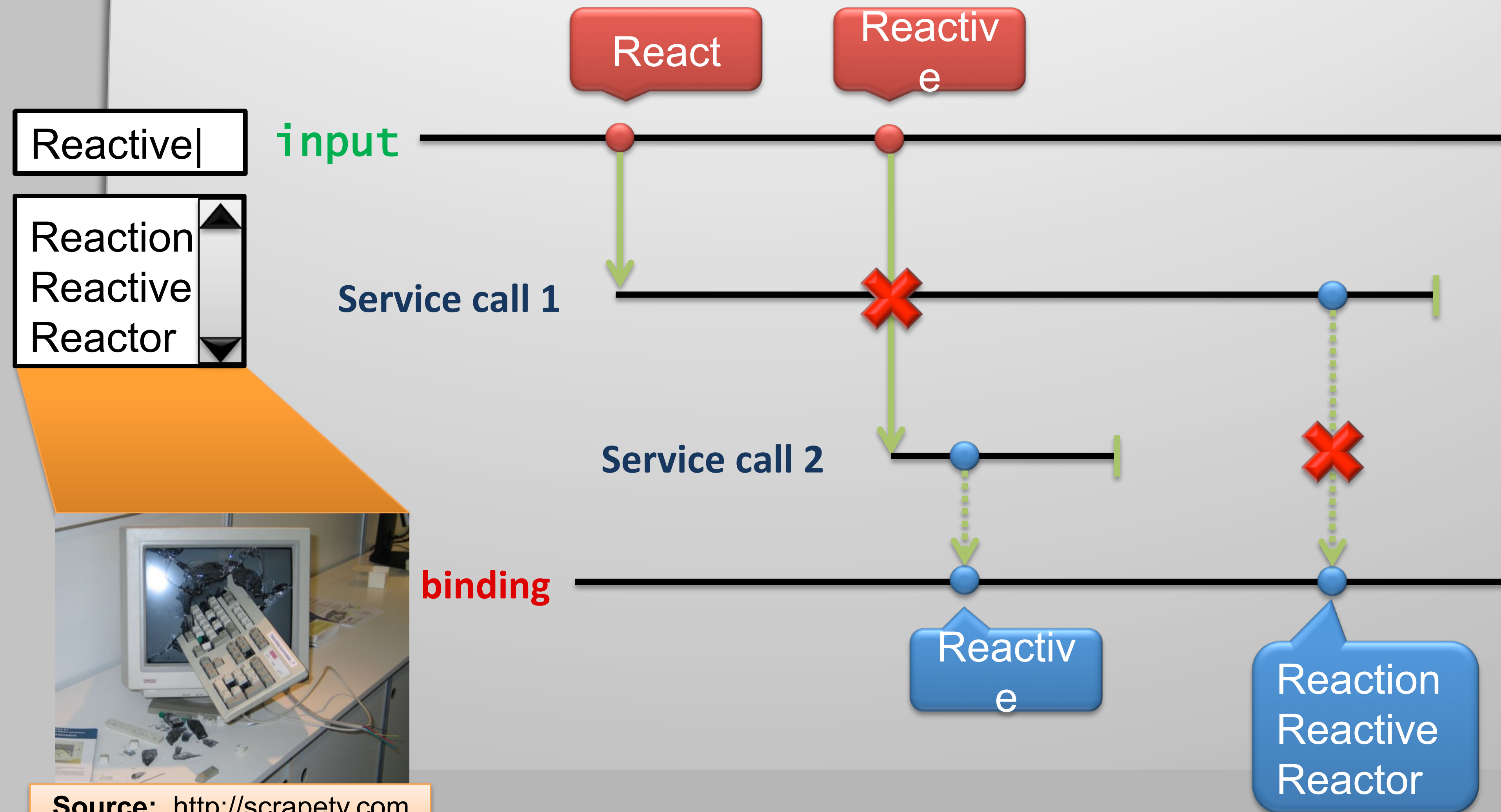
SelectMany – Composition at its best



Demo

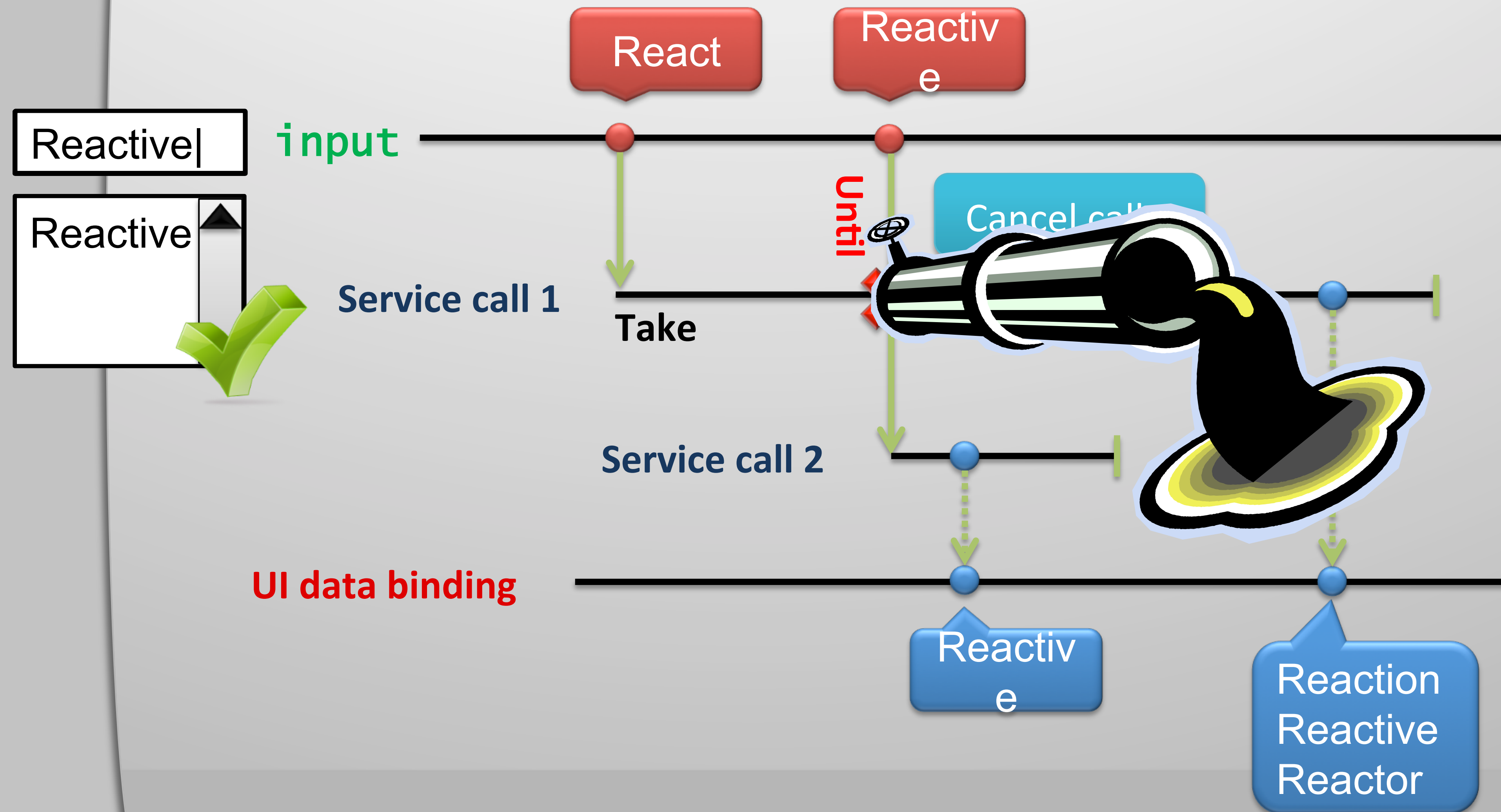
Composition and Querying

Asynchronous programming is hard…



Composition and Querying

Fixing out of order arrival issues



Composition and Querying

Applying the TakeUntil fix

```
// IObservable<string> from TextChanged events
var changed = Observable.FromEvent<EventArgs>(txt, "TextChanged");
var input = (from text in changed
             select ((TextBox)text.Sender).Text);
            .DistinctUntilChanged()
            .Throttle(TimeSpan.FromSeconds(1));

// Bridge with the dictionary web service
var svc = new DictServiceSoapClient();
var lookup = Observable.FromAsyncPattern<string, DictionaryWord[]>
                                       (svc.BeginLookup, svc.EndLookup);

// Compose both sources using SelectMany
var res = from term in input
          from words in lookup(term).TakeUntil(input)
          select words;
```

Very local fix

Composition and Querying

Applying the TakeUntil fix

```
// IObservable<string> from TextChanged events
var changed = Observable.FromEvent<EventArgs>(txt, "TextChanged");
var input = (from text in changed
             select ((TextBox)text.Sender).Text);
            .DistinctUntilChanged()
            .Throttle(TimeSpan.FromSeconds(1));

// Bridge with the dictionary web service
