Rx: curing your asynchronous programming blues

Bart J.F. De Smet
Cloud Programmability Team
bartde@microsoft.com

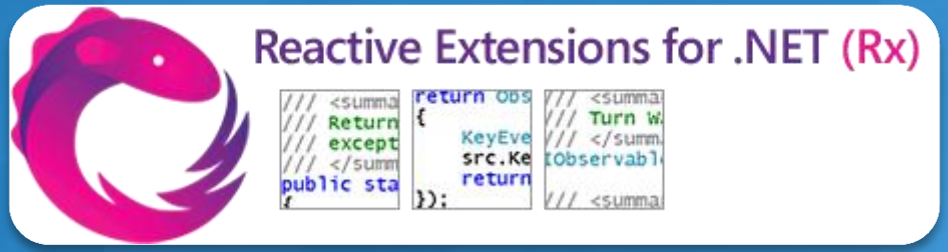
Mission statement

Too hard today…

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

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

Queries? LINQ?



Download at **MSDN DevLabs**

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

Essential Interfaces
Enumerables – a pull-based world

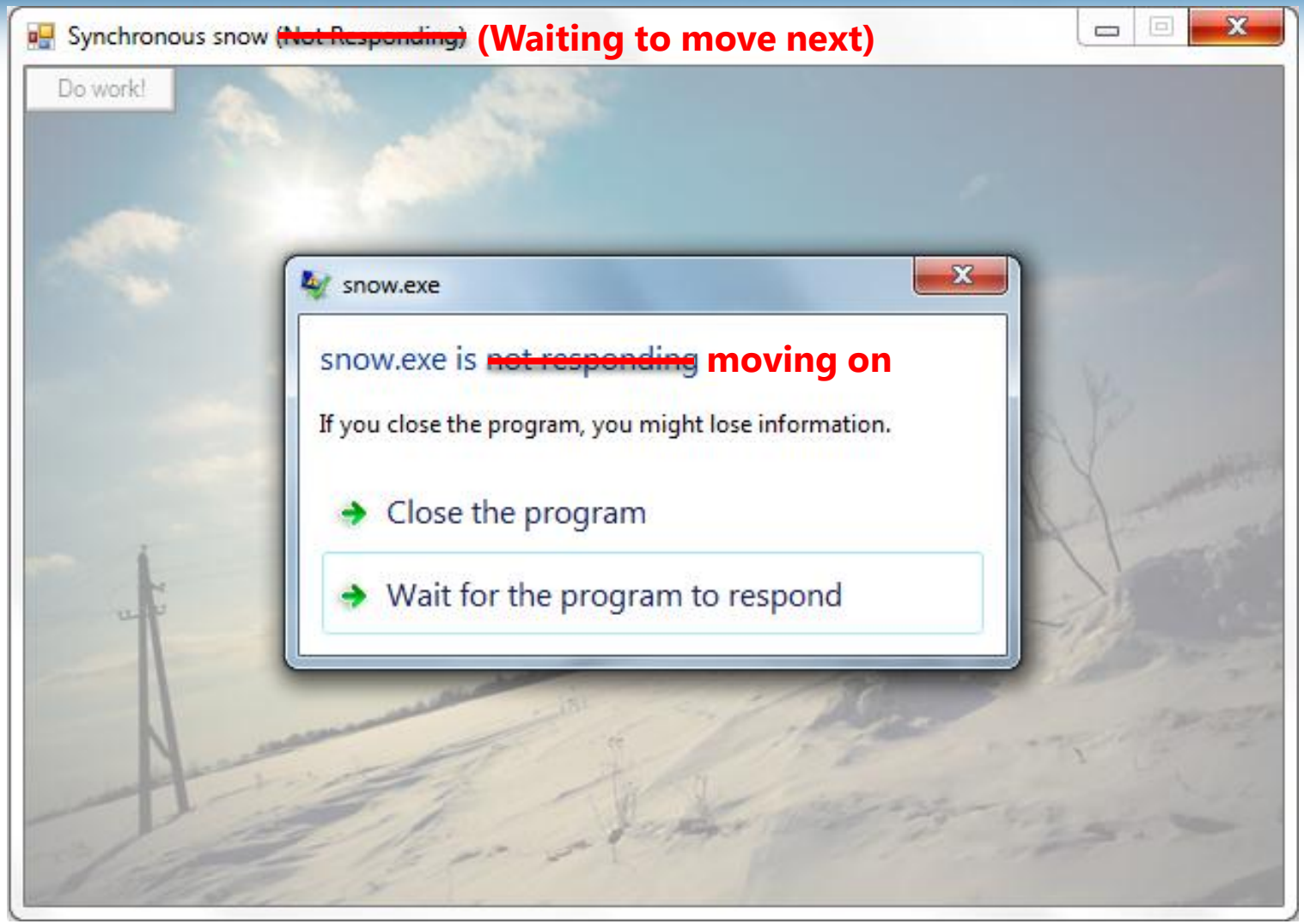

```
interface IEnumerable<out T>
{
    IEnumerator<T> GetEnumerator();
}

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

C# 4.0 covariance

You could get ***stuck***

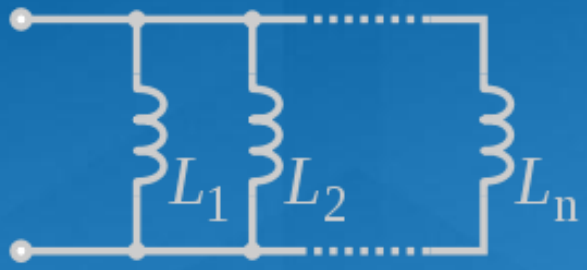
Essential Interfaces

Enumerables – a pull-based world



Mathematical duality
Because the Dutch are (said to be) cheap

- Electricity: inductor and capacitor



- Logic: De Morgan's Law

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

- Programming?

Mathematical duality

Because the Dutch are (said to be) cheap

- Electricity: indu… …citor

- Logic: $\neg(A \wedge B \ldots) \equiv \neg A \wedge \neg B$
- Programm…

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 1 – Simpler and more explicit

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

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

What's the dual of IEnumerable?
Step 1 – Simpler and more explicit

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

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

C# didn't *borrow* Java
checked exceptions

What's the dual of IEnumerable?
Step 1 – Simpler and more explicit

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

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

What's the dual of IEnumerable?
Step 1 – Simpler and more explicit

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

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

We really got an enumerator ***and*** a disposable

What's the dual of IEnumerable?
Step 2 – Swap input and output

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

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

Will ***only*** dualize the synchrony aspect

What's the dual of IEnumerable?
Step 2 – Swap input and output

