REACTIVE EXTENSIONS 101
УПРАВЛЕНИЕ ХАОСОМ

Александр Роговский

SPEAKER INTRODUCTION

РОГОВСКИЙ АЛЕКСАНДР

Unity Developer
в RTS Games
Department

ВВЕДЕНИЕ

АРХИТЕКТУРА ПРИЛОЖЕНИЯ

- Общая картина, которая описывает, как все устроено внутри ПО
- У каждой программы есть архитектура, но не всякая архитектура подходит конкретному приложению
- Каждый из нас может заниматься созданием архитектуры
- Проверить правильность решений можно с помощью формулы: нам важно “X”, потому мы выбираем “Y”, принимая “Z” недостаток
- Архитектура приложения со временем меняется
- Архитекторы занимаются поиском уступок и компромиссов

ВВЕДЕНИЕ

СРАЗУ СОЗДАТЬ ИДЕАЛЬНУЮ АРХИТЕКТУРУ НЕ ПОЛУЧИТСЯ

- Не всегда есть возможность проверить тот или иной подход
- Требования к приложению могут поменяться
- Необходимо заложить определенную эластичность для изменений в архитектуре

СОДЕРЖАНИЕ

- ОСНОВНЫЕ ПРОБЛЕМЫ ПРИ НАПИСАНИИ ПРОГРАММ
- АБСТРАГИРУЕМСЯ ОТ ПРОБЛЕМ С ПОМОЩЬЮ RX
- DEMO И ВЫВОДЫ

какие проблемы может решить Rx

Как Rx решает проблемы

СОЗДАНИЕ ПРОГРАММ

ПРОБЛЕМНЫЕ МЕСТА ПРИ СОЗДАНИИ ПРОГРАММ

- **Предсказуемость кода**

  На большом проекте время, затраченное одним разработчиком на усвоение кода, может выливаться в часы работы, замедляя разработку проекта.

- **Написание бизнес логики**

  Создание и изменение бизнес логики в одном месте может вызывать проблемы в других местах. Это влечет за собой затраты на разработку, тестирование и поиск багов.

- **Обработка результатов выполнения разных операций**

  Чаще всего обработка разных операций - не тривиальная задача. При невнимательности можно допустить ошибку, поиск и исправление которой займетмного времени.

ПРЕДСКАЗУЕМОСТЬ КОДА

ПРЕДСКАЗУЕМОСТЬ КОДА

ССЫЛОЧНЫЕ ТИПЫ В .NET

Данные, переданные по ссылке в какой-то метод, могут быть изменены этим методом.
Так как вызывающий код никогда об этом не узнает, то появляется место для неявного бага.

Код, в котором изменение данных по ссылке используется намеренно,
тяжело модифицировать и расширять.

```
var data = new MyData { MyIntValue = 10 };
SomeClass.SomeMethod(data);

Console.WriteLine(data.MyIntValue);
```

ПРЕДСКАЗУЕМОСТЬ КОДА

ОДНОВРЕМЕННЫЙ ДОСТУП

Наличие многопоточности в приложении создает головную боль для разработчика. Возникают ситуации, когда разные потоки меняют одни и те же данные. Появляется место для ошибок в виде возникновения гонок (race condition), дедлоков. Рефакторинг такого когда замедляется.

```
public void Save(int publicRyan)
{
  _privateRyan = publicRyan;
  //...something something long operation...
  Console.WriteLine(_privateRyan);
}
```

ПРЕДСКАЗУЕМОСТЬ КОДА

СОСТОЯНИЕ ОБЪЕКТА

Из-за того, что объект может находится в разных состояниях, приходится добавлять в код ветвления. Наличие сложных ветвлений создает места для возможных ошибок:

- Забыли обработать недавно добавленное состояние, что влечет за собой неправильное поведение программы
- Отсутствие понимания, как изменение такого кода отразится на всей программе

```
public void DoSomething(MyData myData)
{
    if (someCondition)
    {
        if (anotherSomeCondition)
        {
            AnotherDoSomething(myData, SomeEnum.SomeEnum1,
              "hang-with-me-in-my-mmo");
        }
        else if (myData.SomeField == SomeValue)
        {
            if (myData.AnotherField != null)
                AnotherDoSomething(myData, SomeEnum.AnotherEnum,
                  "so-many-places-we-can-go");
            else
            {
                if (myData.AndAnotherField == null)
                    AnotherDoSomething(myData, SomeEnum.AndAnotherEnum,
                      "you-will-never-see-my-actual-face");
                else
                    AnotherDoSomething(myData, SomeEnum.AnotherAnother,
                      "our-love-our-love-will-be-in-virtual-space");
            }
        }
        return;
    }
}
```

ПРЕДСКАЗУЕМОСТЬ КОДА

ПРЕДСКАЗУЕМОСТЬ КОДА ВЛИЯЕТ НА СКОРОСТЬ РАЗРАБОТКИ

- Если код легко читать, разработчик быстро и без проблем усваивает неизвестный ему код программы.
- Если код легко понимать, разработчик быстро понимает, какие изменение ему нужно сделать для достижения поставленной задачи. Побочные эффекты и изменение данных не влекут за собой никаких проблем.
- Разработчик тратить 80% своего времени на чтение кода.

