



Паттерны функционального программирования для C# разработчиков

НИКОЛАЙ ГУСЕВ
mailto:nikolay.gusev@db.com



Релизная версия
после года разработки

Реально хорошо написанная

Что будет с ней после
еще пары релизов?

Функциональное программирование





О чем поговорим

- Проблемы ООП языков
- Полезные приемы из ФП



Теория



Практика





Не обязательно
бросаться в омут с
головой

Все сегодняшние
темы полезны как
вместе, так и по
отдельности



Преимущества описанных методик



- Код проще понимать и поддерживать
- Компилятор становится лучшим другом
- Меньше багов





HOLIDAYWORLD.COM





Часть I

Типичные проблемы и варианты их решения с помощью ФП



I. Типичные проблемы и варианты их решения с помощью ФП

- Когда наследования недостаточно - Discriminated Unions
- NRE и как с ним бороться - Option<T>
- Альтернатива исключениям - Result<T>



1. Discriminated Unions



Type switching - пример

```
public double GetArea(Shape shape)
{
    var r = shape as Rectangle;
    if (r != null) return r.Height * r.Width;

    var c = shape as Circle;
    if (c != null) return Math.PI * c.Radius * c.Radius;

    throw new NotSupportedException();
}
```





Type switching – C# 7

```
public double GetArea(Shape shape)
{
    switch (shape)
    {
        case Rectangle r:
            return r.Height * r.Width;
        case Circle c:
            return Math.PI * c.Radius * c.Radius;
        default:
            throw new NotSupportedException();
    }
}
```



Type switching – наследование



Но ведь можно метод `GetArea()` сделать абстрактным методом класса `Shape`





Недостатки решения через наследование

- Лишние зависимости в классе
- У клиентов нет возможности добавлять новые методы



Discriminated unions



Discriminated Unions

- Enum-like значения
- Содержат разные данные для разных кейсов
- Нельзя расширять набор кейсов извне
- Исчерпывающий matching с проверкой компилятором



Discriminated Unions – F#

```
type Shape =  
| Rectangle of height : double * width : double  
| Circle    of radius : double
```



Discriminated Unions – F#

```
let getArea (shape: Shape) : double =  
    match shape with  
    | Rectangle(height, width) -> height * width  
    | Circle (radius) -> Math.Pi * radius * radius
```



No exception



Discriminated Unions – F#

```
type Shape =  
| Rectangle of height : double * width : double  
| Circle of radius : double  
| Triangle of side : double
```


Discriminated Unions – F#



```
let getArea (shape: Shape) =  
    match shape with  
    | Rectangle(height, width) -> height * width  
    | Circle (radius) -> Math.Pi * radius * radius
```



FS0025 Незавершенный шаблон соответствует данному выражению. К примеру, значение "Triangle (_)" может указывать на случай, не покрытый шаблоном(ами).





Окей, как можно было бы это использовать в C#?

Используем автогенерацию кода!



Discriminated Unions – описываем кейсы

```
[UnionBase] ← Marker Attribute
public abstract partial class Shape { }

public partial class Rectangle : Shape
{
    public double Height { get; }
    public double Width { get; }
}

public partial class Circle : Shape
{
    public double Radius { get; }
}
```

Описав возможные кейсы, запускаем кодогенерацию и переходим к ...



Discriminated Unions – использование Match

```
public static double GetArea(Shape shape)
=>shape.Match(rectangle: r => r.Height * r.Width,
              circle: c => Math.Pi * c.Radius * c.Radius);
```

Автосгенерированный
метод

typeof(Rectangle)
typeof(Circle)

Именованные параметры

Discriminated Unions – C#



```
[UnionBase]  
public abstract partial class Shape { }
```

```
public partial class Triangle : Shape  
{  
    public double Side { get; }  
}
```

```
public partial class Rectangle : Shape  
{  
    public double Height { get; }  
    public double Width { get; }  
}
```

```
public partial class Circle : Shape  
{  
    public double Radius { get; }  
}
```

Запускаем кодогенерацию и пытаемся
скомпилировать...



Discriminated Unions – C#

```
public static double GetArea(Shape shape)
=>shape.Match(rectangle: r => r.Height * r.Width,
              circle:    c => Math.Pi * c.Radius * c.Radius);
```



CS7036 Отсутствует аргумент, соответствующий
требуемому формальному параметру "triangle"...



Discriminated Unions – за кулисами



```
private String name = ...  
String[] arrayname = ...  
String[] singlename = singlename.replaceAll("\\s+", " ");  
for (String singlename = singlename.replaceAll("\\s+", " "); singlename != null; singlename = singlename.replaceAll("\\s+", " ")) {  
    String[] settings = singlename.split("\\s+");  
    if (settings[0].compareTo("") == 0) {  
        if (name.compareTo("") != 0) {  
            name += " ";  
        }  
        name += settings[0];  
    } else if (settings[0].compareTo("d") == 0) {  
        if (name.compareTo("") != 0) {  
            name += " ";  
        }  
        name += DateUtils.format(etr.getDate(settings[1]));  
    } else if (settings[0].compareTo("n") == 0) {  
        if (name.compareTo("") != 0) {  
            name += " ";  
        }  
        name += DoubleFormat.format(etr.getDouble(settings[1]));  
    }  
}
```

Discriminated Unions – за кулисами



// Авто-сгенерированный код

```
public abstract partial class Shape
{
    public T Match<T>(Func<Rectangle, T> rectangle,
                    Func<Circle, T> circle)
    {
        var r = this as Rectangle;
        if (r != null) return rectangle(r);

        var c = this as Circle;
        if (c != null) return circle(c);

        // Недостижимый код!
        throw new NotSupportedException();
    }

    internal abstract void Seal(); // Не даем наследоваться от Shape.
                                   // Кейсы закрываем через
    ...                           // “sealed partial class Rectangle/Circle”
}
```

