Simplifying Asynchrony and Multithreading

Edin Kapić

@ekapic www.edinkapic.com

Concepts

Asynchronous operations in Rx

Concurrency and multithreading

Schedulers

Asynchronous Operations

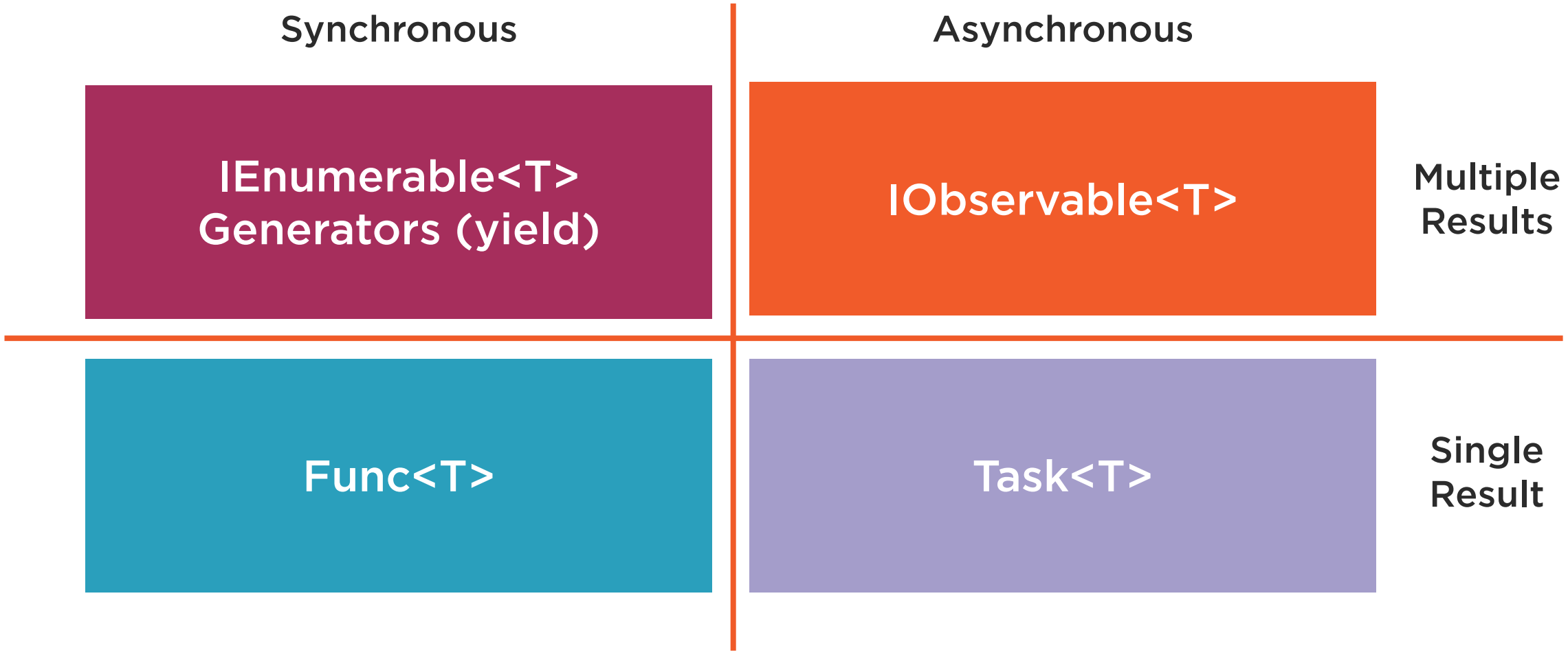
Executed out-of-order

Non-blocking

Callbacks

Native in Rx

Asynchrony in Rx



Multithreading in Rx

Rx isn’t multithreaded, but free-threaded and least concurrent by default.