```
interface IEnumerableDual<T>
{
  (IDisposable & void) GetEnumerator(IEnumerator<T>);
}

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

Set

What's the dual of IEnumerable?
Step 2 – Swap input and output

```
interface IEnumerableDual<T>
{
    IDisposable SetEnumerator(IEnumerator<T> x);
}

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

This is an ***output*** too

What's the dual of IEnumerable?
Step 2 – Swap input and output

```
interface IEnumerableDual<T>
{
    IDisposable SetEnumerator(IEnumerator<T> x);
}

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

Discrete domain with ***true*** and ***false***

What's the dual of IEnumerable?
Step 2 – Swap input and output

```
interface IEnumerableDual<T>
{
    IDisposable SetEnumerator(IEnumerator<T> x);
}

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

T

If you got ***true***, you really got a ***T***

What's the dual of IEnumerable?
Step 2 – Swap input and output

```
interface IEnumerableDual<T>
{
    IDisposable SetEnumerator(IEnumerator<T> x);
}

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

void

If you got ***false***,
you really got ***void***

What's the dual of IEnumerable?
Step 2 – Swap input and output

```
interface IEnumerableDual<T>
{
    IDisposable SetEnumerator(IEnumerator<T> x);
}

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

What's the dual of IEnumerable?
Step 2 – Swap input and output

```
interface IEnumerableDual<T>
{
    IDisposable SetEnumerator(IEnumeratorDual<T>);
}

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

Got

But C# doesn't have ***discriminated unions***…
Let's splat this into ***three methods*** instead!

What's the dual of IEnumerable?
Step 2 – Swap input and output

```
interface IEnumerableDual<T>
{
    IDisposable SetEnumerator(IEnumeratorDual<T>);
}

interface IEnumeratorDual<T>
{
    void    GotT(T value);
    void    GotException(Exception ex);
    void    GotNothing(void);
}
```

What's the dual of IEnumerable?
Step 3 – Consult the Gang of Four…

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

interface IObserver<T>
{
    void   GotT(T value);
    void   GotException(Exception ex)
    void   GotNothing();
}
```

What's the dual of IEnumerable?
Step 4 – Variance annotations

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

interface IObserver<in T>
{
    void   GotT(T value);
    void   GotException(Exception ex);
    void   GotNothing();
}
```

Do you ***really*** know C# 4.0?

Used to ***detach*** the observer…

What's the dual of IEnumerable?
Step 5 – Color the bikeshed (*)