Discriminated Unions – за кулисами



```
// Авто-сгенерированный код
public abstract partial class Shape
{
    ...

    public void Do(Action<Rectangle> rectangle,
                  Action<Circle> circle)
    {
        var r = shape as Rectangle;
        if (r != null) {rectangle(r); return;}

        var c = shape as Circle;
        if (c != null) {circle(c); return;}

        // Недостижимый код!
        throw new NotSupportedException();
    }
    ...
}
```

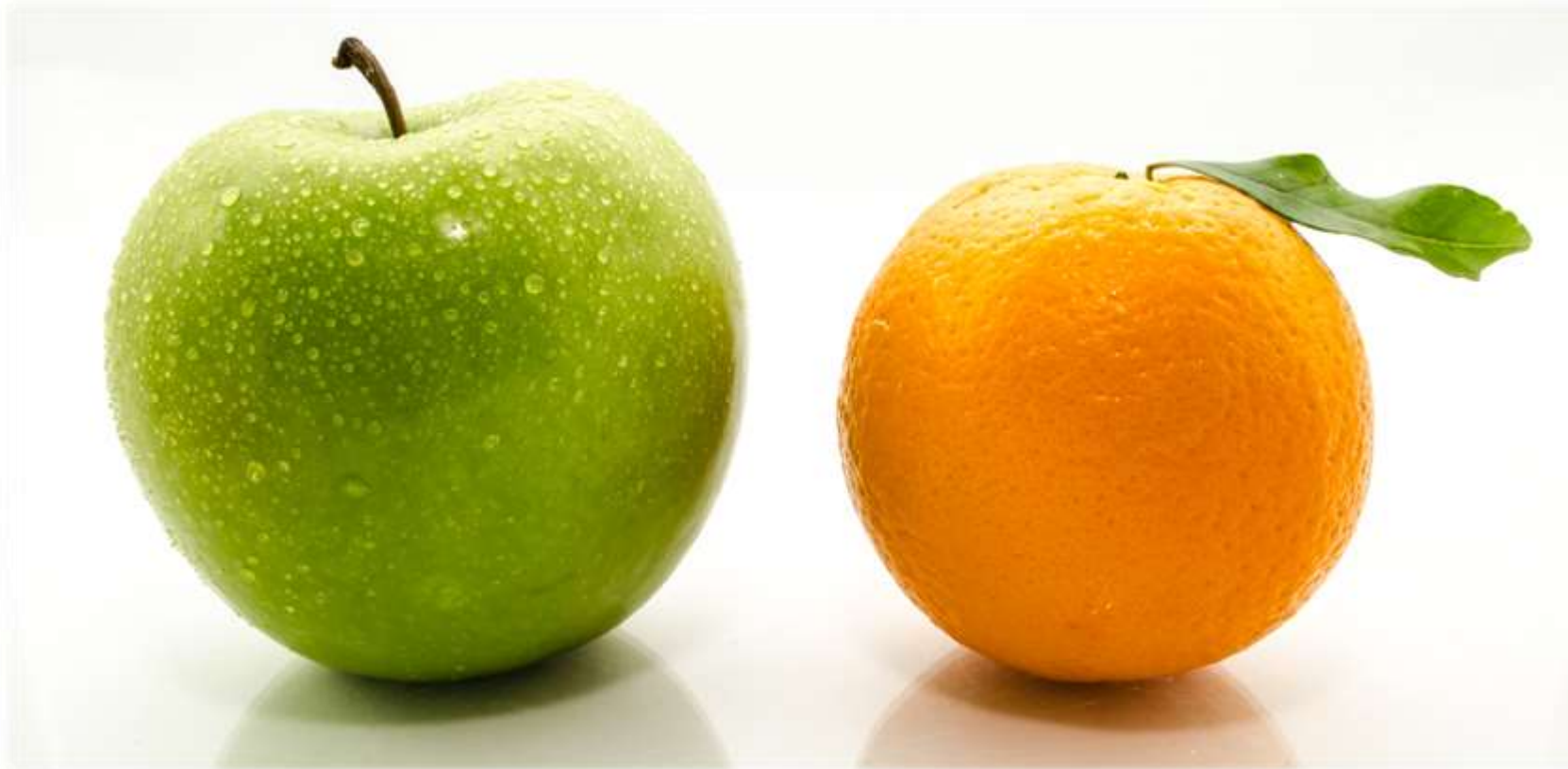


Discriminated Unions – за кулисами

Что еще генерируется автоматически:

- Конструкторы Rectangle, Circle
- Статические конструкторы Shape.Rectangle, Shape.Circle
- Equals, GetHashCode, ==, !=

Наследование или Discriminated Unions?