Multithreading

More than one execution thread

Offloading computations

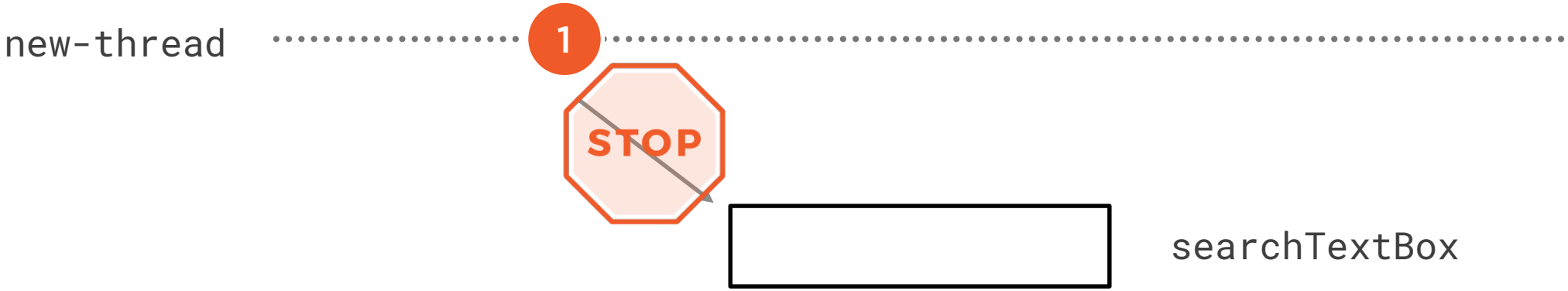
Thread safety

Abstracted in Rx

Multithreading in Rx

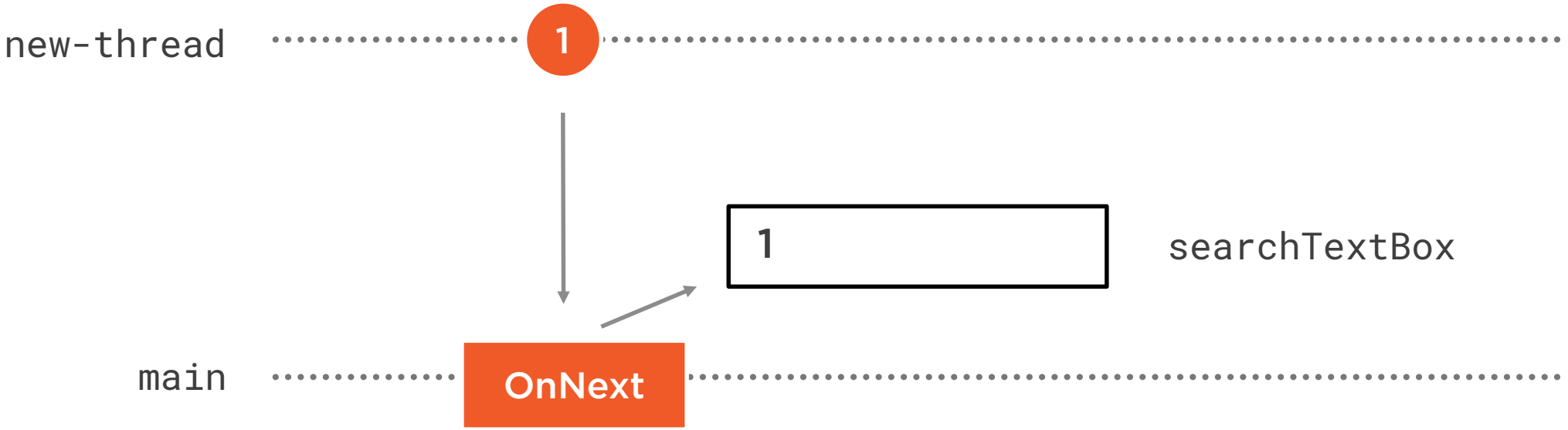

```
Observable.Interval(TimeSpan.FromSeconds(1))
  .Subscribe(...);
```

Multithreading UI Access




```
Observable.Interval(TimeSpan.FromSeconds(1))
  .Subscribe(t => searchTextBox.Text = t.ToString());
```