НАПИСАНИЕ БИЗНЕС ЛОГИКИ

НАПИСАНИЕ БИЗНЕС ЛОГИКИ

НАПИСАНИЕ И ИЗМЕНЕНИЕ БИЗНЕС ЛОГИКИ МОЖЕТ ВЫЗЫВАТЬ БОЛЬ И СТРАДАНИЕ

- Возникают трудности, когда мы должны ждать, пока сработает определенное событие.
- Бизнес логика может быть странной, что порождает странный код
- Очень тяжело оценить то, как программа себя поведет после изменений.

```
SomeClass.SomeMethod(() =>
{
    Callback(() =>
    {
        Callback(() =>
        {
            Callback(() =>
            {
                Callback(() =>
                {
                    Callback(() =>
                    {
                        Callback(() =>
                        {
                            Callback(() =>
                            {
                                Callback(() => { Console.WriteLine("★"); });
                            });
                        });
                    });
                });
            });
        });
    });
});
```

НАПИСАНИЕ БИЗНЕС ЛОГИКИ

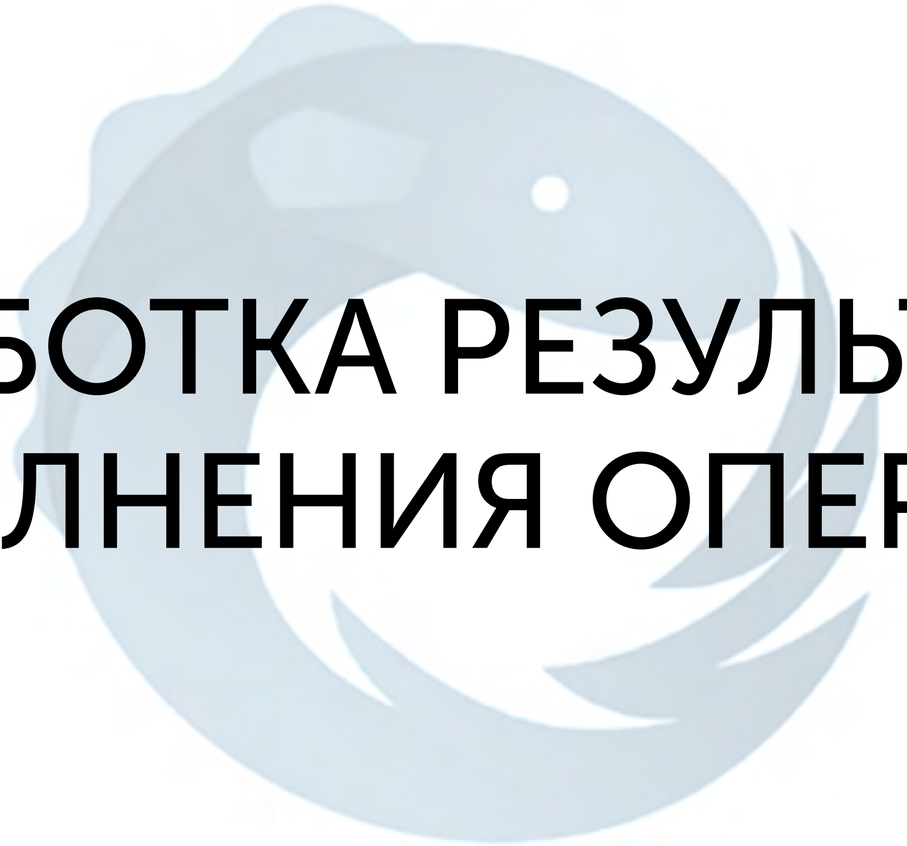
ЧЕГО ХОТЕЛОСЬ БЫ ПРИ РАБОТЕ С БИЗНЕС ЛОГИКОЙ

- Простота **реализации** бизнес логики.

  Разработчик не создает потенциальных мест для совершения ошибок, добавляет новую бизнес логику путем ввода новых правил для обработки данных.

- Простота **модификации** бизнес логики.

  Изменение логики поведения программы в одном месте не влияет на поведение программы в другом.



ОБРАБОТКА РЕЗУЛЬТАТОВ ВЫПОЛНЕНИЯ ОПЕРАЦИЙ

СИНХРОНИЗАЦИЯ

ЧЕГО ХОТЕЛОСЬ БЫ

- Простота в **управлении потоками.**

  Унифицированный подход при работе с потоками.