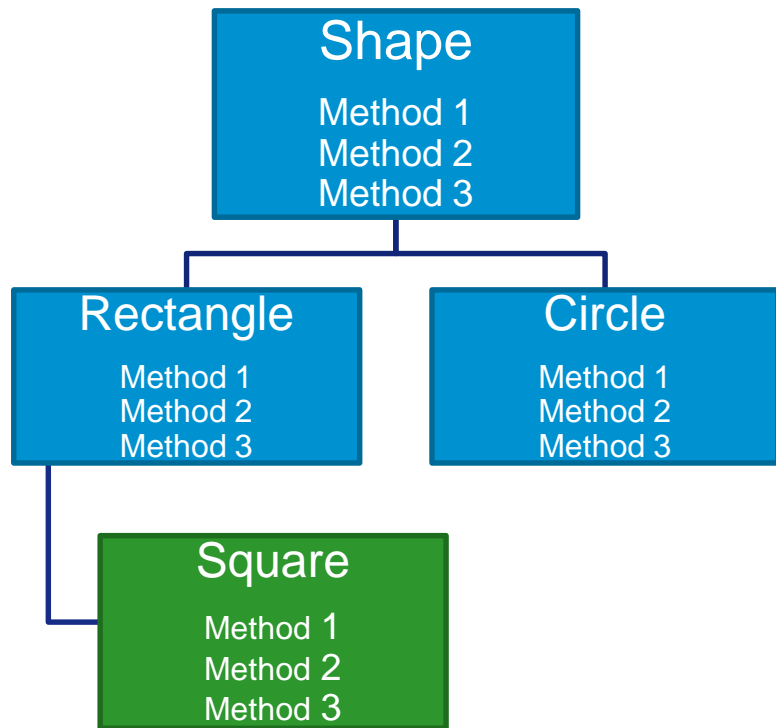




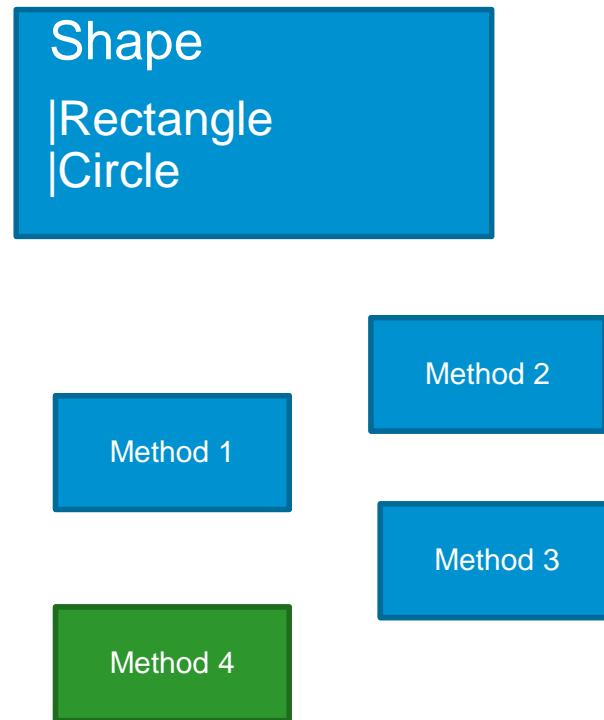
Discriminated Unions и Наследование дополняют друг друга

	Discriminated Unions	Наследование
<i>Легко добавить</i>	Новые методы	Новые типы
<i>Тяжело добавить</i>	Новые типы	Новые методы

Наследование



Discriminated Unions






Языки, поддерживающие Discriminated Unions (aka Algebraic Data Types)

- F#
- Haskell
- Scala
- TypeScript (2.0)
- Swift
- Rust
- Nemerle
- Kotlin
- ...



Discriminated Unions – итоги

- Switch по типам
- Проверка компилятором покрытия всех кейсов
- Composition over Inheritance 
- Замена иерархиям
- Иногда наследование подходит больше



2. Null значения

Я привожу к каждой
второй ошибке в ваших
приложениях.
Можно зайду?



Null значения



"I call it my **billion-dollar mistake**. It was the invention of the null reference in 1965...My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, **simply because it was so easy to implement**. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably **caused a billion dollars of pain and damage in the last forty years.**"



- Сэр Чарльз Энтони Ричард Хоар



Null значения - недостатки

- Может ли аргумент, поле или свойство принимать значение null
- Компилятор не находит обращения к null
- Размытие системы типов
- Бесполезные null-чеки
- Java Optional





Option<T>

- Аналог Nullable
- Хорошая альтернатива использованию null-значений
- Может содержать значение типа T...
- или не содержать ничего

Option можно представить в виде Discriminated union



```
// F# Discriminated Union
type Option<'T> =
| Some of 'T
| None
```




Option - пример создания экземпляров

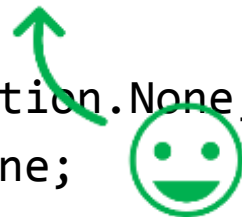
```
using static Option;
```

```
public Option<string> CreateSome() => Option.Some("abc");
```

```
public Option<string> CreateSome() => Some("abc");
```

```
public Option<string> CreateNone() => Option.None;
```

```
public Option<string> CreateNone() => None;
```



Лаконичный,
похожий на
встроенный в язык
вариант



Option - пример создания экземпляров

```
public Option<string> FromUntrustedRef(string s)  
    => s.OptionFromNullable();
```

Option – извлечение значения



Плохой вариант

```
Option<string> option = Some("C#");  
if (option.HasValue)  
{  
    return option.Value;  
}  
return "No value";  
// "C#"
```

Безопасный вариант

```
Option<string> option = Some("C#");  
return  
option.Match(some: str => str,  
             none: () => "No value");  
// "C#"
```

typeof(string)

Можно вызвать это свойство без предварительной проверки

typeof(string)



Option - безопасный доступ к Dictionary

```
Option<TV> TryGetValue<TK, TV>(this Dictionary<TK, TV> dictionary, TK key)
{
    TV value;
    if (dictionary.TryGetValue(key, out value)) return Some(value);
    return None;
}
```