var svc = new DictServiceSoapClient();
var lookup = Observable.FromAsyncPattern<string, DictionaryWord[]>
                                       (svc.BeginLookup, svc.EndLookup);

// Alternative approach for composition using:
//   IObservable<T> Switch<T>(IObservable<IObservable<T>> sources)
var res = (from term in input
           select lookup(term)).Switch();
```

Hops from source to source

Demo

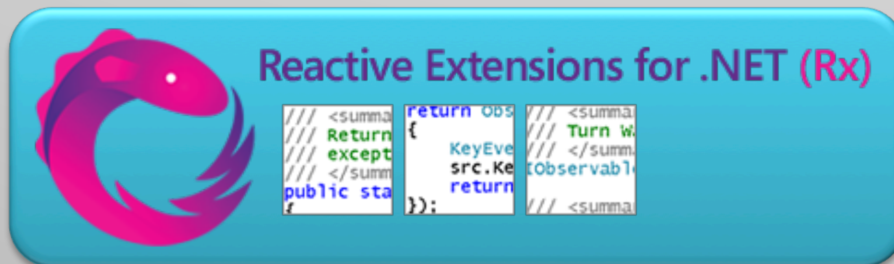
Mission Accomplished

Way ***simpler*** with Rx

$(f \circ g)(x) = f(g(x))$

Rx is a library for composing asynchronous and event-based programs using observable sequences

Queries! **LINQ!**



- .NET 3.5 SP1 and 4.0
- Silverlight 3 and 4
- XNA 3.1 for XBOX and Zune
- Windows Phone 7
- JavaScript (RxJS)

Download at **MSDN Data Developer Center** or use **NuGet**

Related Content

- Hands-on Labs
  - Two flavors:
    - Curing the asynchronous blues with the Reactive Extensions (Rx) for .NET
    - Curing the asynchronous blues with the Reactive Extensions (Rx) for JavaScript
  - Both can be found via the Rx forums
- Rx team web presence
  - **Rx team blog** – http://blogs.msdn.com/rxteam
  - **DevLabs** – http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx
  - **MSDN forums** – http://social.msdn.microsoft.com/Forums/en-US/rx
  - **Channel9** – http://channel9.msdn.com/Tags/Rx

Thanks!

Bart J.F. De Smet