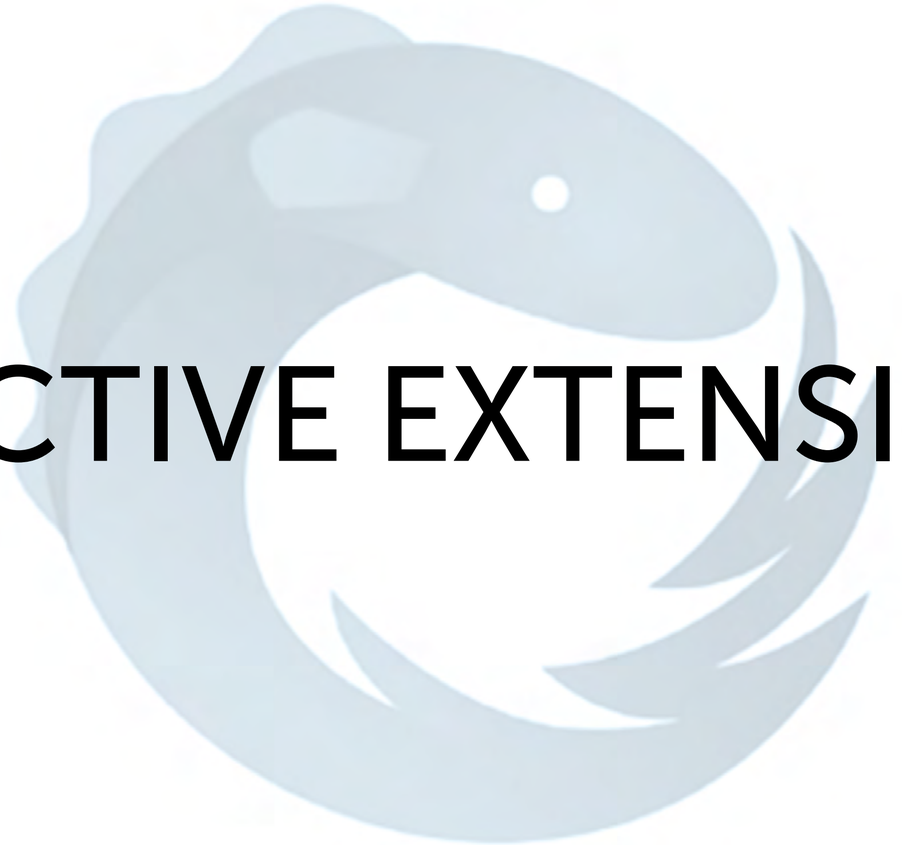
- Удобство в **композиции результатов** нескольких операций.

  Объединение результатов нескольких операций не должно порождать потенциальных мест для багов.

ИТОГИ ХОТЕЛОК

ВЕЩИ, КОТОРЫЕ МЫ ХОТИМ ИМЕТЬ ПОСЛЕ ДВУХ-ТРЕХ ЛЕТ РАЗРАБОТКИ ПРОЕКТА:

- Код **легко читать**
- Код **легко понимать**
- Простота **реализации** бизнес логики
- Простота **модификации** бизнес логики
- Простота в **управлении потоками**
- Удобство в **композиции результатов** нескольких операций
- Нет поводов для стресса



REACTIVE EXTENSIONS

REACTIVE EXTENSIONS

ТЕХНОЛОГИЯ REACTIVE EXTENSIONS СОСТОИТ ИЗ ТРЕХ КОМПОНЕНТОВ

- **Observer pattern** предполагает наличие и позволяет объединить две сущности: поставщик данных и потребитель данных.
- **Linq-подобные команды** позволяют добавлять и модифицировать бизнес логику не затрагивая другие места в проекте.
- **Планировщики** упрощают работу с потоками.

REACTIVE EXTENSIONS

ПОЧЕМУ В RX НЕ ИСПОЛЬЗУЮТСЯ СТАНДАРТНЫЕ СОБЫТИЯ

- Тяжело обрабатывать срабатывания нескольких событий
- События не предоставляют никакой возможности для обращения к данным через некоторое время
- События - самое частое место для создания утечек памяти
- У событий нет встроенных механизмов сигнализирования о завершении потока данных
- События почти не предоставляют никакой помощи при написании многопоточных приложений

REACTIVE EXTENSIONS

ДЕКЛАРАТИВНЫЙ ПОДХОД

Декларативный подход состоит в том, что мы говорим программе только результат, который мы хотим получить, а как именно это произойдет - нас не сильно интересует.
Достигается это путем применения функций (linq-подобных операторов) к данным (потоку данных).
С помощью RX операторов можно очень просто писать и модифицировать бизнес логику.
Каждый оператор является маленьким черным ящиком, который знает как правильно получить данные, как трансформировать их под свои нужды и передать дальше. Из-за такого подхода мы не пишем один и тот же код снова и снова (т.е. переиспользуем уже существующий).
Это помогает увеличить плотность полезной информации в коде.