...

```
var dictionary = new Dictionary<string, int> { { "C#", 42 } };
Option<int> option = dictionary.TryGetValue("C#");
var str = option.Match(some: i => "Got value:" + i,
                      none: () => "No value");
Console.WriteLine(str); // "Got value 42"
```

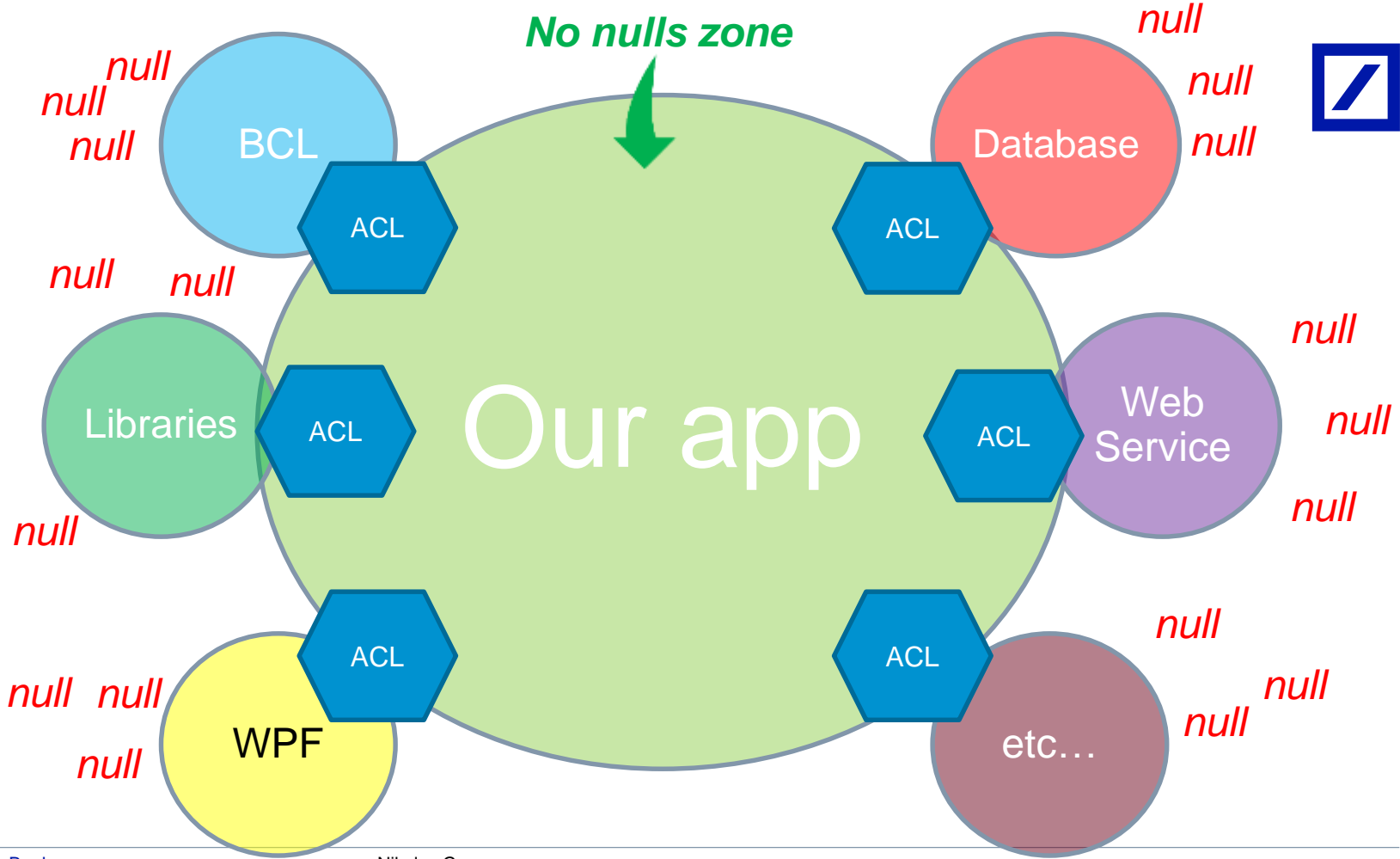


Все еще проверяете аргументы на null в 2017?

```
public void TransferMoney(Bank bank,  
                           Offshore offshore,  
                           Client client,  
                           ...)  
{  
    if (bank == null) throw new ArgumentNullException();  
    if (offshore == null) throw new ArgumentNullException();  
    if (client == null) throw new ArgumentNullException();  
  
    ...  
}
```



Бесполезный код





Option - итоги

- Успешно заменяет собой null-значения
- Compile-time проверка
- Оборачиваем внешние nullable значения
- Resharper “Implicit Nullability” plugin



3. Исключения



Исключения - особенности

- «Исключения»
- TryParse, TryX...
- Удар по производительности
- Скрытый goto
- Приводят к необдуманному коду
- Неявность контракта





Result<T, TError>

- Содержит либо объект типа T...
- либо ошибку типа TError



Result можно представить в виде Discriminated union

```
type Result<'T, 'TError> =  
| Success of 'T  
| Failure of 'TError
```



Result<T, TError> - безопасный доступ к Dictionary

```
Result<TV, string> TryGetValue<TK, TV>(this Dictionary<TK, TV> dictionary, TK key)
{
    TV value;
    if (dictionary.TryGetValue(key, out value)) return Success(value);
    return Failure($"No element with key=${key} found!");
}
```

...

```
Dictionary<string, int> dictionary = new Dictionary<string, int> { { "C#", 42 } };
Result<int, string> result = dictionary.TryGetValue("C#");
var str = result.Match(success: i => "Got value:" + i,
                       failure: err => err);
Console.WriteLine(str); // Got value 42
```

typeof(int)
typeof(string) – тип TError



Result или Option?