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

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

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

Essential Interfaces

Observables – a push-based world

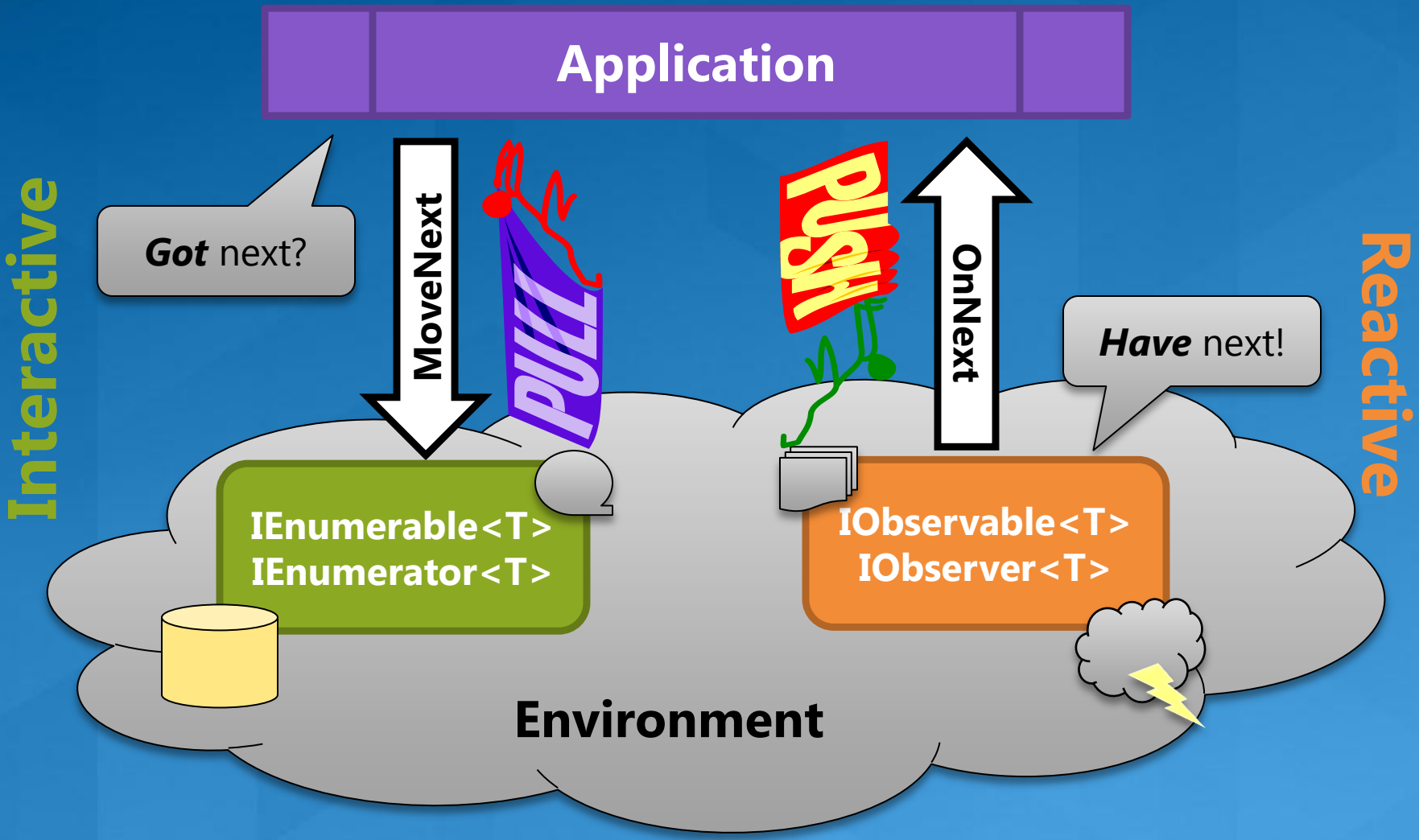

```
interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

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

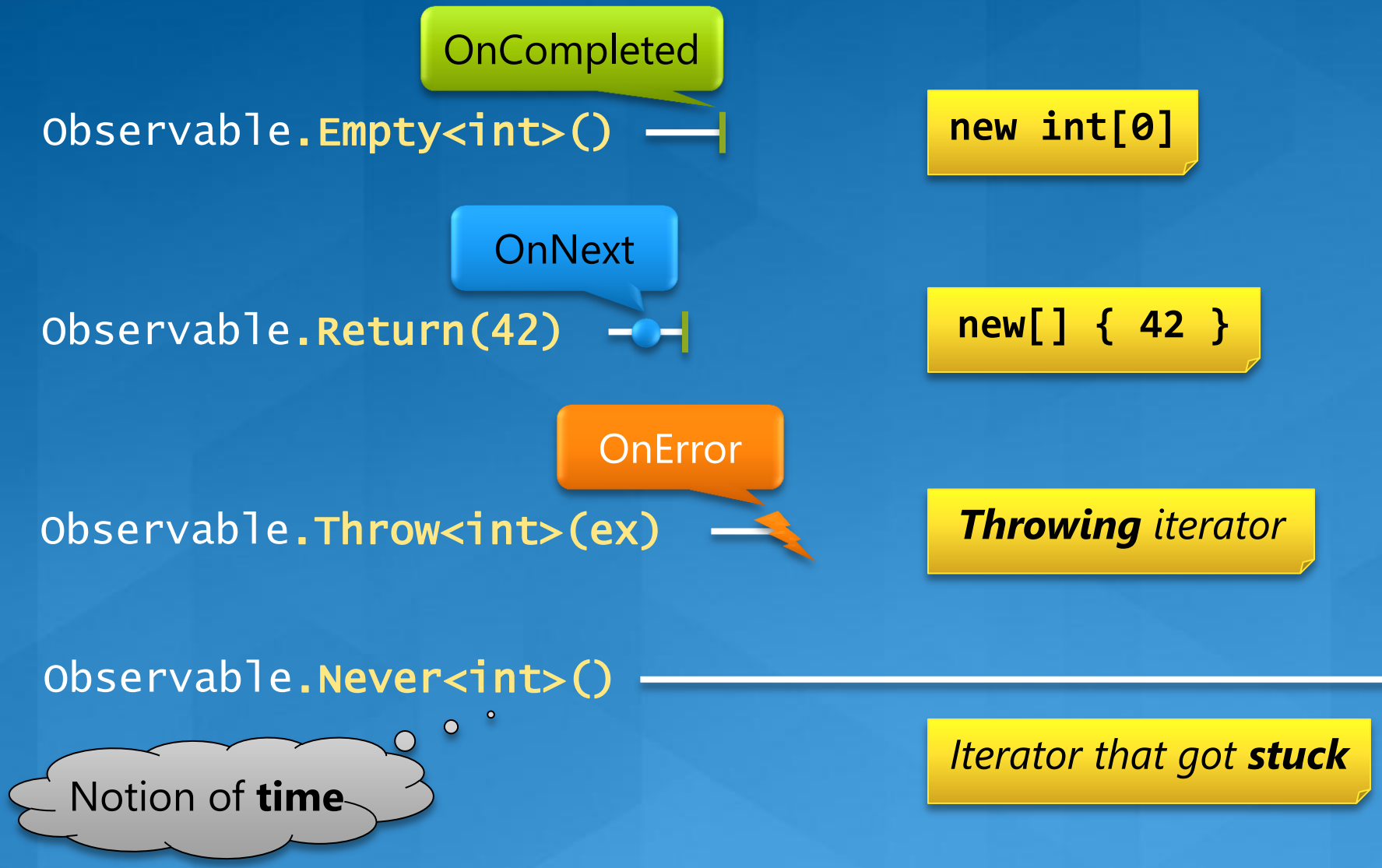


demo

Essential Interfaces

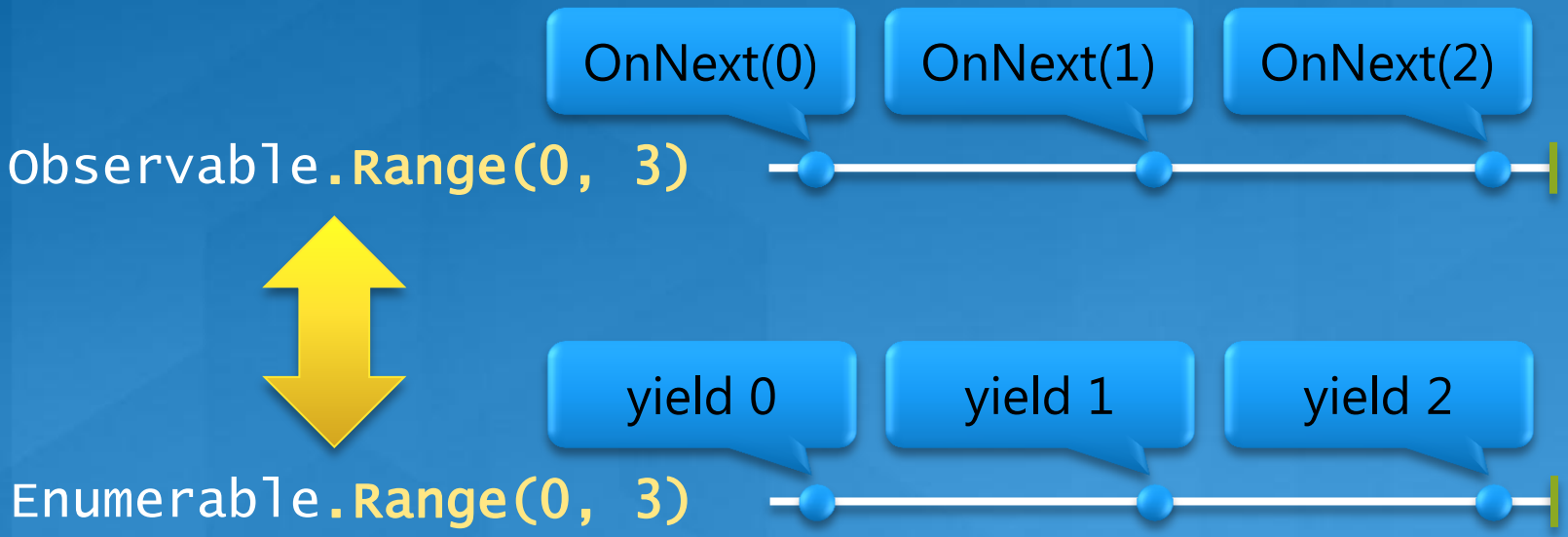
Getting Your Observables

Primitive constructors



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 – our most generic creation operator

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

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 – our most generic creation operator

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

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…
```