Плюсы:

- код быстрее читать и легче понимать
- меньше шанс допустить ошибку
- устранение лишних зависимостей

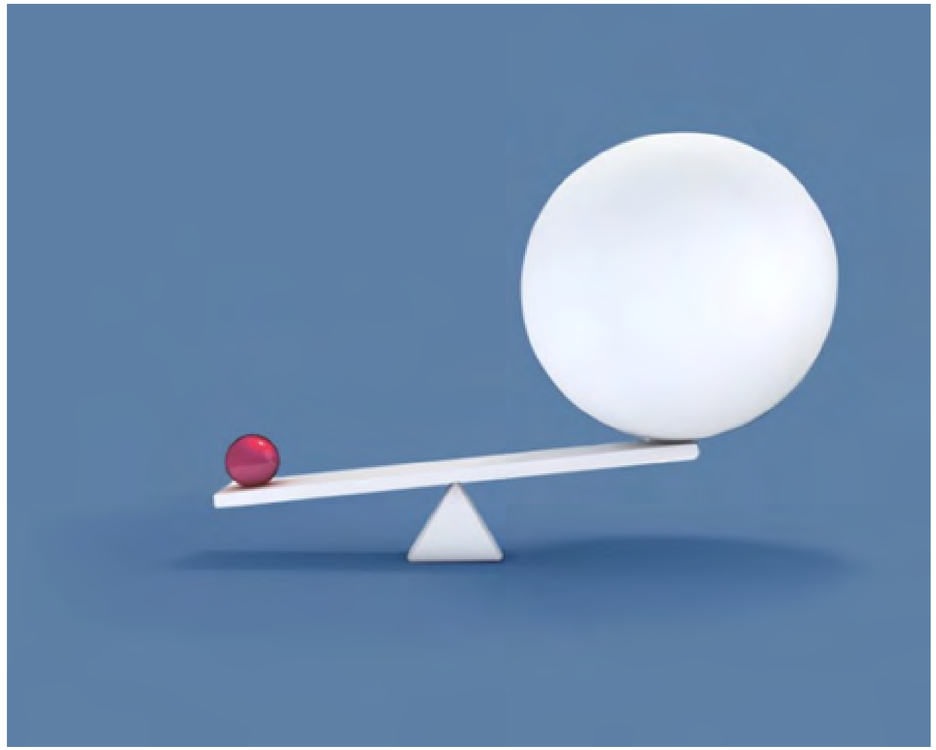
REACTIVE EXTENSIONS

МЕНЬШЕ КОДА - МЕНЬШЕ ПРОБЛЕМ

Из-за того, что в Rx операторах уже реализовано большинство стандартных операций, разработчику не приходится при необходимости писать свою реализацию, он берет уже существующую. Операторы Rx очень часто превращают задачи, которые раньше было непонятно как реализовать, в несколько строк кода.

Плюсы:

- переиспользование уже протестированного и отлаженного кода



REACTIVE EXTENSIONS

УДОБНАЯ ОБРАБОТКА ОШИБОК

При работе с асинхронными потоками данных стандартные методы обработки исключений (try/catch) чаще всего не подходят. При использовании потока данных разработчик помимо потребителя данных указывает потребителя ошибок.

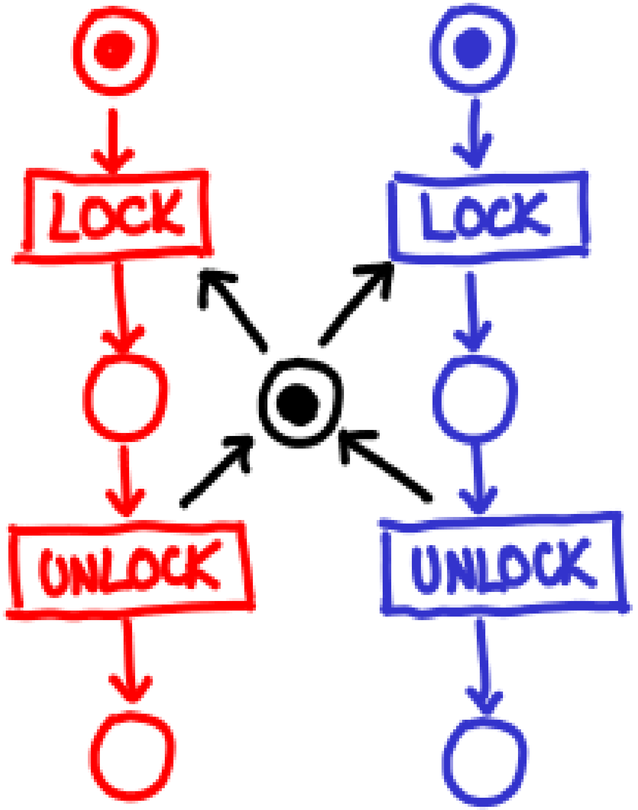
Плюсы:

- отсутствие неочевидных мест возникновения ошибок
- обработка в одном месте, в одном стиле

REACTIVE EXTENSIONS

ПАРАЛЛЕЛИЗМ ОДНОЙ ЛЕВОЙ

С помощью планировщиков в RX создали абстракцию, которая позволяет отгородиться от низкоуровневой работы с потоками. Разработчику необходимо один раз описать все возможные варианты обработки данных с помощью потоков. Остальное время при работе с потоками надо указывать тот или иной ранее написанный вариант.