Одна причина - Option

- *Dictionary.TryGetValue(TKey key)*
- *Enumerable.SingleOrDefault()*

Несколько причин – используем Result

- *File.Open*
- *SqlConnection.Open*



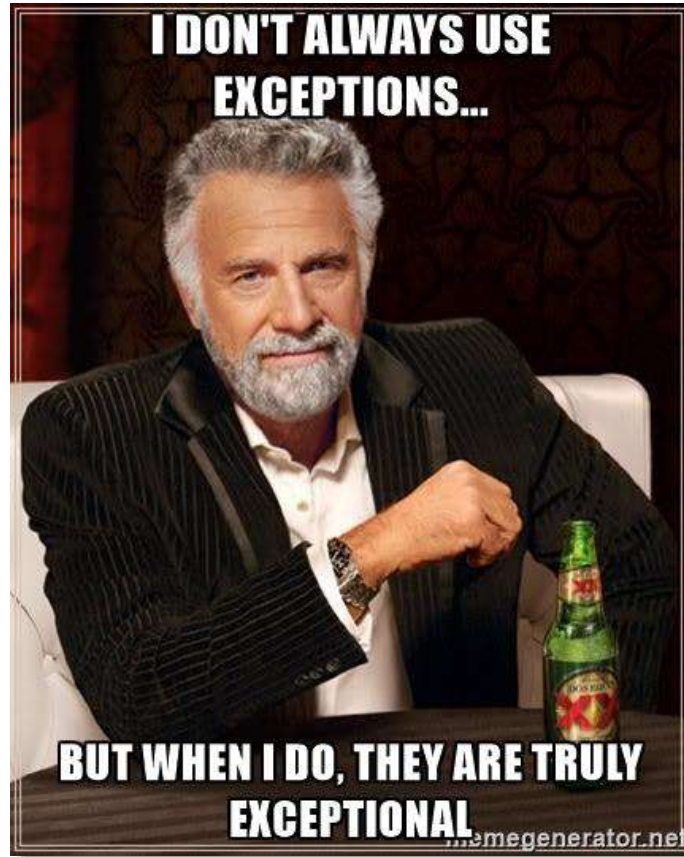
Result – передача ошибок по цепочке

```
// IdError = ParseError | CannotBeEmpty
public Result<CustomerId, IdError> TryGetCustomerId(string id){...}

// GetCustomerError = IdError | DbReadError | TimeoutError
public Result<Customer, CustomerError> TryGetCustomer(...)
{ ... GetCustomerId(...); ... }

// CustomerRenderError = CustomerError | ...
public Result<HttpResponse, CustomerRenderError> TryRender (...)
{ ... TryGetCustomer(...); ... }
```







Итоги - преимущества Result<T, TError>

- Явно прописанный контракт
- Нет побочного эффекта - исключение
- Нет эффекта goto
- Компилятор заставляет проверить результат на ошибку
- Resharper “Exceptional” plugin



Часть II

Расширяем арсенал ФП приемов



II. Расширяем арсенал ФП приемов

- Totality
- Select, SelectMany и Linq



4. Totality

Purity



- Функция возвращает одно и то же значение для одних и тех же входных аргументов.
- Функция использует только свои аргументы для вычисления результата. Функция не может читать внешнее состояние.
- Функция не влияет на внешнее состояние.





Totality

- Функция возвращает валидные значения для всего диапазона ВОЗМОЖНЫХ ВХОДНЫХ значений





Totality - пример

```
public static double Divide(double dividend,  
                             double divisor)  
{  
    if (divisor == 0.0) ???  
    return dividend / divisor;  
}
```

← Что делать
с нулевым делителем?



Totality – пример, решение №1

```
public static double Divide(double dividend,  
                             double divisor)  
{  
    if (divisor == 0.0) throw new Exception();  
    return dividend / divisor;  
}
```



Exception



```
double Divide(double dividend, double divisor)
```





Totality – пример, решение №2

```
public static Option<double> Divide(double dividend,  
                                     double divisor)  
{  
    if (divisor == 0.0) return None;  
    return Some(dividend / divisor);  
}
```



Totality – пример, решение №3

```
public static T Divide<T>(double dividend,
                          double divisor,
                          Func<double, T> onSuccess,
                          Func<T> onFail)
{
    if (divisor == 0.0) return onFail();
    return onSuccess(dividend / divisor);
}
```



Totality – пример, решение №4

```
public static double Divide(double dividend,  
                             NonZeroDouble divisor)  
{  
    return dividend / divisor.Value;  
}
```



Totality – примеры нарушений принципа

```
class Dictionary<T, TKey> {  
    T Get(TKey key) {...}  
}  
  
struct DateTime {  
    DateTime Parse(string str) {...}  
}  
  
class Repository {  
    Client GetClient(string id) {...}  
}
```



Totality – приемы

- Расширяем множество выходных значений
- Сужаем множество входных значений
- Даем клиенту решать, что делать
- Dependently typed languages (Idris, Agda, Coq)



Totality – преимущества применения

- Компилятор проверяет корректность кода
- Честные «функции»
- Меньше багов



5. Select, SelectMany и Linq

Сложность материала





Что хочется получить

```
Option<int> value = Some(1).Select(x => x * 2); // Some(2)
```

```
Option<int> sum =  
    from a in Some(1)  
    from b in Some(2)  
    from c in Some(3)  
    select a + b + c; // Some(6)
```



Select - IEnumerable

```
IEnumerable<TR> Select<T, TR>(this IEnumerable<T> e,  
                             Func<T, TR> f)
```



Select - IEnumerable

```
IEnumerable<int> a = new List<int> { 1, 2, 3 };  
IEnumerable<int> result1 = a.Select(x => x * 2); // { 2, 4, 6}  
  
IEnumerable<int> b = new List<int>();  
IEnumerable<int> result2 = b.Select(x => x * 2); // { }
```

Diagram annotations: A green arrow points from the text `typeof(int)` to the `int` in the first line's generic type parameter. Another green arrow points from the text `typeof(int)` to the `int` in the third line's generic type parameter.