ObserveOn

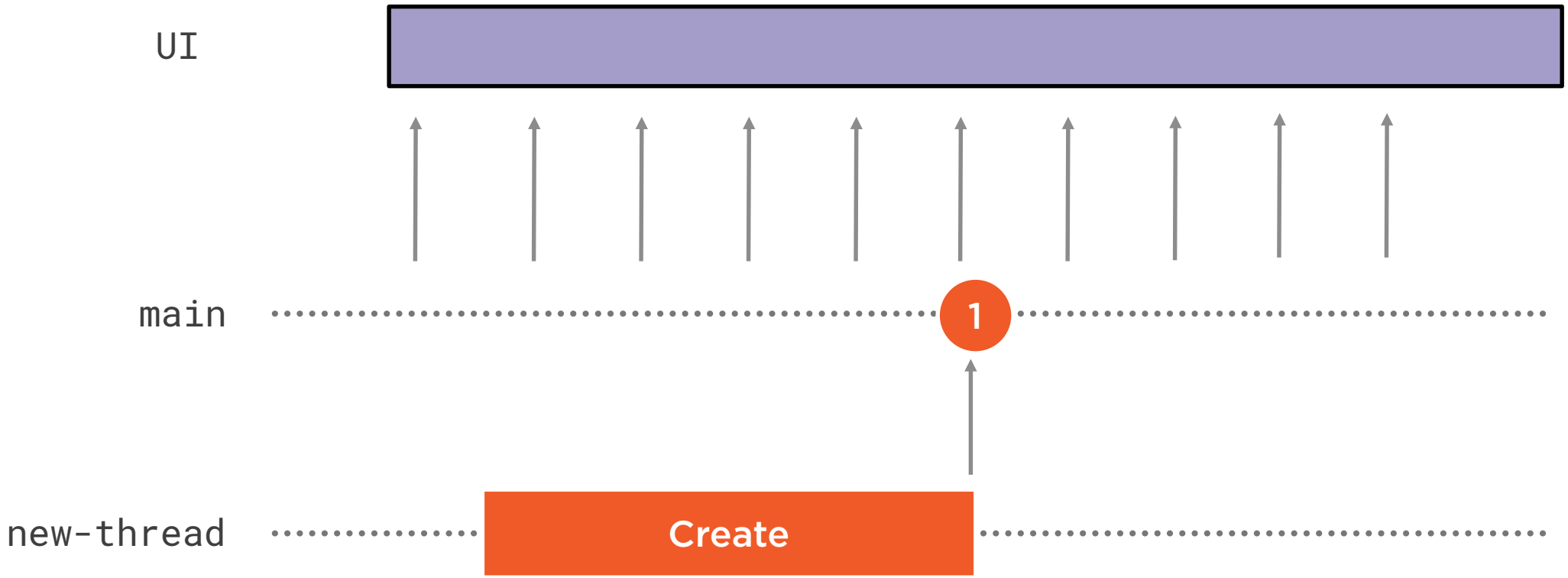



```
Observable.Interval(TimeSpan.FromSeconds(1))
  .ObserveOn(SynchronizationContext.Current)
  .Subscribe(t => searchTextBox.Text = t.ToString());
```

Costly Computations in Rx


```
Observable.Create( /* costly computations */ )
.Subscribe(...)
```

Offloading Computations with SubscribeOn




```
Observable.Create( /* costly computations */ )
  .SubscribeOn(NewThreadScheduler.Default)
  .ObserveOn(SynchronizationContext.Current)
  .Subscribe(...)
```

Schedulers

Schedulers allow us to schedule an action in Rx

- IScheduler interface

It is an abstraction over a future execution

- All Rx concurrency is managed by a scheduler

Rx also provides ready-made schedulers

```
new Thread(() => { /* do work */ }).Start()
ThreadPool.QueueUserWorkItem(_ => { /* do work */ }, null)
Task.Factory.StartNew(() => { /* do work */ })
syncCtx.Post(_ => { /* do work */ }, null)
Dispatcher.BeginInvoke(() => { /* do work */ })
```

Schedulers abstract future actions

```
scheduler.Schedule(() => { /* do work */ })
```

IScheduler Interface

```
public interface IScheduler
{
    DateTimeOffset Now { get; }
    IDisposable Schedule<TState>(TState state,
      Func<IScheduler, TState, IDisposable> action);
    IDisposable Schedule<TState>(TState state,
      DateTimeOffset dueTime,
      Func<IScheduler, TState, IDisposable> action);
    IDisposable Schedule<TState>(TState state,
      TimeSpan dueTime,
      Func<IScheduler, TState, IDisposable> action);
}
```

Schedulers

Immediate

CurrentThread

EventLoop

Dispatcher

NewThread

TaskPool
ThreadPool

Schedulers Everywhere

Observable.Return(42)

Observable.Create()

```
scheduler.Schedule(a);
scheduler.Schedule(b);
```

ImmediateScheduler

```
a => OnNext(42);
b => OnCompleted();
```

Which Scheduler Should You Use?