Плюсы:

- меньше шанс допустить ошибку
- переиспользование существующих моделей поведения

КОМПОНЕНТЫ REACTIVE EXTENSIONS

КОМПОНЕНТЫ RX

Основу работы Rx составляет подход, при котором мы разделяем данные и функции обработки данных. Поставщик данных реализовывает интерфейс IObservable<T>. Интерфейс обязывает реализовать метод Subscribe, с помощью которого все желающие могут подписаться на появление новых данных. Подписчики обязаны реализовать интерфейс IObserver<T>, который обязывает реализовать три метода:

- метод обработки данных (OnNext)
- метод обработки исключений (OnError)
- метод обработки окончания потока (OnCompleted)

```
public interface IObservable<T>
{
    IDisposable Subscribe(IObserver<T> observer);
}
public interface IObserver<T>
{
    void OnCompleted();
    void OnError(Exception error);
    void OnNext(T value);
}
```

КОМПОНЕНТЫ RX

ПОТОКИ ДАННЫХ ДЕЛЯТСЯ НА ДВА ТИПА

- **холодный (cold)**
  Поток начинает продуцировать элементы только после того, как кто-то подписался. Инициализация происходит для каждого подписчика отдельно
- **горячий (hot)**
  Поток начинает продуцировать элементы сразу после создания, независимо от того, кто подписался. Инициализация происходит один раз при создании

КОМПОНЕНТЫ RX

Поток данных создается при помощи различных стандартных методов. После этого все желающие могут подписаться, передав объект, которые реализовывает интерфейс с тремя методами: OnNext, OnError, OnComplete. Чаще всего перегрузки метода Subscribe позволяют указывать лямбду только для метода OnNext.

```
IObservable<int> source = /* Some source */;
IDisposable subscription = source.Subscribe(
    x  => Console.WriteLine("OnNext:  {0}", x),
    ex => Console.WriteLine("OnError: {0}", ex.Message),
    () => Console.WriteLine("OnCompleted")
);
```

КОМПОНЕНТЫ RX

Самый простой способ создать поток данных - метод расширения Observable.Create. Он принимает лямбду, которая вызывается при подписке. Лямбда создаст элементы для потока и сообщит о них всех подписчиков.
Лямбда должна возвращать IDisposable. Таким образом подписчик сможет явно вызвать Dispose и заставить поток данных освободить ресурсы. В данном примере возвращается заглушка.

```
IObservable<int> source = Observable.Create<int>(observer =>
{
    observer.OnNext(1);
    observer.OnNext(2);
    observer.OnNext(444);
    observer.OnNext(4);
    observer.OnNext(-4);
    observer.OnNext(105);
    observer.OnCompleted();
    return Disposable.Empty;
});
```

КОМПОНЕНТЫ RX

Для облегчения работы с данными уже создано множество базовых методов. Один из них - Empty. Поток данных сообщит с помощью вызова метода OnCompleted о том, что элементы закончились и будет таков.

```
IObservable<int> source = Observable.Empty<int>();
```

RESULT:

OnCompleted

КОМПОНЕНТЫ RX

Чаще всего базовые операторы могут быть использованы как заглушки или для тестов. Например, метод Throw создает поток данных, который сообщает всех подписчиков об ошибке.

```
IObservable<int> source = Observable.Throw<int>(new Exception("Oops"));
```

RESULT:

OnError: Oops

КОМПОНЕНТЫ RX

Если необходимо вернуть одно значение, то для этого подойдет оператор Return. Поток вызовет OnNext для всех подписчиков с тем значением, которое вы укажете, после чего также уведомит всех, что элементы закончились при помощи вызова OnCompleted.

```
IObservable<int> source = Observable.Return(42);
```

RESULT:

OnNext: 42

OnCompleted

КОМПОНЕНТЫ RX

Другой пример - метод Range. Принимает два параметра: откуда начинать и сколько вернуть элементов. Отлично подходит для генерации потоков данных на лету.

```
IObservable<int> source = Observable.Range(5, 3);
```

RESULT:

OnNext: 5

OnNext: 6

OnNext: 7

OnCompleted

КОМПОНЕНТЫ RX

В качестве заглушки отлично подойдет поток данных, который мы может создать с помощью метода Never. Этот поток данных никогда ничего не сообщит своим подписчикам. Silence is gold!

```
IObservable<int> source = Observable.Never<int>();
```

RESULT:

КОМПОНЕНТЫ RX

У нас есть поток данных и мы хотим применить к нему какую-то бизнес логику. Например, нам нужно обработать все элементы больше 100. Для этой цели отлично подойдет оператор Where. После этого мы для удобства добавляем время срабатывания события при помощи оператора Timestamp и подписываемся на поток. При появлении элемента больше 100 в потоке, мы выводим информацию о нем в консоль.

```
source
    .Where(i => i > 100)
    .Timestamp()
    .Subscribe(d =>
    {
        Console.WriteLine("Time: '{0}', Value: '{1}'.", d.Timestamp, d.Value);
    });
```

```
Time: '22.08.2017 14:40:38 +00:00', Value: '123152'.
Time: '22.08.2017 14:40:38 +00:00', Value: '5234'.
Time: '22.08.2017 14:40:38 +00:00', Value: '444'.
Time: '22.08.2017 14:40:38 +00:00', Value: '105'.
```