F10

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

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

Getting Your Observables

Create – our most generic creation operator

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

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…
```

Getting Your Observables

Create – our most generic creation operator

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

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

Getting Your Observables

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?

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…

Exceptions?

Hidden data source

```
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
);
```

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
  - other sources like SSIS, PowerShell, 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

Bridging Rx with the World

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**

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

Parameterization of operators

Will run on the source's scheduler

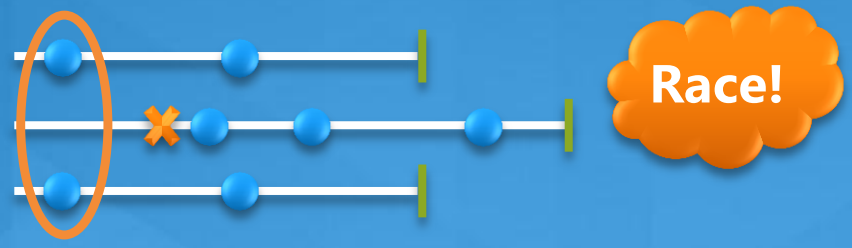
Composition and Querying
Concurrency and synchronization

- 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:

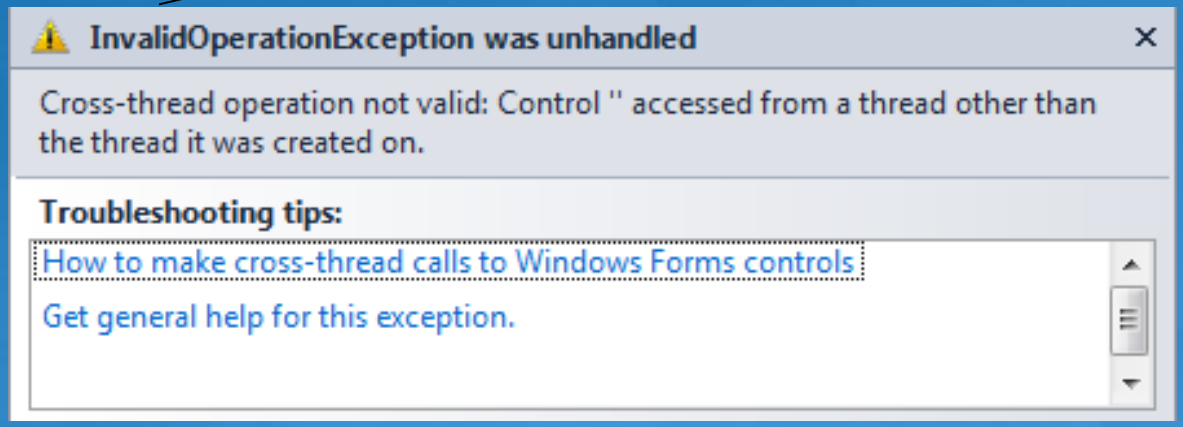
```
source1
source2
source1.Amb(source2)
```



Composition and Querying
Concurrency and synchronization

- How to **synchronize**?

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



- IScheduler
- WPF dispatcher
- WinForms control
- SynchronizationContext

- Compositionality to the rescue!

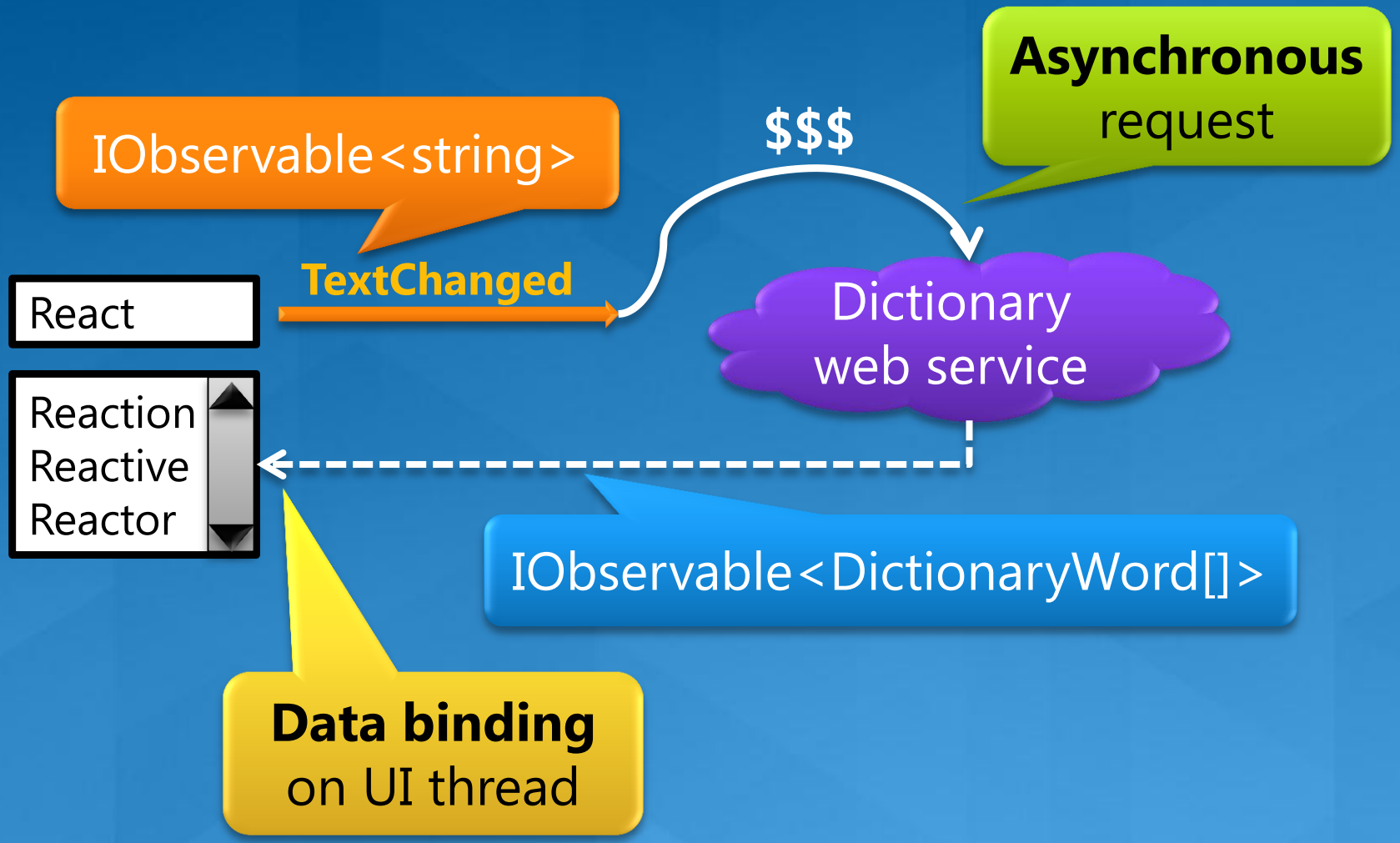
```
xs.ObserveOn(frm)
.Subscribe(x => lbl.Text = "Answer = " + x);
```

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

```
// 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)
          select words;
```

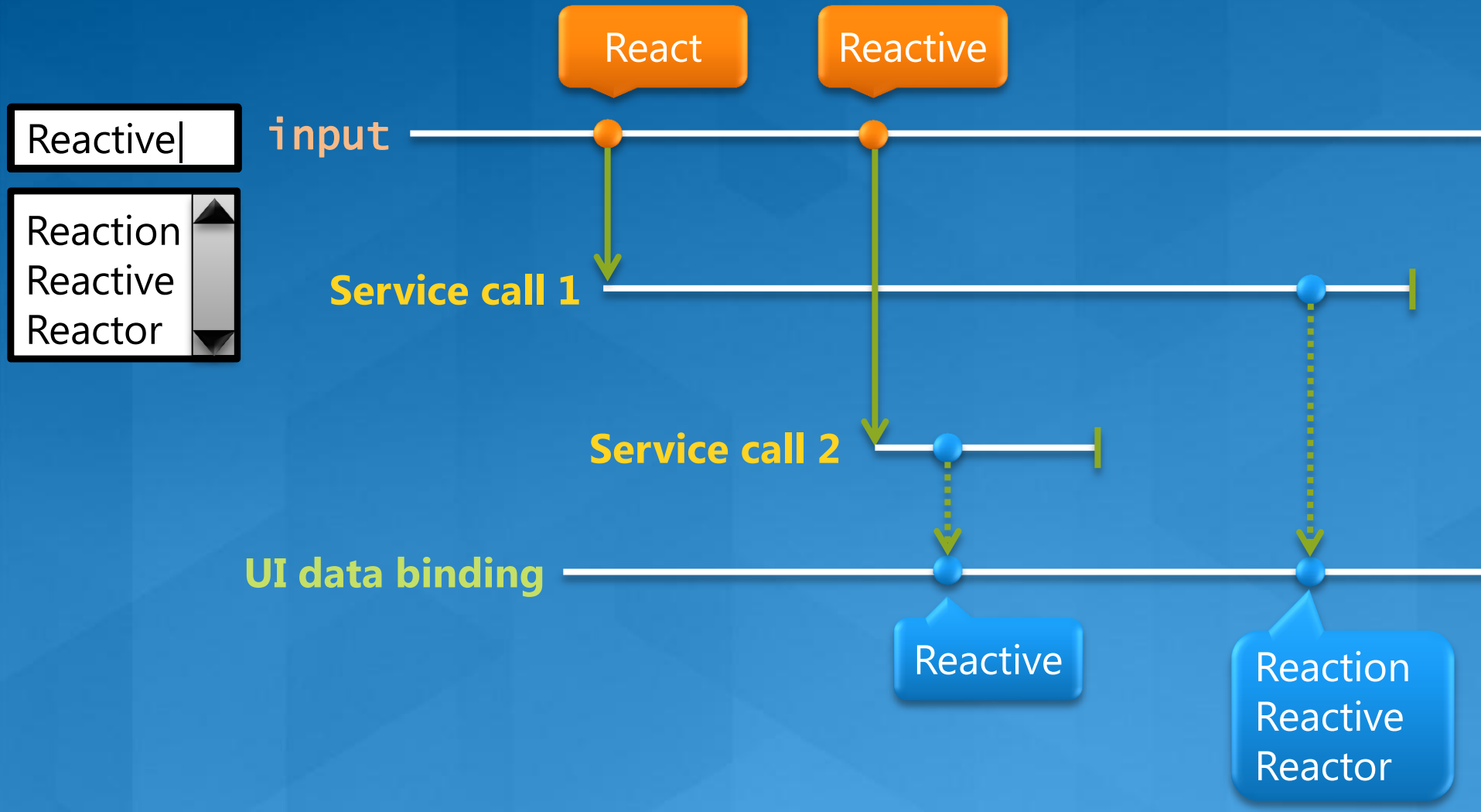
```
input.SelectMany(term =>
    lookup(term))
```

demo

Composition and Querying

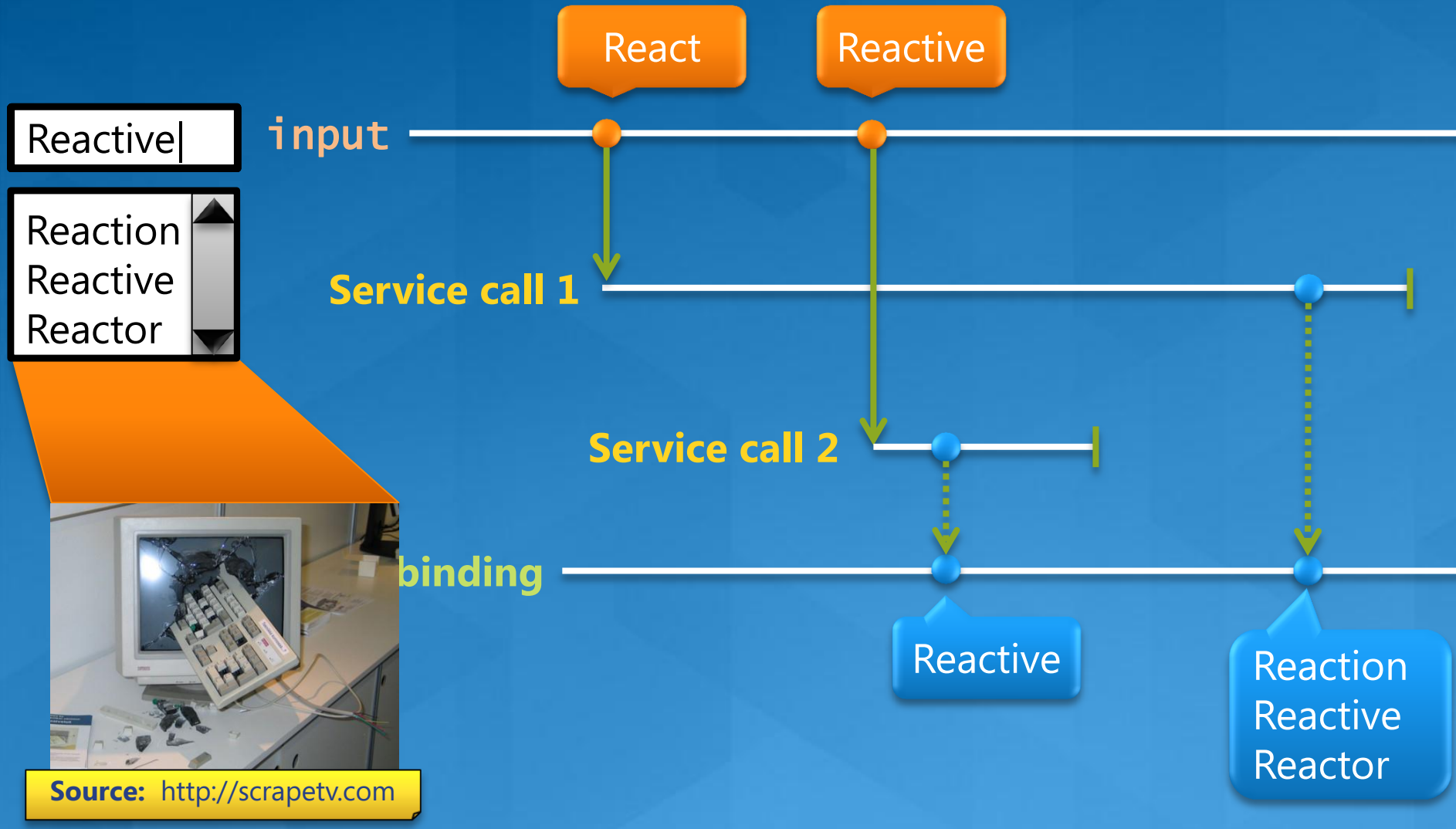