UI Applications

Observe on Dispatcher scheduler to be able to access the UI

Subscribe on TaskPool/ThreadPool for offloading computations from the UI thread

Use NewThreadScheduler for intensive computations

Back-end Applications

Use EventLoopScheduler to poll data from a data source

Use NewThreadScheduler for intensive computations

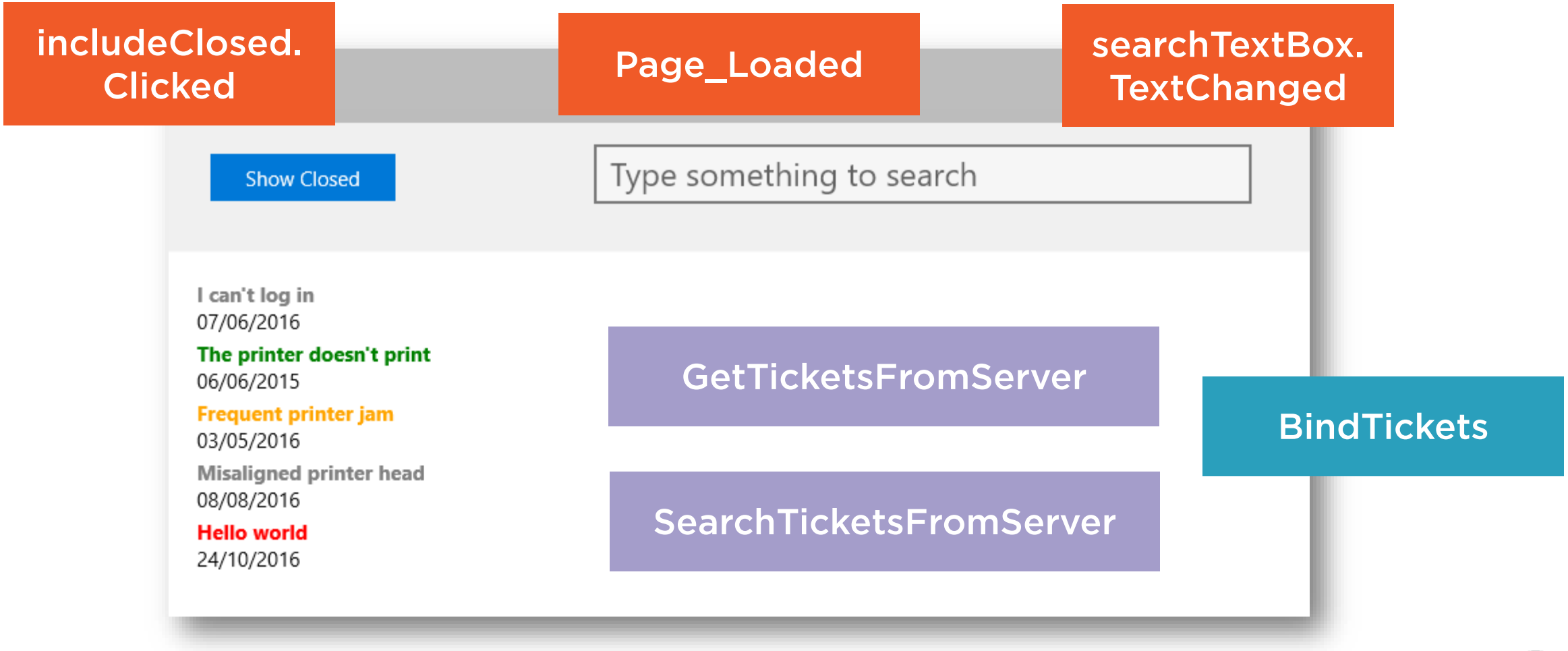
Fake Schedulers

VirtualTimeScheduler

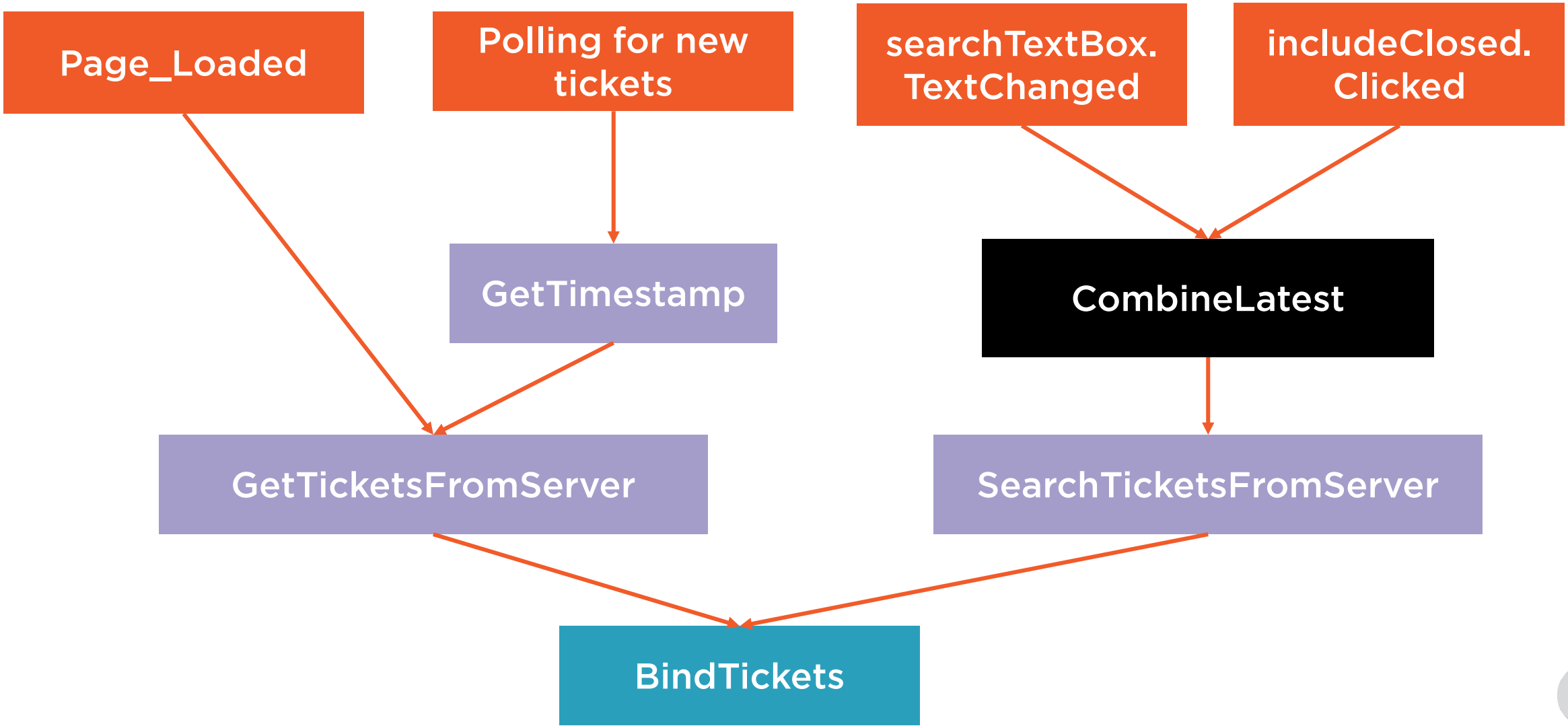
HistoricalScheduler

Wrapping Up: Reactive Thinking

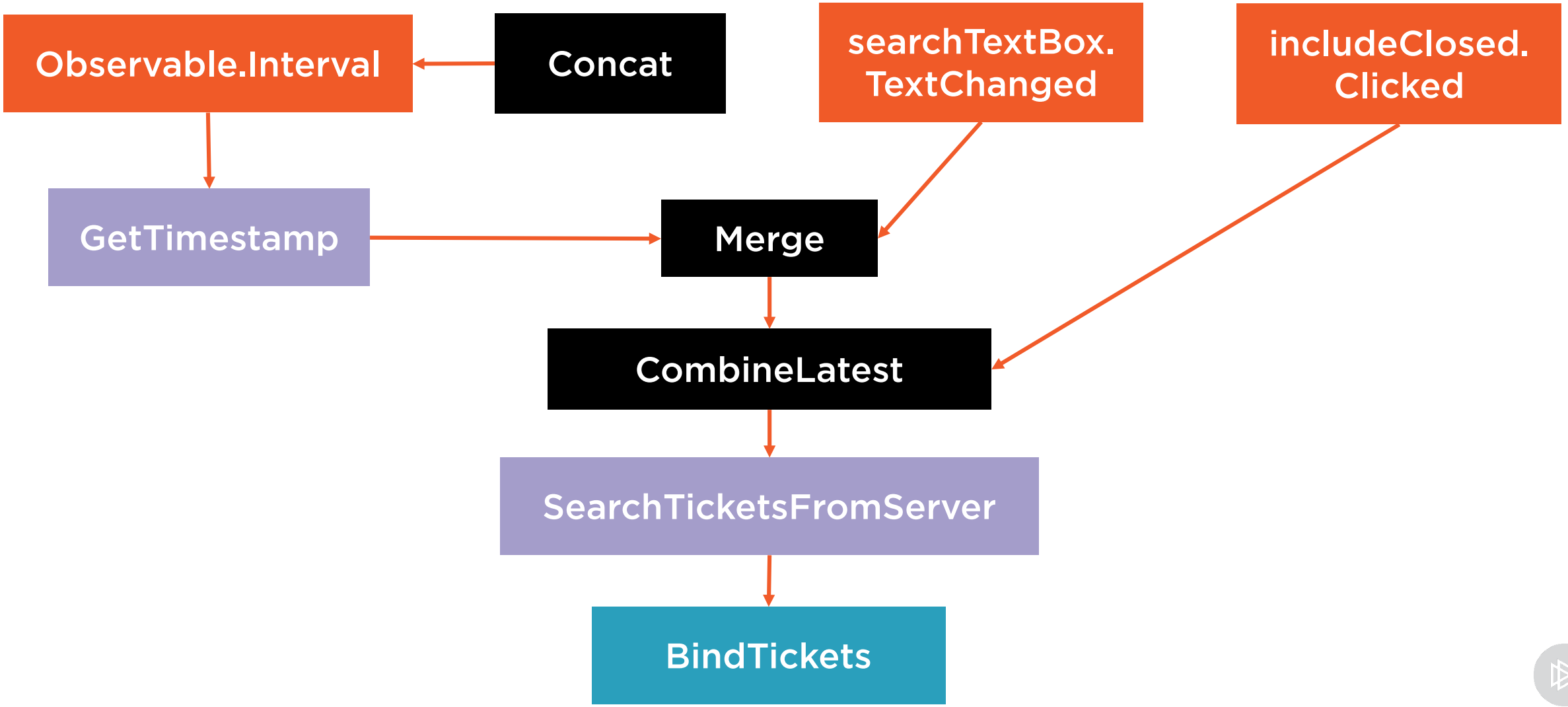
Revisiting Reactive Tickets



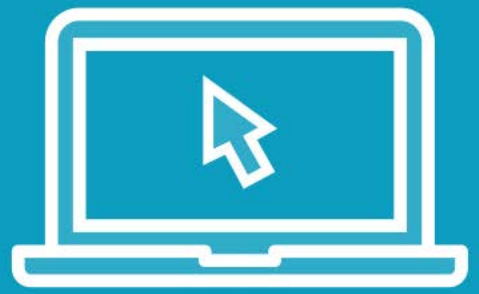
Revisiting Reactive Tickets



Revisiting Reactive Tickets



Demo



Streamlining the sequence of tickets

Polling for new tickets in a separate thread

Summary

Rx hides away the complexity of asynchrony, multithreading and concurrency with schedulers

- Many ready-made schedulers
- Virtual schedulers for testing

SubscribeOn and ObserveOn allow for simple concurrency management