КОМПОНЕНТЫ RX

Представим, что мы собираем данные о температуре. Данные с датчика поступают каждые десять секунд. Но меняются они гораздо реже. Чтобы не нагружать нашу программу, мы не будем реагировать на новые данные до тех пор, пока температура не изменится. Сделать это можно при помощи оператора DistinctUntilChanged. Теперь каждое следующее значение будет отличатся от предыдущего.
Основной поток не меняется. Кому надо, тот будет получать обновления каждые десять секунд.

```
source
  .Where(i => i > 100)
  .DistinctUntilChanged()
  .Timestamp()
  .Subscribe(d =>
  {
    Console.WriteLine("Time: '{0}', Value: '{1}'.", d.Timestamp, d.Value);
  });
```

```
Time: '22.08.2017 14:49:20 +00:00', Value: '105'.
Time: '22.08.2017 14:49:20 +00:00', Value: '195'.
_
```

КОМПОНЕНТЫ RX

Представим, что мы собираем данные о температуре. Задача перед нами: знать о всех изменениях температуры, которые становятся новым максимумом. Сделать это можно при помощи оператора Scan. Оператор хранит текущее состояние. При появлении в потоке нового элемента, оператор применяет к нему функцию, которую мы передадим вторым параметром. Функция принимает новое значение и текущее состояние. Изначальное состояние - int.MinValue. Результат выполнения функции сохраняется как текущий результат.

```
source
  .Where(i => i > 100)
  .Scan(int.MinValue, Math.Max)
  .DistinctUntilChanged()
  .Timestamp()
  .Subscribe(d =>
  {
    Console.WriteLine("Time: '{0}', Value: '{1}'.", d.Timestamp, d.Value);
  });
```

```
Time: '22.08.2017 14:57:11 +00:00', Value: '105'.
Time: '22.08.2017 14:57:11 +00:00', Value: '195'.
Time: '22.08.2017 14:57:11 +00:00', Value: '200'.
```

КОМПОНЕНТЫ RX

ПЛАНИРОВЩИКИ

Rx, умолчанию, однопоточный. Однако Rx предоставляет абстракцию, которая позволяют работать с многопоточностью в декларативной стиле, а также подвергать тестированию.
В Rx нам может понадобится многопоточность в двух случаях:

- при подписке на поток (инициализация потока данных)
- при обработке новых данных

То, какой использовать поток, определяет специальный класс, реализующий интерфейс IScheduler. В комплекте с Rx идут стандартные планировщики, но вы можете написать и использовать свой.

```
public interface IScheduler
{
    DateTimeOffset Now { get; }

    IDisposable Schedule<TState>(TState state, Func<IScheduler, TState, IDisposable> action);
    IDisposable Schedule<TState>(TState state, TimeSpan dueTime, Func<IScheduler, TState, IDisposable> action);
    IDisposable Schedule<TState>(TState state, DateTimeOffset dueTime, Func<IScheduler, TState, IDisposable> action);
}
```

КОМПОНЕНТЫ RX

Планировщик, который управляет инициализацией, задается с помощью метода расширения SubscribeOn. Например, чуть раньше мы обсуждали метод Observable.Create. Внутрь этого метода мы передаем лямбду. Лямбда порождает новые элементы. Если мы вызовем SubscribeOn, то явно укажем, какой планировщик использовать при подписке.
Для указания планировщика по обработке новых элементов используется метод ObserveOn.

```
public static class Observable
{
  public static IObservable<TSource> ObserveOn<TSource>(
    this IObservable<TSource> source,
    IScheduler scheduler)
  {...}
  public static IObservable<TSource> SubscribeOn<TSource>(
    this IObservable<TSource> source,
    IScheduler scheduler)
  {...}
}
```

КОМПОНЕНТЫ RX

Независимость операторов Rx друг от друга позволяет нам без проблем включать вызовы SubscribeOn и ObserveOn в цепочку. Если эти операторы не вызвать, то будут использованы стандартные значения. Если это возможно, то использование планировщиков лучше избегать. Нам дали в руки инструмент для упрощения работы с многопоточностью. Однако многопоточность по прежнему не тривиальная проблема и является местом, где легко допустить ошибку.

```
source
  .Where(i => i > 100)
  .Scan(int.MinValue, Math.Max)
  .DistinctUntilChanged()
  .Timestamp()
  .SubscribeOn(Scheduler.Immediate)
  .ObserveOn(Scheduler.NewThread)
  .Subscribe(d =>
  {
    Console.WriteLine("Time: '{0}', Value: '{1}'.", d.Timestamp, d.Value);
  });
```

DEMO

ВЫВОДЫ

ЧЕГО МЫ ХОТЕЛИ:

- Код **легко читать**
- Код **легко понимать**

КАК МЫ ЭТОГО ДОБИВАЕМСЯ

- Отсутствие состояний
- Использование отдельных операторов

```
source
  .Select(d => d.MyIntValue)
  .Where(i => i > 100)
  .DistinctUntilChanged()
  .Timestamp()
  .Subscribe(d =>
  {
    Console.WriteLine("Time: '{0}', Value: '{1}'.", d.Timestamp, d.Value);
  });
```

ВЫВОДЫ

ЧЕГО МЫ ХОТЕЛИ:

- Простота **реализации** бизнес логики
- Простота **модификации** бизнес логики

КАК МЫ ЭТОГО ДОБИВАЕМСЯ

- Декларативный подход
- Переиспользование существующего кода

```
source
  .Where(i => i > 100)
  .Scan(int.MinValue, Math.Max)
  .DistinctUntilChanged()
  .Timestamp()
  .Subscribe(d =>
  {
    Console.WriteLine("Time: '{0}', Value: '{1}'.", d.Timestamp, d.Value);
  });
```

ВЫВОДЫ

ЧЕГО МЫ ХОТЕЛИ:

- Простота в **управлении потоками**
- Удобство в **композиции результатов** нескольких операций

```
source
  .Where(i => i > 100)
  .Scan(int.MinValue, Math.Max)
  .DistinctUntilChanged()
  .Timestamp()
  .SubscribeOn(Scheduler.Immediate)
  .ObserveOn(Scheduler.NewThread)
  .Subscribe(d =>
  {
    Console.WriteLine("Time: '{0}', Value: '{1}'.", d.Timestamp, d.Value);
  });
```

*Добавление абстракции в виде планировщиков и методов SubscribeOn и ObserveOn

КАК МЫ ЭТОГО ДОБИВАЕМСЯ

- Добавление уровня абстракции
- Объединение нескольких потоков в один

```
var source1 = Observable.Return("Press Enter to return.");
var source2 = Observable.Create<string>(observer =>
{
  Thread.Sleep(1000);
  observer.OnNext("I'm Mr Meeseeks look at me!");
  return Disposable.Create(() => Console.WriteLine("All done!"));
});

var mergedSource = source1.Merge(source2);

mergedSource.Subscribe(someString =>
{
  Console.WriteLine("Just received some string: '{0}'.", someString);
});
```

*Объединение нескольких потоков в один

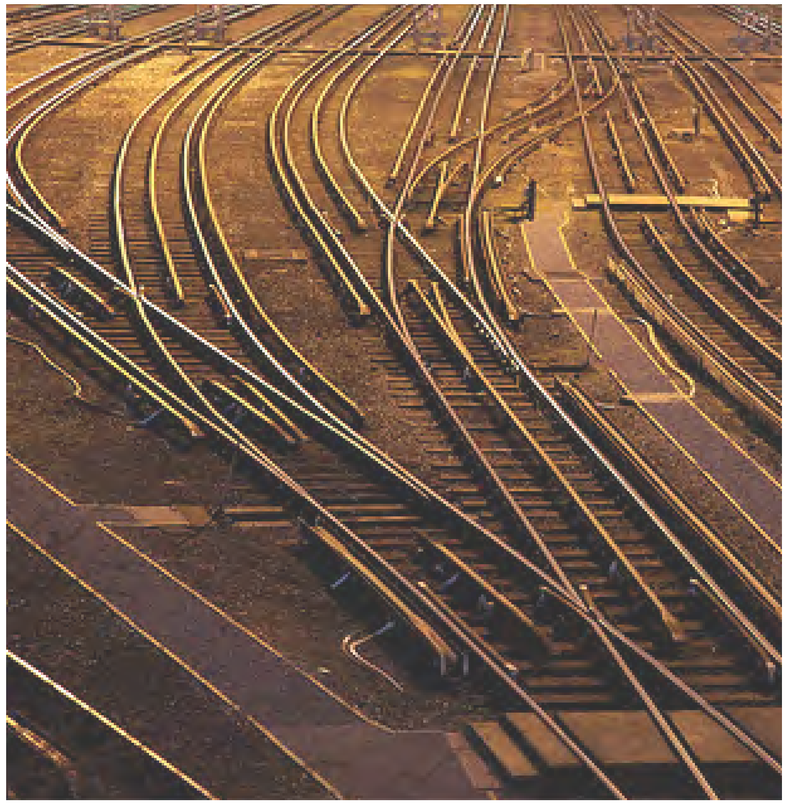
ВЫВОДЫ

ОДНА ИЗ ГЛАВНЫХ ЗАДАЧ ПРИ ИСПОЛЬЗОВАНИИ RX - ДОБИТЬСЯ ПРОЗРАЧНОСТИ:

- ошибки не выскакивают в неожиданных местах
- использование многопоточности происходит в одном стиле
- отсутствие неявных зависимостей

КАК RX ДОБИВАЕТСЯ ПРОЗРАЧНОСТИ:

- разделение понятий данные и функции обработки данных
- переиспользование уже существующего кода (декларативный подход)
- устранение работы с состояниями (если это возможно)
- четкое разделение ответственности операторов



ПЛЮСЫ

- **Асинхронный код**
  По умолчанию, работа с Rx предполагает полную асинхронность. Потоки не блокируется для того, чтобы подождать результата выполнения той или иной операции.
  Но если вдруг вам надо будет блокировать выполнение потока до определенного момента, то у вас будет такая возможность, но это будет каким-то исключением, которое применяется в крайнем случае. А по умолчанию - полная асинхронность.
- **Дополнительный уровень абстракции**
  Работая в среде, где надо постоянно в голове держать кучу деталей, очень легко допустить ошибку. Мы добавляем уровень абстракции, чтобы лишить программиста (или по крайней мере уменьшить возможность) совершить ошибку.
- **Простота комбинирования результатов**
  Объединение потоков - одна из самых мощных техник. Она позволяет упростить, ускорить разработку и допускать меньше ошибок.
- **Удобство обработки событий**
  Появляется возможность получать, фильтровать, обрабатывать события: во первых - асинхронно, во вторых - в одном стиле, в третьих - без опасений сломать функционал в других местах программы.
- **Корректная обработка ошибок и отмена операций**
  Для каждого отдельного случая нам не надо придумывать механизм обработки исключений и отмены операций. Все работает однообразно, везде можно задать, что делать в случае ошибки, или отменить операцию.

МИНУСЫ

- **Тяжелая кривая обучения**

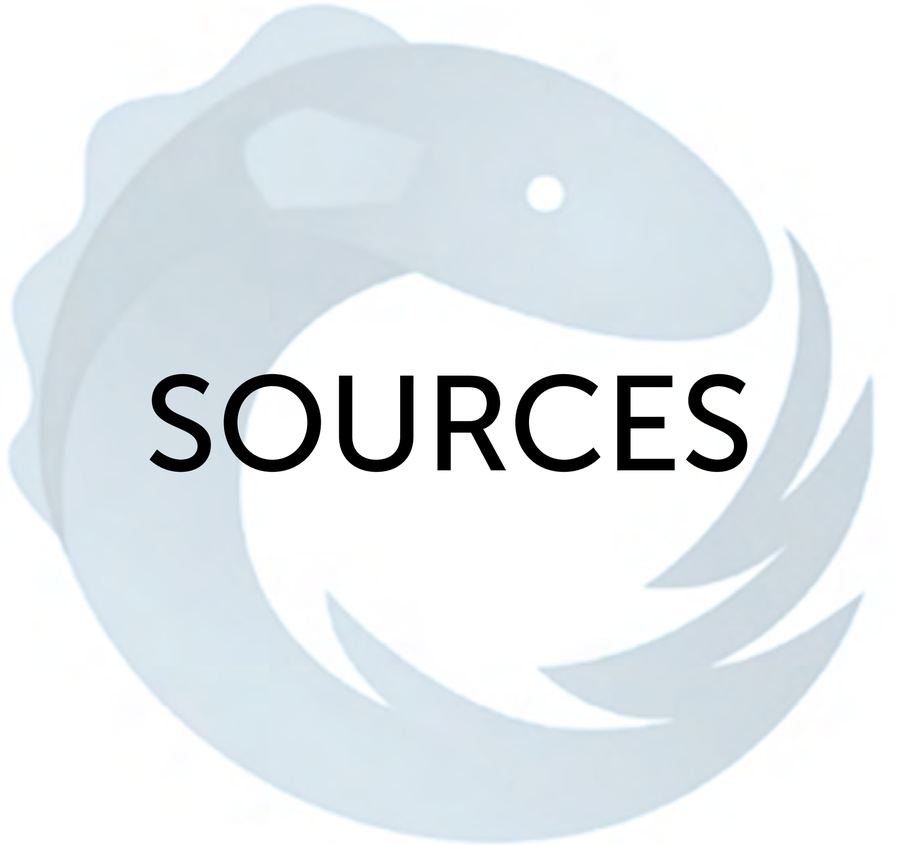
  Людям, которые раньше не встречались с такими понятиями, как immutability, разделение данных и функций, чистые функции, отсутствие побочных эффектов требуется ~два месяца на то, чтобы изучить, принять и начать использовать Rx

- **Проблема параллелизма не решается полностью**

  Разработчики получают в руки мощный инструмент, однако сам по себе он ничего не делает

- **Относительно тяжело тестировать**

  Асинхронные потоки данных тяжело тестировать



SOURCES

http://reactivex.io

Описание всех существующих операторов. Присутствует дерево решений

http://rxmarbles.com

Интерактивная песочница. Выбираете оператор, меняете положение входящих данных и смотрите на результат

http://introtorx.com

Интернет-книга, в которой рассматриваются все аспекты Rx с примерами

https://github.com/Reactive-Extensions/Rx.NET

Исходный код Rx. Присутствуют примеры.

THANK YOU!

QUESTIONS?

www.developers.plarium.com