

# Reaktivní programování v .NET

Tomáš Petříček

<http://tomasp.net/blog>  
[tomas@tomasp.net](mailto:tomas@tomasp.net)

# Co je „reaktivní programování“?

Psaní aplikací které reagují na události

- » Klasické .NET „eventy“

  - Například MouseDown, KeyPress, atd...

- » Asynchronní operace

  - Například dokončení stahování souboru apod.

Jak se to dělá v současné době...

- » Imperativní zápis

  - Přidávání handlerů pomocí “+=“

- » Modifikace nějakého lokálního stavu

# Jak by to šlo řešit lépe?

## Deklarativní programování

- » Obecný přístup k zápisu řešení
- » Popisujeme „co chceme získat“ a ne „jak to udělat“
- » Nemusíme řešit implementační detaily

## Deklarativní přístup v .NETu

- » .NET atributy, XAML zápis, data binding, ...
- » **LINQ**: Language Integrated Query

# Agenda

Úvod

**Deklarativní programování v LINQu**

Microsoft Rx Framework

LINQ dotazy pro práci s událostmi

Jak jsou reprezentované události?

Práce s událostmi pomocí Observable třídy

Reaktivní programování v F#

Asynchronní programování s událostmi

# Deklarativní programování v LINQu

```
var q =  
    from product in db.Products  
    where product.Price > 100.0  
    select product.Name;
```

## Deklarativní zápis dotazů

- » Popisuje „jaké“ chceme získat výsledky, ne „jak“
- » Bere nějaký vstup typu **IEnumerable**
- » Generuje nějaké **IEnumerable** jako výsledek

# Agenda

## Úvod

Deklarativní programování v LINQu

## Microsoft Rx Framework

**LINQ dotazy pro práci s událostmi**

Jak jsou reprezentované události?

Práce s událostmi pomocí Observable třídy

## Reaktivní programování v F#

Asynchronní programování s událostmi

# LINQ dotazy pro události

Kdybychom měli **IEnumerable<MouseEventArgs>**...

```
var filtered =  
    from evt in mouseDowns  
    where evt.Button == MouseButton.Right  
    select String.Format("Right click: {0}, {1}",  
        evt.X, evt.Y);
```

Deklarativní práce se seznamy...

- » Bereme nějakou kolekci (**IEnumerable**) jako vstup
- » Generujeme nějakou kolekci jako výstup
- » Můžeme ji předat dále nebo zpracovat přes **foreach**

# LINQ dotazy pro události

Podobně funguje **IObservable<MouseEventArgs>**...

```
var filtered =  
    from evt in mouseDowns  
    where evt.EventArgs.Button == MouseButton.Right  
    select String.Format("Right click: {0}, {1}",  
        evt.EventArgs.X, evt.EventArgs.Y);
```

Deklarativní práce se událostmi...

- » Bereme nějakou událost (**IObservable**) jako vstup
- » Generujeme nějakou událost jako výstup
- » Můžeme ji předat dále nebo zpracovat přes **Subscribe**

# Jak to vypadá celé?

Vytváří IObservable z události pomocí reflection

```
var mouseDowns = Observable.  
    FromEvent<MouseEventArgs>(btnClick, "MouseDown")
```

Deklarativně vytváří vyfiltrovanou událost

```
var filtered =  
    from evt in mouseDowns  
    where evt.EventArgs.Button == MouseButton.Right  
    select String.Format("Right click: {0}, {1}",  
        evt.EventArgs.X, evt.EventArgs.Y);
```

```
filtered.Subscribe(msg => MessageBox.Show(msg));
```

Zaregistrujeme kód pro zpracování filtrované události

# Demo

Práce s událostmi pomocí LINQu

# Agenda

## Úvod

Deklarativní programování v LINQu

## Microsoft Rx Framework

LINQ dotazy pro práci s událostmi

**Jak jsou reprezentované události?**

Práce s událostmi pomocí Observable třídy

## Reaktivní programování v F#

Asynchronní programování s událostmi

# Co je vlastně IEnumerable?

Objekt, který generuje **IEnumerator**:

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

Rozhraní nám umožňuje následující:

- » Posunout se na další element
- » Zjistíme zda další element existuje
- » Zjistíme hodnotu dalšího elementu

# Jak reprezentovat události

Objekt, kterému registrujeme **IObserver**:

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

Po registraci bude volat událost jednotlivé metody:

- » V případě, že je k dispozici další element
- » V případě, že „observable“ končí
- » V případě, že došlo k nějaké chybě

# IObservable v .NETu

Lze vytvářet pomocí **Observable.FromXyz**

Standardní události jako například **MouseDown**

- » Volají **OnNext** v případě že se událost stane
- » Nikdy nekončí, nikdy se nevolá **OnCompleted**

Asynchronní volání a čekání na výsledek

- » Zavolá **OnNext** jednou, poté ihned **OnCompleted**

Vlastnost která se může měnit (**DependencyProperty**)

- » Může upozorňovat na změny jako události

# Registrace IObservable objektů

Stejný princip jako IEnumerator a IObservable

» **Kolekce** – vracejí hodnoty jako výsledek

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

» **Reaktivní** – „něco“ zavolá naší metodu/objekt

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

Matematicky: Reaktivní svět je duální ke kolekcím

# Agenda

## Úvod

Deklarativní programování v LINQu

## Microsoft Rx Framework

LINQ dotazy pro práci s událostmi

Jak jsou reprezentované události?