Select - Option

```
Option<int> a = Some(1);
```

```
Option<int> result1 = a.Select(x => x * 2); // Some(2)
```



Распакованное значение - typeof(int)

```
Option<int> b = None;
```

```
Option<int> result2 = b.Select(x => x * 2); // None;
```

Простая аналогия:

Option – IEnumerable без элементов или с одним элементом



Select - Result

```
public IEnumerable<Order> GetCustomerOrders(CustomerId id) => ...
```

```
...
```

```
Result<CustomerId, string> id1 = Success(new CustomerId(123));
```

```
Result<IEnumerable<Order>, string> result1 = id1.Select(id => GetCustomerOrders(id));
```

```
// result1: Success(..orders..)
```

GetCustomerOrders(...) return type

typeof(CustomerId)

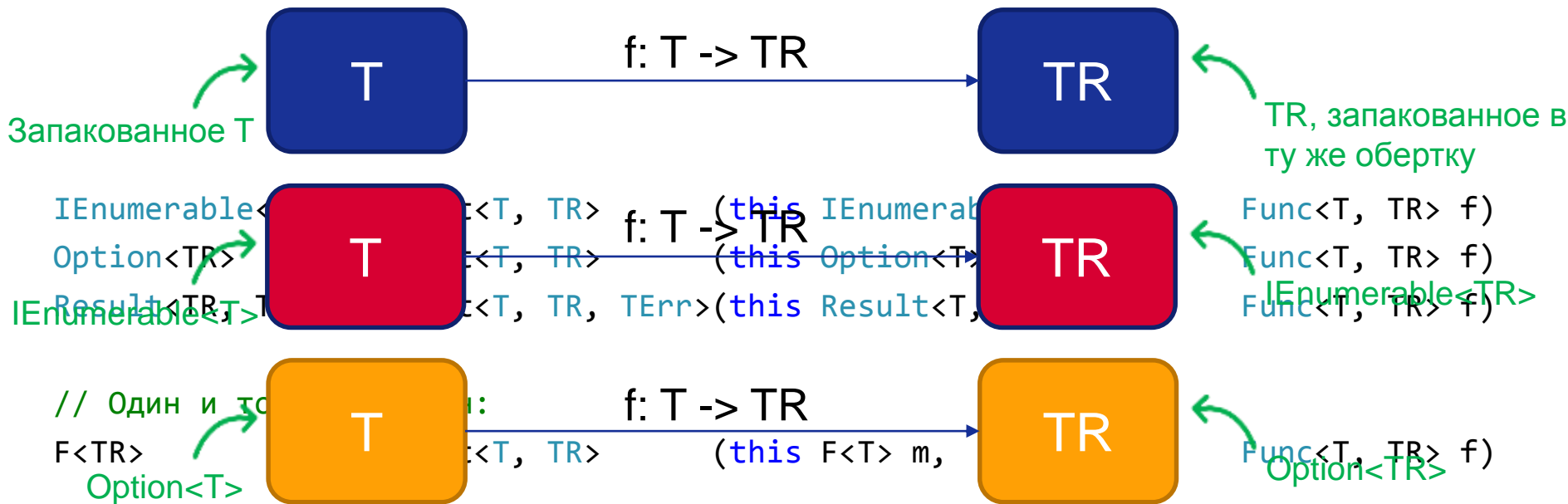
```
Result<CustomerId, string> id2 = Failure("Terrible things happened");
```

```
Result<IEnumerable<Order>, string> result2 = id2.Select(id => GetCustomerOrders(id));
```

```
// result2: Failure("Terrible things happened");
```



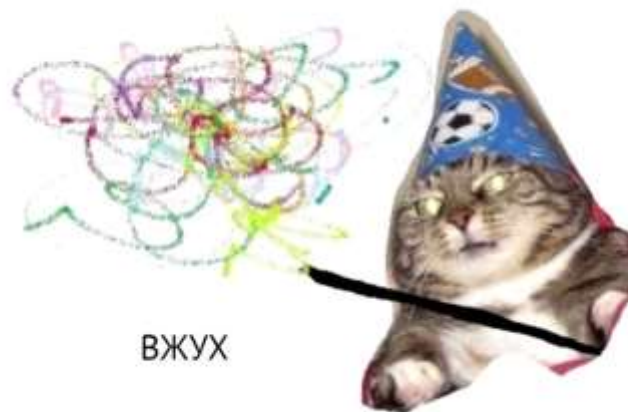
Select (aka 'map')





Select – Чего мы добились?

- Преобразование упакованных значений
- Возможность применения любых стандартных функций





SelectMany - IEnumerable

```
IEnumerable<TR> SelectMany<T, TR>(this IEnumerable<T> e,  
                                  Func<T, IEnumerable<TR>> f)
```



Каждый элемент T преобразуется в
IEnumerable<TR>. Все IEnumerable<TR>
конкатенируются.