Composition and Querying

Asynchronous programming is hard…



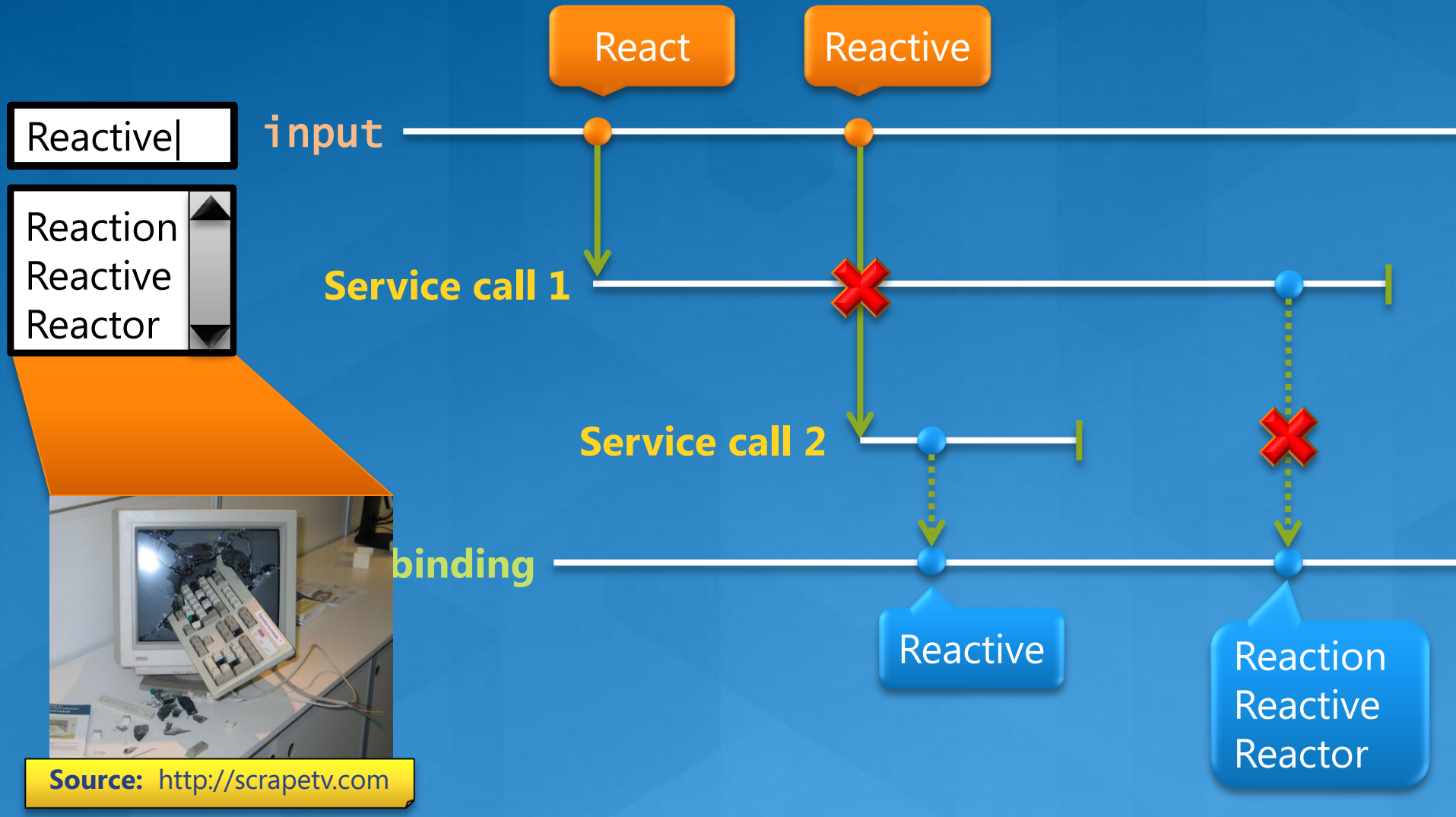
Composition and Querying

Asynchronous programming is hard…



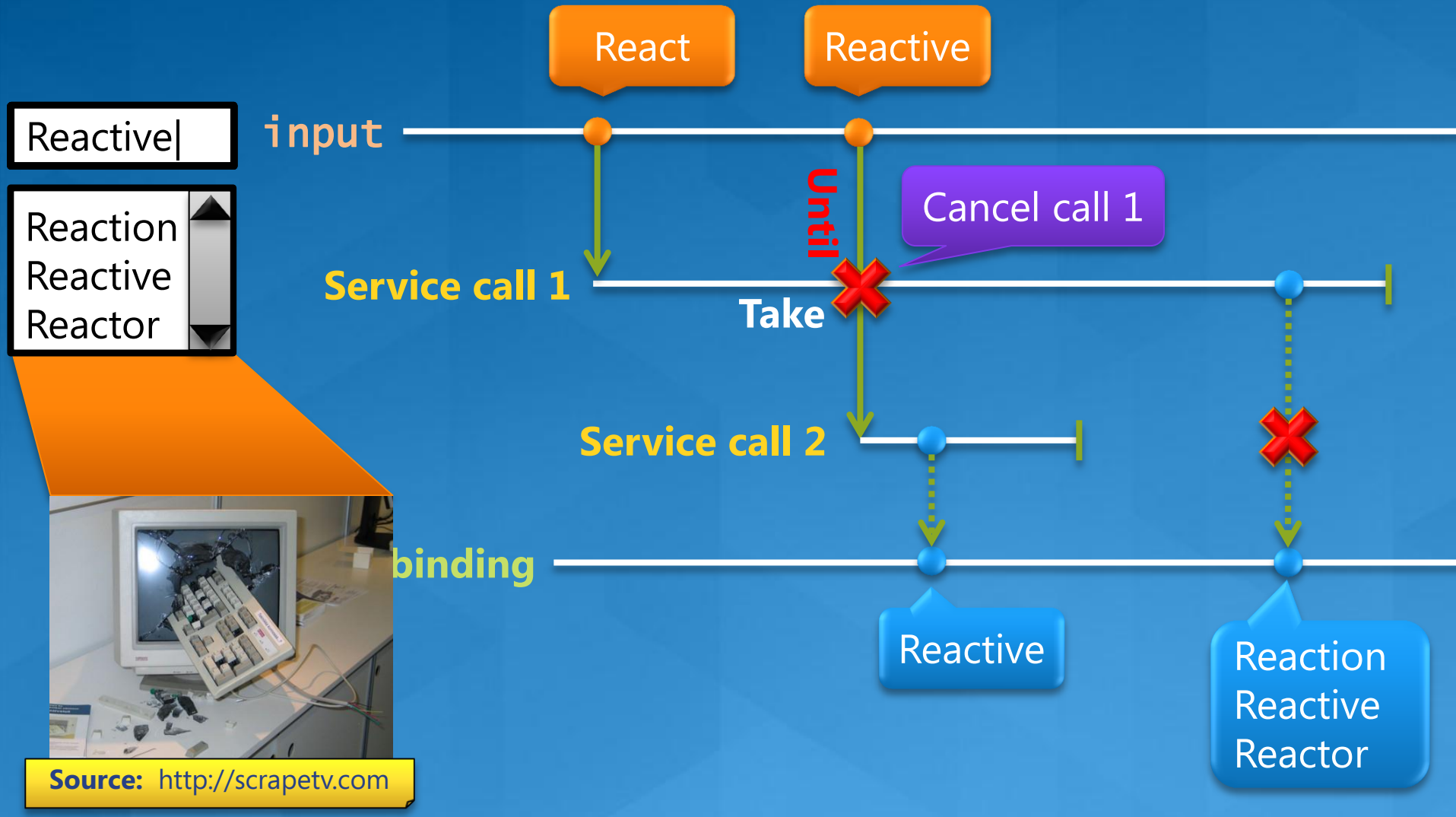
Composition and Querying

Asynchronous programming is hard…



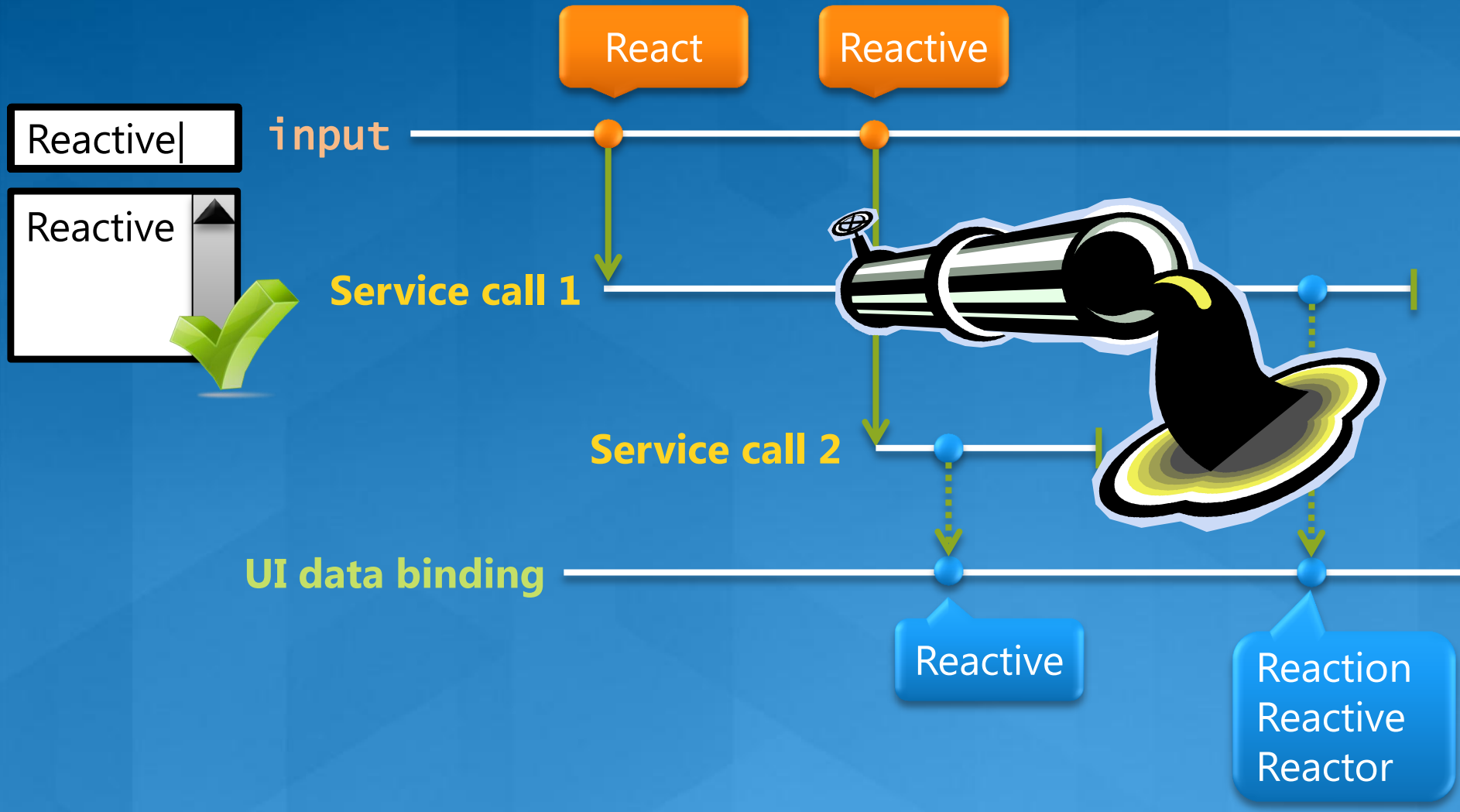
Composition and Querying

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 ☺

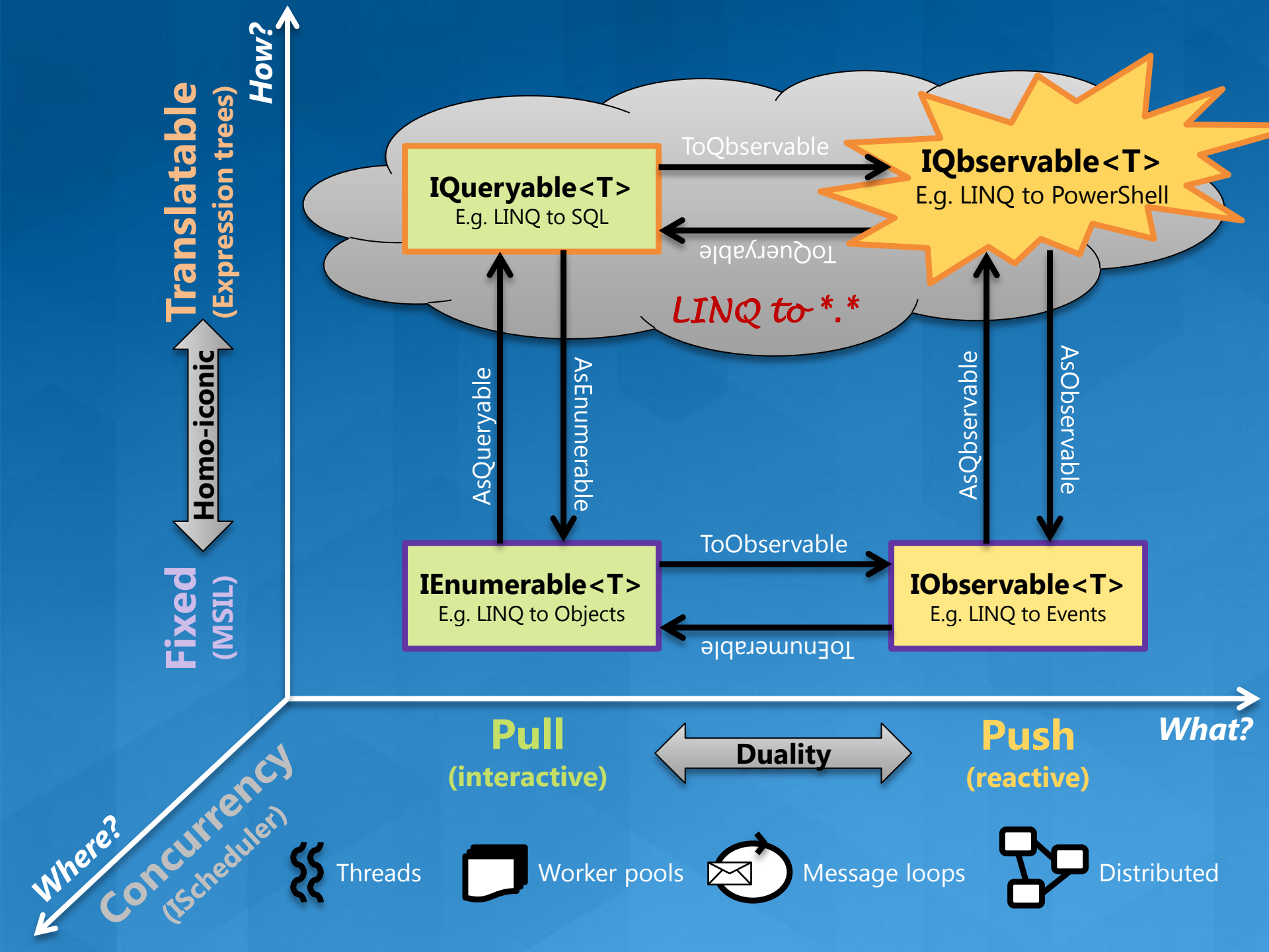
Composition and Querying
Applying the TakeUntil fix

```
// 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

Fixing asynchronous issues


How?
Translatable
(Expression trees)
Homo-iconic
Fixed
(MSIL)
IQueryable<T>
E.g. LINQ to SQL
ToQObservable
ToQueryable
IQbservable<T>
E.g. LINQ to PowerShell
LINQ to *.*
AsQueryable
AsEnumerable
AsQbservable
AsObservable
IEnumerable<T>
E.g. LINQ to Objects
ToObservable
ToEnumerable
IObservable<T>
E.g. LINQ to Events
What?
Pull
(interactive)
Duality
Push
(reactive)
Where?
Concurrency
(IScheduler)
Threads
Worker pools
Message loops
Distributed

demo

LINQ to WMI Events (WQL)

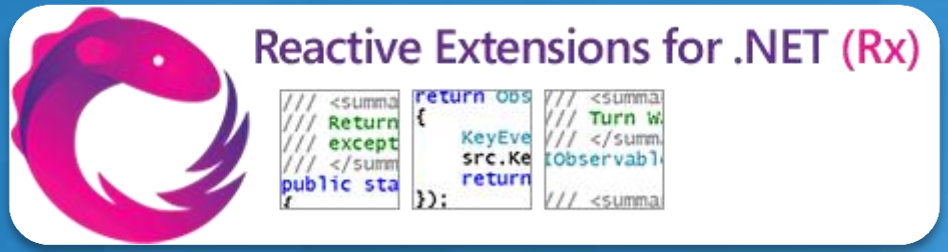
Mission accomplished

Way ***simpler*** with Rx

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

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

Queries! LINQ!



Download at **MSDN DevLabs**

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

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