**Práce s událostmi pomocí Observable třídy**

## Reaktivní programování v F#

Asynchronní programování s událostmi

# Další operátory pro události

## Běžné metody známé z LINQ dotazů

- » **Aggregate** – výpočet jedné hodnoty (např. Max, Count)

```
R Aggregate<T, R>(IObservable<T> source,  
                  R seed, Func<R, T, R> aggregate);
```

- » **Merge** – spojení více událostí stejného typu

```
IObservable<T> Merge<T>(IObservable<T>[] sources);
```

- » **Take/Skip** – ignorování nějakých z událostí

```
IObservable<T> Take<T>(IObservable<T> src, int count);  
IObservable<T> Skip<T>(IObservable<T> src, int count);
```

# Další operátory pro události

## Další metody vhodné pro události

- » **Scan** – jako Aggregate, ale hlásí průběžný stav

```
IObservable<R> Scan<T, R>(IObservable<T> src,  
                        R seed, Func<R, T, R> accumulator);
```

- » **Until** – spojení více událostí stejného typu

```
IObservable<T> Until<T, O>(IObservable<T> src,  
                          IObservable<O> other)
```

- » **Interval** – generování událostí každých X sekund

```
IObservable<long> Interval(int duration)
```

# Demo

Scan a další operátory...

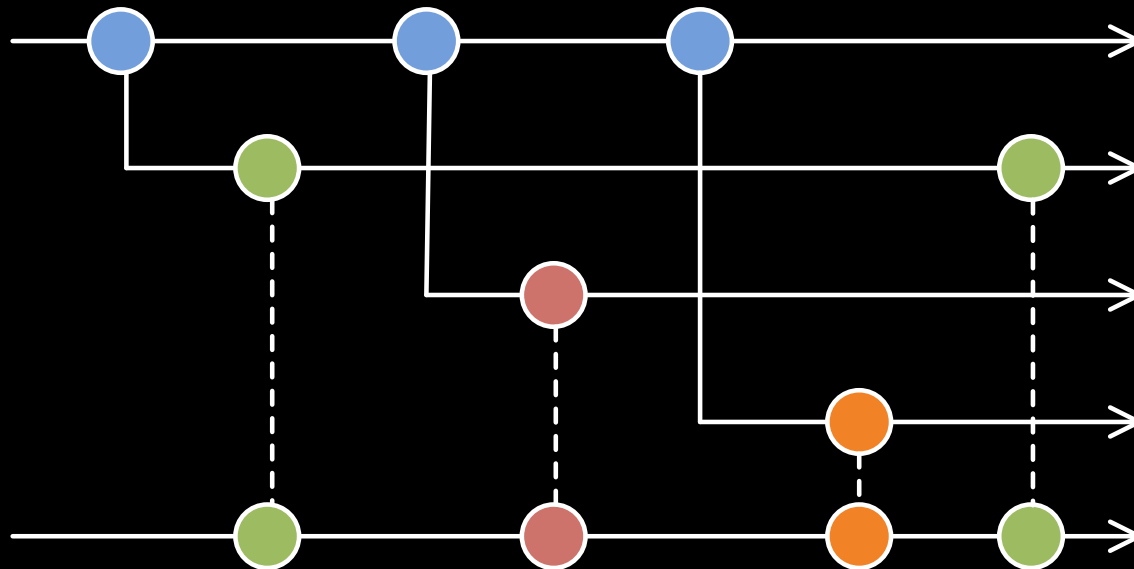
# Operace „Flatten“

`IEnumerable<IEnumerable<T>> -> IEnumerable<T>`

» Spojení všech prvků „kolekce kolekcí“ do jedné...

`IObservable<IObservable<T>> -> IObservable<T>`

» Spojení všech událostí „generovaných událostí“ do jedné...



# Demo

Dotazy s více **from** klauzulemi

# Agenda

## Úvod

Deklarativní programování v LINQu

## Microsoft Rx Framework

LINQ dotazy pro práci s událostmi

Jak jsou reprezentované události?

Práce s událostmi pomocí Observable třídy

## Reaktivní programování v F#

**Asynchronní programování s událostmi**

# Asynchronní programování

Zápis programu tak, aby neblokoval vlákno

```
let http(url:string) =  
    async { let req = HttpWebRequest.Create(url)  
            let! rsp = req.AsyncGetResponse()  
            let reader = new StreamReader(rsp.GetResponseStream())  
            return! reader.AsyncReadToEnd() }  
  
let pages = Async.Parallel [ http(url1); http(url2) ]
```

Lze použít pro různé „návrhové vzory“

» Paralelizace, reaktivní programování atd...