// Сравним с Select:

```
IEnumerable<TR> Select<T, TR>(this IEnumerable<T> e,  
                             Func<T, TR> f)
```



Каждый элемент T преобразуется в TR





SelectMany - IEnumerable

```
IEnumerable<int> e = new List<int> { 1, 2, 3 };  
IEnumerable<int> result = e.SelectMany(x => new[] {x, x * 2});  
// result - { 1, 2, 2, 4, 3, 6 }
```



SelectMany – Linq версия IEnumerable

```
IEnumerable<int> list = new List<int> { 1, 2, 3 };  
IEnumerable<int> result = from x in list  
                          from y in new[] { x, x * 2 }  
                          select y;  
  
// { 1, 2, 2, 4, 3, 6 }
```

typeof(int)

typeof(int)

Все 'y' конкатенируются
в результирующий
IEnumerable<int>



SelectMany – IEnumerable имплементация

```
IEnumerable<TR> SelectMany<T, TR>(IEnumerable<T> enumerable,  
                                   Func<T, IEnumerable<TR>> selector)
```

```
{  
    foreach (T e in enumerable)  
    {  
        foreach (TR result in selector(e))  
            yield return result;  
    }  
}
```

Select aka 'map'
SelectMany aka 'flatMap'

Убираем вложенность
конкатенацией



SelectMany – Option имплементация

```
Option<TR> SelectMany<T, TR>(Option<T> option,  
                             Func<T, Option<TR>> selector)  
=> option.Match(some: value => selector(value),  
               none: () => None);
```

SelectMany – Option Linq



`typeof(int)`
`Option<int> sum =`

`from a in Some(1)`

`from b in Some(2)`

`from c in Some(3)`

`select a + b + c; // Some(6)`

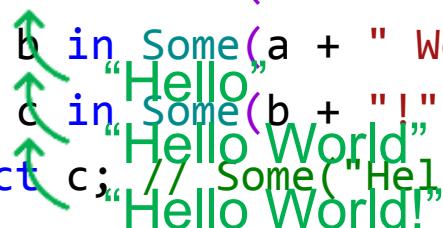
Получили сумму, завернутую в Option,
не распаковывая значений вручную

Применяем функцию, работающую
с int значениями, а не с Option<int>



SelectMany – передача по цепочке

```
Option<string> greeting =  
    from a in Some("Hello")  
    from b in Some(a + " World")  
    from c in Some(b + "!")  
    select c; // Some("Hello World!")
```





SelectMany – Option, ранний выход

Эффект проверки каждого значения на None

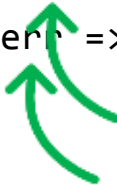
```
Option<int> sum =
```

```
    from a in Some(1) ← None? Нет, продолжаем  
    from b in None     ← None! Возвращаем None  
    from c in Some(3)  для всего выражения  
    select a + b + c; // None ←
```



SelectMany – Result имплементация

```
Result<TR, TError> SelectMany<T, TR, TError>(Result<T, TError> result,  
                                             Func<T, Result<TR, TError>> selector)  
{  
    return result.Match(success: value => selector(value),  
                        failure: err => Failure(err));  
}
```

 `typeof(T)`
`typeof(TError)`



SelectMany - Result

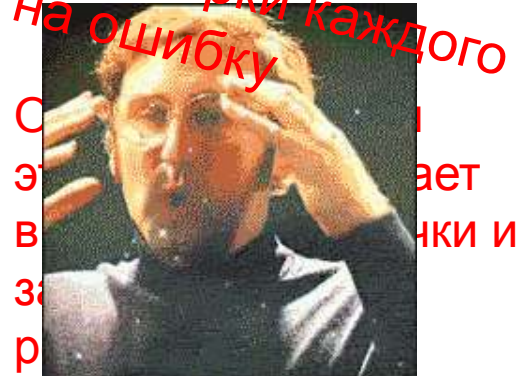
CustomerId customerId = ...

```
var customerItems =  
    from customer in TryGetCustomer(customerId)  
    from orders in TryGetOrders(customer)  
    from items in TryGetItems(orders)  
    select items;
```

typeof(Customer)

```
Result<Customer, string> TryGetCustomer(CustomerId id) => ...  
Result<IEnumerable<Order>, string> TryGetOrders(Customer customer) => ...  
Result<IEnumerable<Item>, string> TryGetItems(IEnumerable<Order> orders) => ...
```

Эффект проверки каждого
вызова на ошибку





SelectMany - Result

На самом деле все довольно просто...

```
var customer = GetCustomer(customerId);  
var orders = GetOrders(customer);  
var items = GetItems(orders);
```

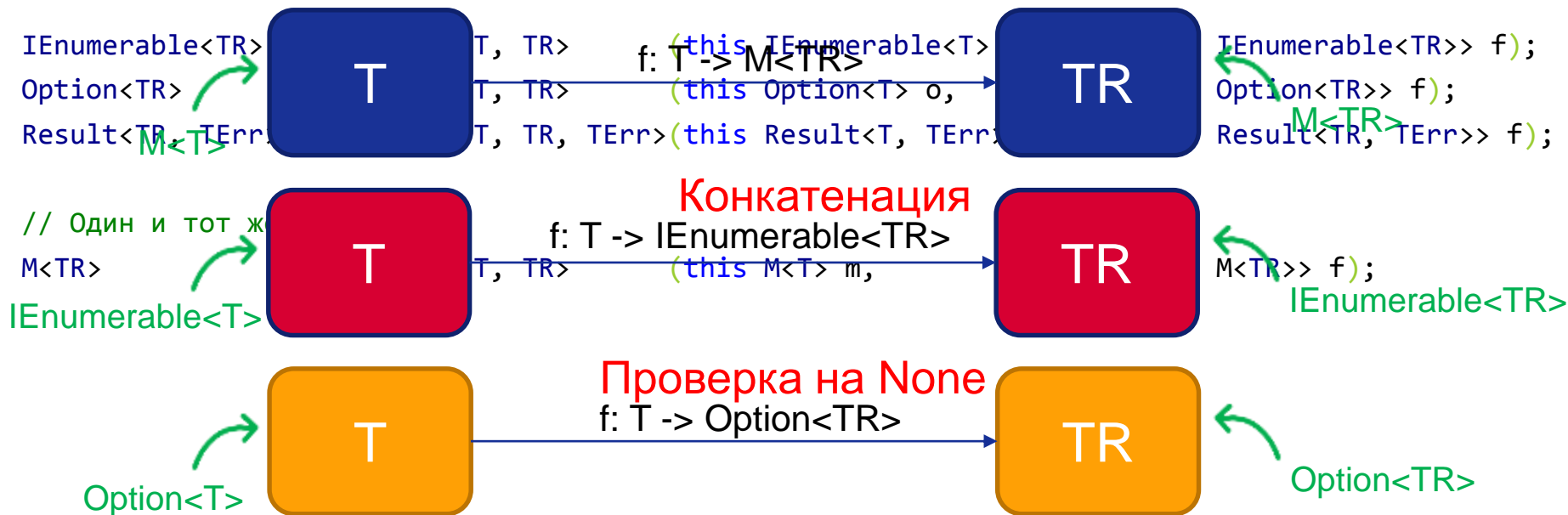
```
from customer in TryGetCustomer(customerId)  
from orders in TryGetOrders(customer)  
from items in TryGetItems(orders)  
select items;
```

Одинаковый код

Императивный код с
дополнительным
эффектом проверки на
ошибки

SelectMany (aka 'flatMap')

За счет удаления
вложенности получаем
доп. эффект





Окей, можно использовать Linq для IEnumerable, Option и Result. А какие еще типы-обертки поддерживает этот синтаксис?





«Тысячи их!»

Библиотека Sprache - парсер



Эффект передачи остатка
строки следующему парсеру

```
Parser<string> idParser =  
    from leading in Parse.WhiteSpace.Many()  
    from first in Parse.Letter.Once()  
    from rest in Parse.LetterOrDigit.Many()  
    from trailing in Parse.WhiteSpace.Many()  
    select new string(first.Concat(rest).ToArray());  
  
var id = idParser.Parse(" abc123 ");  
  
Assert.AreEqual("abc123", id);
```

Библиотека FsCheck - генератор случайных значений



```
Gen<Person> personGenerator =
```

```
  from age in Arb.Default.Int32().Generator
```

```
  from name in Arb.Default.String().Generator
```

```
  select new Person(age, name);
```

Эффект генерации
случайных значений
генераторы
Gen<int> и
Gen<string>

```
Gen<Worker> workerGenerator =
```

```
  from person in personGenerator
```

```
  from salary in Arb.Default.Double().Generator
```

```
  select new Worker(person, salary);
```

```
Worker worker = workerGenerator.Eval(...);
```

Получившийся генератор Gen<Person> можно
использовать для создания новых генераторов!



Select, SelectMany и Linq - итоги

- Select
- SelectMany
- Linq
- Меньше кода – меньше багов.



ИТОГИ





Что мы сегодня рассмотрели

- Discriminated Unions как альтернативу Наследованию
- Option как замену null значений
- Замену Exception'ам – Result, не позволяющий возможным ошибкам пройти незамеченными





Что мы сегодня рассмотрели

- Totality и Purity, делающие наши методы простыми и понятными
- Использование Select и Linq для коробочных типов M<T> (IEnumerable, Option, Result, Parser, Gen)





Что мы сегодня рассмотрели

- Простота и сочетаемость концепций
- Строгая типизация - лучший друг
- C# + ФП
- Плюсы ФП
- ♥



https://github.com/NikolayGusev/CSharp_FP_Presentation

<https://github.com/NikolayGusev/DiscriminatedUnions>

[Lang ext \(C# functional library\)](#)

Что еще послушать?



Basic

Functional programming design patterns by Scott Wlaschin

Functional Principles for Object Oriented Development by Jessica Kerr

Advanced

Railway oriented programming: Error handling in functional languages by Scott Wlaschin

Simple Made Easy by Rich Hickey

Functional Programming from First Principles by Erik Meijer

Expert

Don't fear the Monad by Brian Beckman

Functional Programming Fundamentals by Dr. Erik Meijer

Что еще почитать?



<http://fsharpforfunandprofit.com/>

[Learn You a Haskell](#)

Библиотеки

[Lang ext \(C# functional library\)](#)

[FsCheck \(Property Based Testing\)](#)

[Sprache \(parser\)](#)



ВОПРОСЫ?



Данный материал не является предложением или предоставлением какой-либо услуги. Данный материал предназначен исключительно для информационных и иллюстративных целей и не предназначен для распространения в рекламных целях. Любой анализ третьих сторон не предполагает какого-либо одобрения или рекомендации. Мнения, выраженные в данном материале, являются актуальными на текущий момент, появляются только в этом материале и могут быть изменены без предварительного уведомления. Эта информация предоставляется с пониманием того, что в отношении материала, предоставленного здесь, вы будете принимать самостоятельное решение в отношении любых действий в связи с настоящим материалом, и это решение является основанным на вашем собственном суждении, и что вы способны понять и оценить последствия этих действий. ООО "Технологический Центр Дойче Банка" не несет никакой ответственности за любые убытки любого рода, относящихся к этому материалу.