# Reaktivní programování s *async*

## Paralelní programování pomocí *async*

- » Paralelně běžící „agenti“ kteří komunikují
- » Používají se vlákna z *thread pool*

## Reaktivní programování je další návrhový vzor

- » Používá stejný jazyk, ale jiné knihovny
- » Více agentů běží na **jednom vlákně**
- » Většinou pouze čekají na událost, pak **rychle reagují**

# Příklad: Počítání kliknutí

Zobrazuj

Bere 'int' jako parametr  
a vytváří 'Async<unit>'

levým tlačítkem

```
let rec loop(count) =  
  async {  
    let! me = Reactive.AwaitEvent(lbl.MouseDown)  
    let add = if me.Button = MouseButton.Left then 1 else 0  
    lbl.Text <- sprintf "Clicks: %d" (count + add)  
    return! loop(count + add)  
  }
```

Spustí zbytek kódu až  
když dojde ke kliknutí

Rekurzivně voláme 'loop'

```
loop(0) |> Async.Start
```

Vypadá to jako agregace událostí z dřívějšíka...

» V tomto případě lze napsat snadno pomocí Aggregate

# Příklad: Počítání kliknutí

Změna – omezíme počet kliků za vteřinu

```
let rec loop(count) =  
  async {  
    let! me = Reactive.AwaitEvent(lbl.MouseDown)  
    let add = if me.Button = MouseButton.Left then 1 else 0  
    lbl.Text <- sprintf "Clicks: %d" (count + add)  
    let! _ = Reactive.Sleep(1000)  
    return! loop(count + add)  
  }
```

Spustí další část kódu až  
po 1000 milisekundách

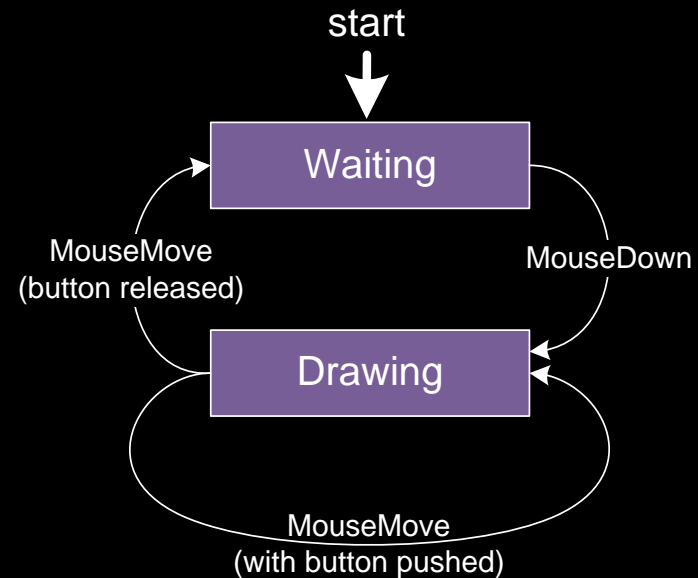
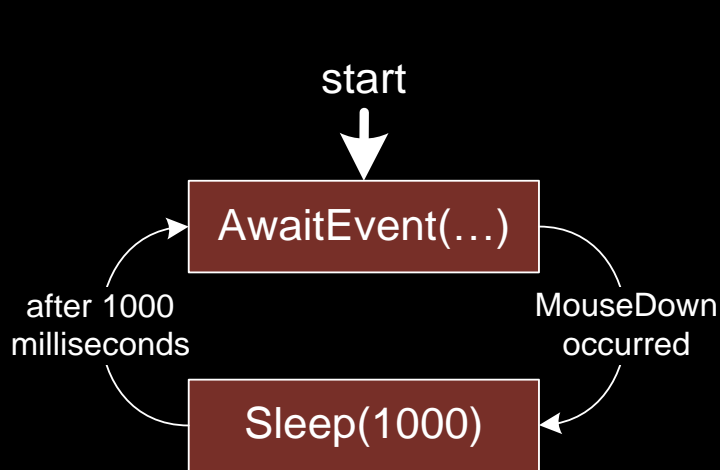
```
loop(0) |> Async.Start
```

Jak lze snadno popisovat agenty obecně?

# Agent jako stavový automat

Vhodné pro (téměř?) všechny složité problémy

- » Stavy – čekání na nějakou událost
- » Přechody – způsobené vyvoláním události



Problém „do budoucna“ – výběr mezi více přechody

# Demo

Ukázková reaktivní aplikace v F#

# Díky za pozornost

» **Otázky a v lepším případě i odpovědi...**

## Další informace:

» Moje mailová adresa:

- [tomas@tomasp.net](mailto:tomas@tomasp.net)

» Reactive Framework:

- <http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx>