



F# и функциональное программирование для C# разработчика



F# и функциональное программирование для C# разработчика

Автор вебинара



Альберт Ташу

.NET Developer. Тренер-консультант учебного центра CyberBionic Systematics и сертифицированный специалист Microsoft



F# и функциональное программирование для C# разработчика

План

1. Кому будет полезен этот вебинар
2. Знакомство с языком программирования F#
3. Основы функционального программирования
4. Применение функциональных подходов при разработке на C#
5. Практика
6. Дополнительная информация

F# и функциональное программирование для C# разработчика

Для кого этот вебинар

- Для .NET разработчиков, не знакомых с F# или функциональным программированием.
- Для тех, кто немного знаком с технологией и хочет углубить свои знания.
- Для .NET разработчиков, желающих писать более чистый и простой код на C#.

F# и функциональное программирование для C# разработчика

Основные идеи данного вебинара

- .NET - это не только C#.
- F# - это весело.
- Различные парадигмы программирования часто служат для достижения одинаковых целей.

F# и функциональное программирование для C# разработчика

Что такое язык F#



- Язык для платформы .NET
- Мультипарадигмальный язык
- Интегрирован в Visual Studio

F# и функциональное программирование для C# разработчика

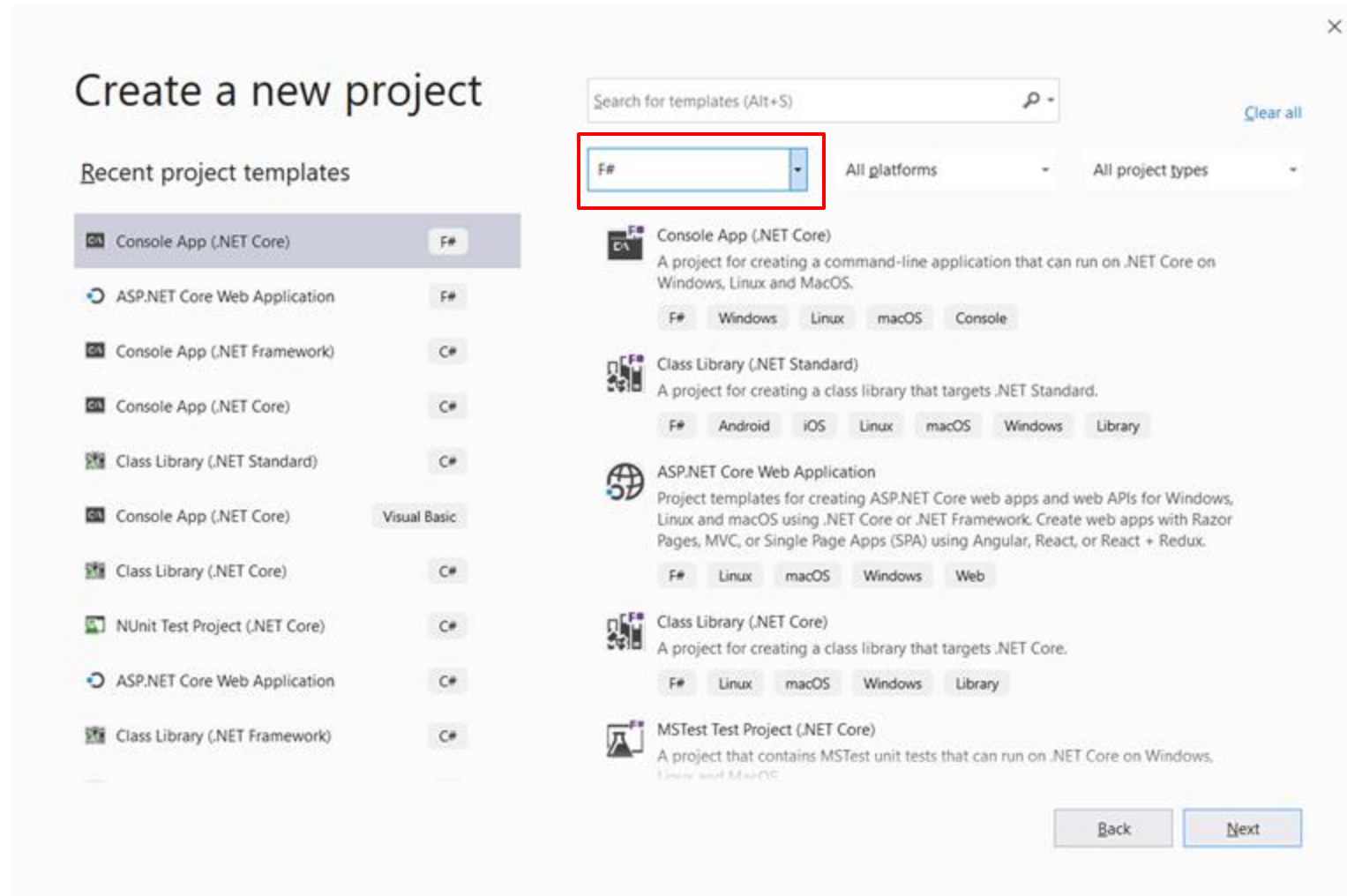
Как запустить код на F#

- Visual Studio (начиная с 2010)
- В интерактивном режиме, используя fsi.exe
- <https://dotnetfiddle.net>



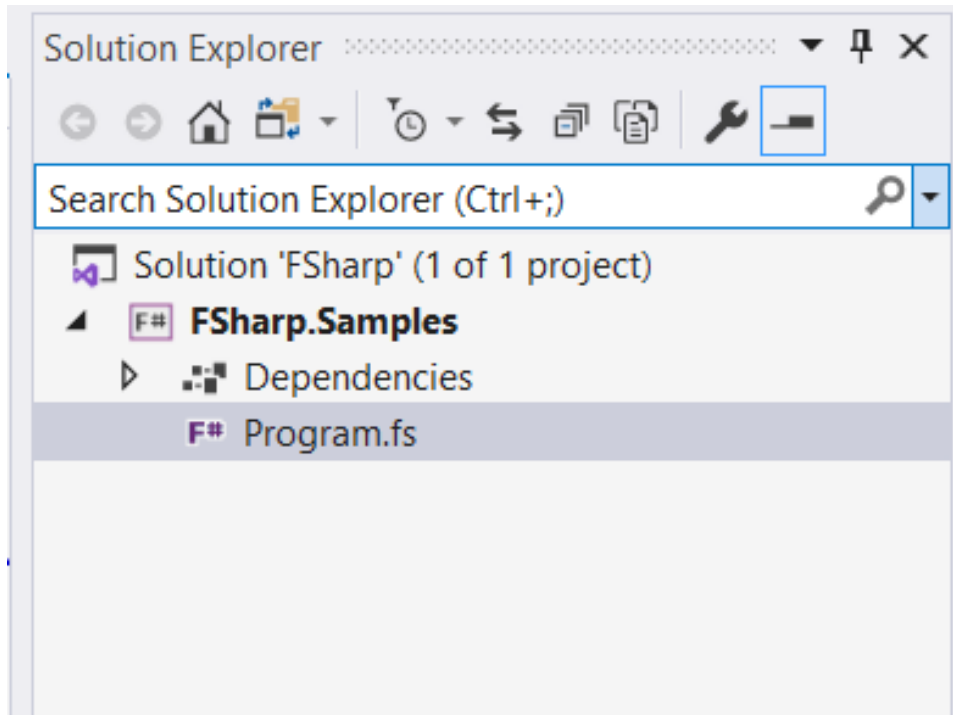
F# и функциональное программирование для C# разработчика

F# в Visual Studio



F# и функциональное программирование для C# разработчика

F# в Visual Studio



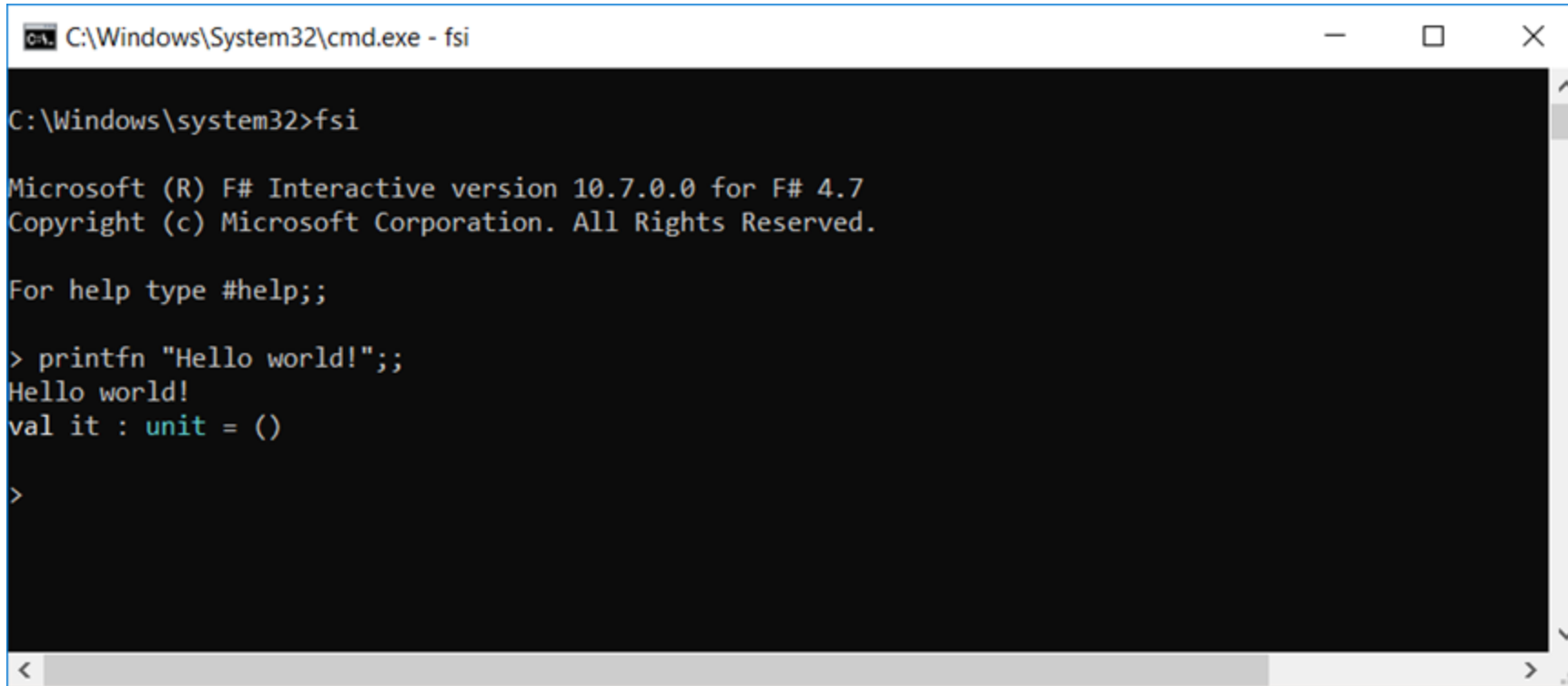
F# использует для организации кода модель, аналогичную модели C# (Project, Solution).

Отличия F# проектов:

- порядок файлов в проекте имеет значение для успешной компиляции проекта;
- порядок файлов можно задавать явно (в отличие от алфавитного порядка).

F# и функциональное программирование для C# разработчика

Интерактивное выполнение F#



```
C:\Windows\System32\cmd.exe - fsi

C:\Windows\system32>fsi

Microsoft (R) F# Interactive version 10.7.0.0 for F# 4.7
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

> printfn "Hello world!";;
Hello world!
val it : unit = ()

>
```

Код на языке F# можно как скомпилировать в исполняемый файл (.exe), так и выполнять в интерактивном режиме, используя интерпретатор fsi.exe.

F# и функциональное программирование для C# разработчика

Интерактивное выполнение F# в Visual Studio

F# Interactive



```
> let greet obj = printfn "Hello %s!" obj;;  
val greet : obj:string -> unit  
  
> greet "World";;  
Hello World!  
val it : unit = ()  
  
>
```

Выполнение F# в интерактивном режиме поддерживается в Visual Studio F# Interactive.

F# Interactive находится во вкладке View > Other Windows > F# Interactive

F# и функциональное программирование для C# разработчика

Интерактивное выполнение F# в Visual Studio

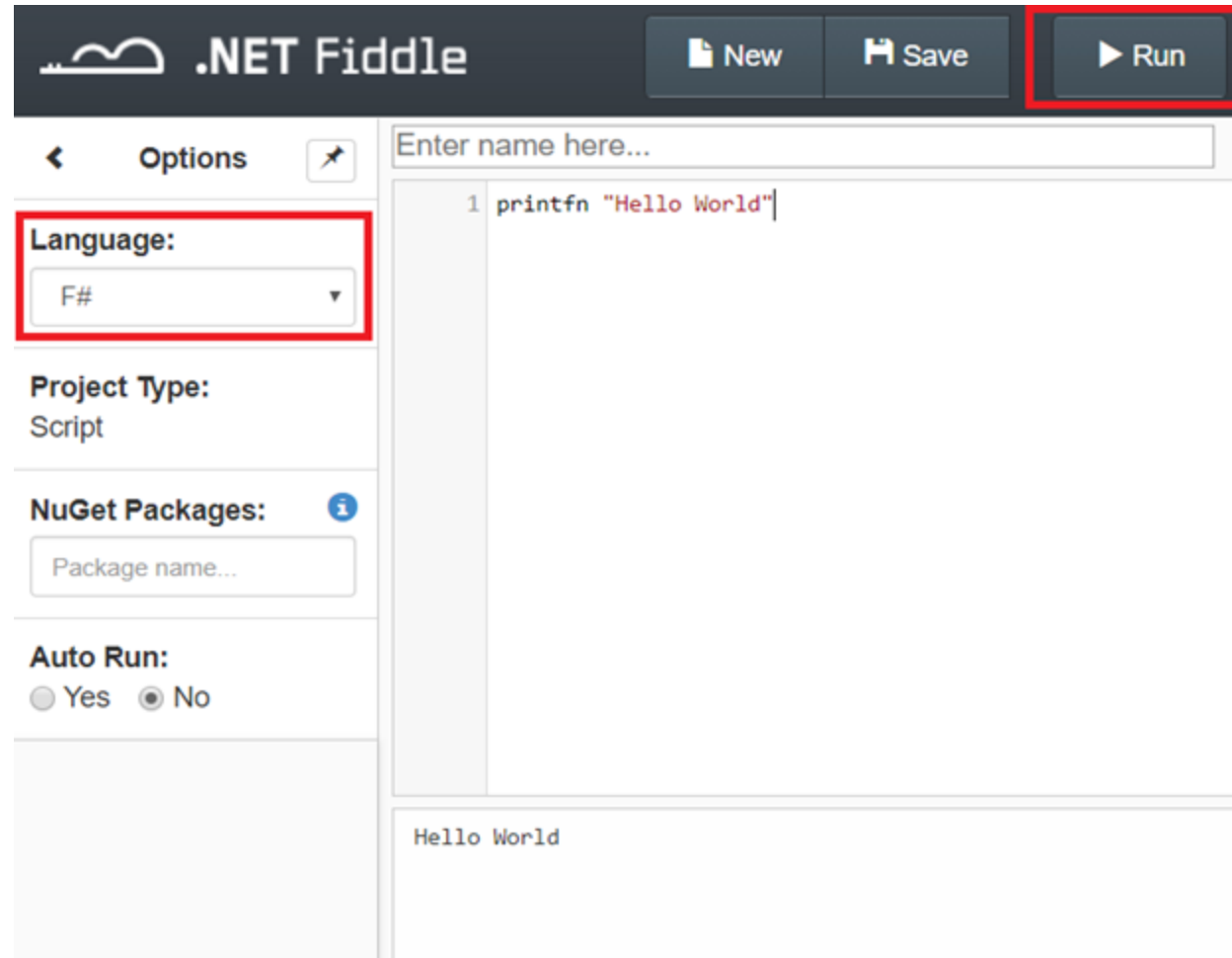
	Source File	Visual F# Items
	Script File	Visual F# Items

Для написания кода на F# и последующего выполнения этого кода в интерактивном режиме используются F# скрипты - файлы с расширением .fsx.

.fsx файлы могут быть добавлены в F# Project в Visual Studio

F# и функциональное программирование для C# разработчика

Выполнение F# в dotnetfiddle.net



F# и функциональное программирование для C# разработчика

Некоторые отличия F# от C#

- Синтаксис
- Привязка неизменяемых значений вместо переменных
- Функции - объекты первого класса
- Выражения вместо инструкций
- Более мощный механизм вывода типов
- Алгебраическая система типов

F# и функциональное программирование для C# разработчика

Синтаксис F#



```
let rec fib n a b =  
  match n with  
  | 0 -> a  
  | 1 -> b  
  | _ -> fib (n - 1) b (a + b)
```

F# относится к семейству языков ML.

Вследствие этого его синтаксис существенно отличается от семейства языков C.

F# и функциональное программирование для C# разработчика

Привязка значений

```
// привязка (binding) значений
// let binding ассоциирует значение с его именем
// аналогично созданию переменной в C#, но значение переменной неизменяемо

// ассоциируем значение 5 и имя intValue
let intValue = 5

//это не присвоение, это сравнение (аналог в C#: intValue == 10)
intValue = 10

//присвоение, возникает ошибка компиляции, потому что intValue неизменяемое значение
intValue <- intValue + 1

let newValue = intValue + 1
```


F# и функциональное программирование для C# разработчика

Неизменяемые значения вместо переменных

C#

```
var value = 10;  
value = value + 1;
```

F#

```
let value = 10  
let newValue = value + 1
```

```
let mutable mutableValue = 10  
mutableValue <- mutableValue + 1
```

F# и функциональное программирование для C# разработчика

Списки, массивы и последовательности

C#

```
var list = new List<int> {1, 2, 3, 4, 5};  
var listFromOneToTen = Enumerable.Range(1,10).ToList();  
  
IEnumerable<int> sequence = new List<int> {1, 2, 3, 4, 5};  
var sequenceFromOneToTen = Enumerable.Range(1, 10);  
  
var array = new int[] {1, 2, 3, 4, 5};  
var arrayFromOneToTen = Enumerable.Range(1, 10).ToArray();
```

F#

```
let list = [1;2;3;4;5]  
let listFromOneToTen = [1..10]  
  
let sequence = seq {1;2;3;4;5}  
let sequenceFromOneToTen = {1..10}  
  
let array = [|1;2;3;4;5|]  
let arrayFromOneToTen = [|1..10|]  
  
let concatenation = listFromOneToTen @ list
```

F# и функциональное программирование для C# разработчика

Функции в F# - объекты первого класса

Объектами первого класса в контексте конкретного языка программирования называются элементы, которые могут:

- быть переданы как параметр;
- быть возвращены из функции;
- быть присвоены переменной.

[Wikipedia]

F# и функциональное программирование для C# разработчика


Функции в F#

```
- let f x =  
  [ sin x ** 2. + 1.
```

Функции - определяются так же, как привязываются обычные значения.

F# и функциональное программирование для C# разработчика

Функции в F#

```
 let applyToList list f =  
    List.map f list
```

Функции можно передавать в качестве аргумента в другие функции.

F# и функциональное программирование для C# разработчика

Функции в F#

```
let combine f1 f2 =  
    f1 >> f2
```

Функции могут быть возвращаемым значением других функций.

F# и функциональное программирование для C# разработчика

Функции и делегаты

C#

```
Func<double, double> f = x =>  
    Math.Pow(Math.Sin(x), 2) + 1;
```

F#

```
let f x =  
    sin x ** 2. + 1.
```

Выражения вместо инструкций

Выражение (expression) - блок программного кода, который вычисляет некоторое значение и возвращает его.

Инструкция (statement) - указание выполнения некоторого действия.

F# и функциональное программирование для C# разработчика

Выражения вместо инструкций

Выражение

```
0 references  
public int Max(int a, int b)  
{  
    return a > b ? a : b;  
}
```

Инструкция

```
0 references  
public int Max(int a, int b)  
{  
    if (a > b)  
    {  
        return a;  
    }  
    else  
    {  
        return b;  
    }  
}
```

F# и функциональное программирование для C# разработчика

Выражения вместо инструкций

```
[-] let ifExpression value =  
    |  
    let result = if value > 0 then 1 else -1  
    result
```

```
[-] let matchExpression value =  
    [-] match value with  
    | 1 -> "one"  
    | 2 -> "two"  
    | x when x < 0 -> "neagtive"  
    | _ -> "other"
```

F# и функциональное программирование для C# разработчика

Выведение типов

0 references

```
public IEnumerable<TSource> Where<TSource>(
    IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    return source.Where(predicate);
}
```

F# и функциональное программирование для C# разработчика

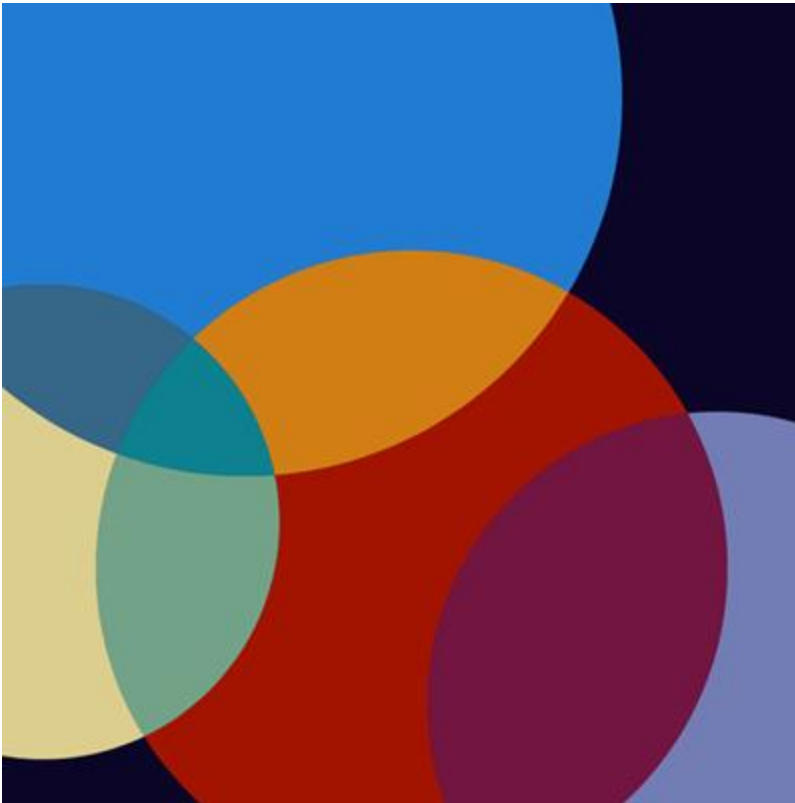
Выведение типов

```
[-] let where source predicate =  
    [ : List.filter predicate source
```

```
>  
val where : source:'a list -> predicate:('a -> bool) -> 'a list
```

F# и функциональное программирование для C# разработчика

Система типов F#

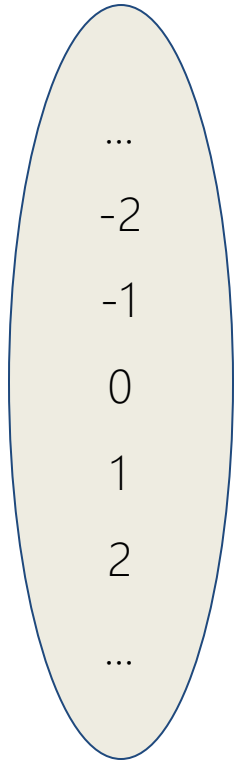


F# использует алгебраическую систему типов.

С помощью типов-сумм и типов-произведений можно очень гибко моделировать предметную область приложения.

F# и функциональное программирование для C# разработчика

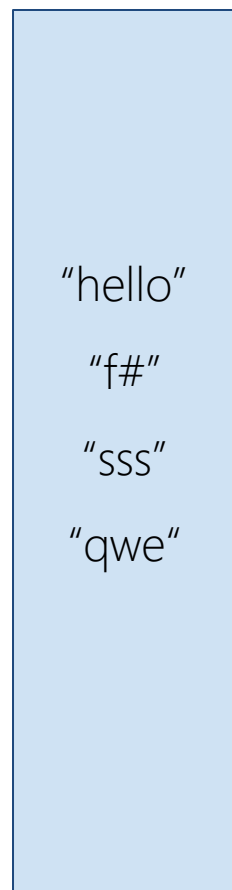
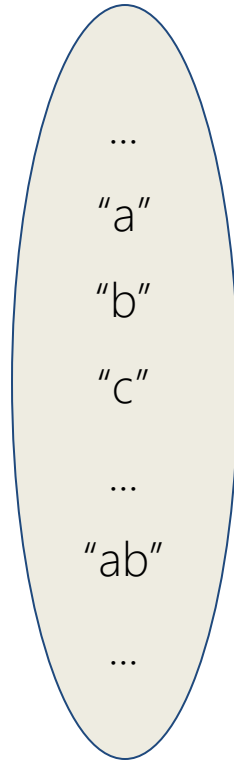
Тип-произведение (AND type)



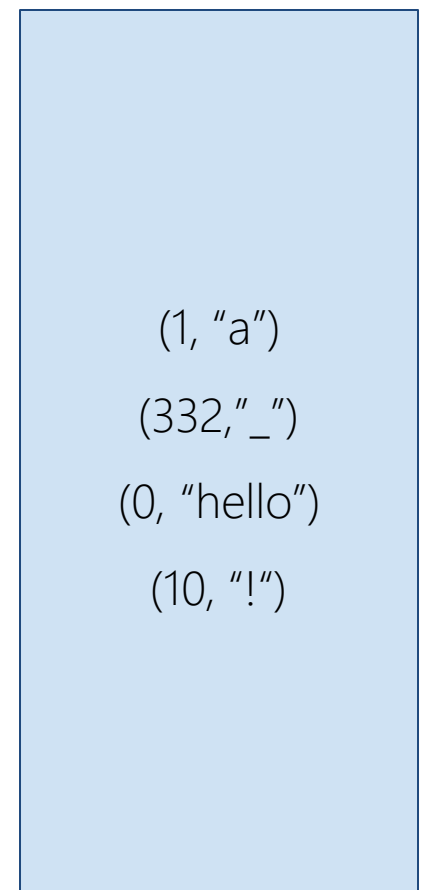
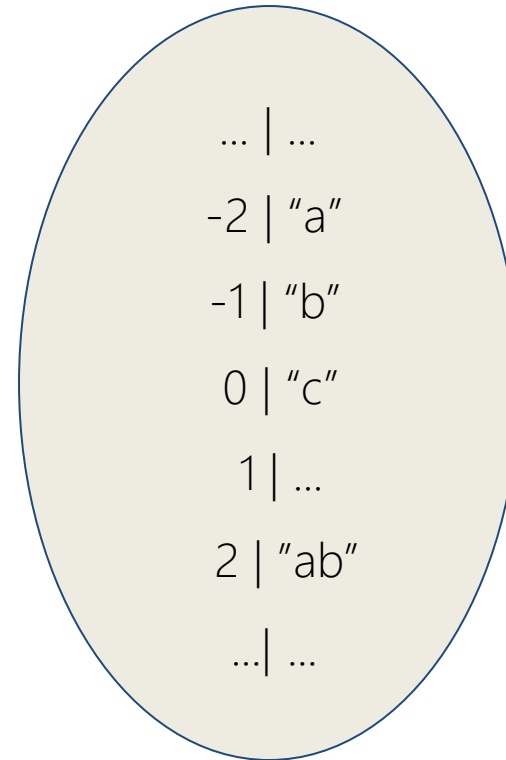
int



string



int * string



F# и функциональное программирование для C# разработчика

Tuple

Кортеж - сгруппированный набор не именованных значений.

```
let tuple = ("string", 10, 1.)
```

Значения могут быть разных типов.

```
>  
val tuple : string * int * float = ("string", 10, 1.0)
```

Кортежи неизменяемы.

```
let (first, second, third) = tuple
```

F# и функциональное программирование для C# разработчика

Record

Запись - набор именованных значений.

Тоже неизменяемы.

```
type Customer = {  
    Name : string;  
    OrdersTotal : int;  
}
```

```
let mike = {  
    Name = "Mike";  
    OrdersTotal = 0;  
}
```

```
let mikeAfter1stOrder =  
    {mike with OrdersTotal = 1}
```


F# и функциональное программирование для C# разработчика

Record

Имеют встроенные механизмы для проверки на структурное равенство.

```
mikeAfter1stOrder = mike
```

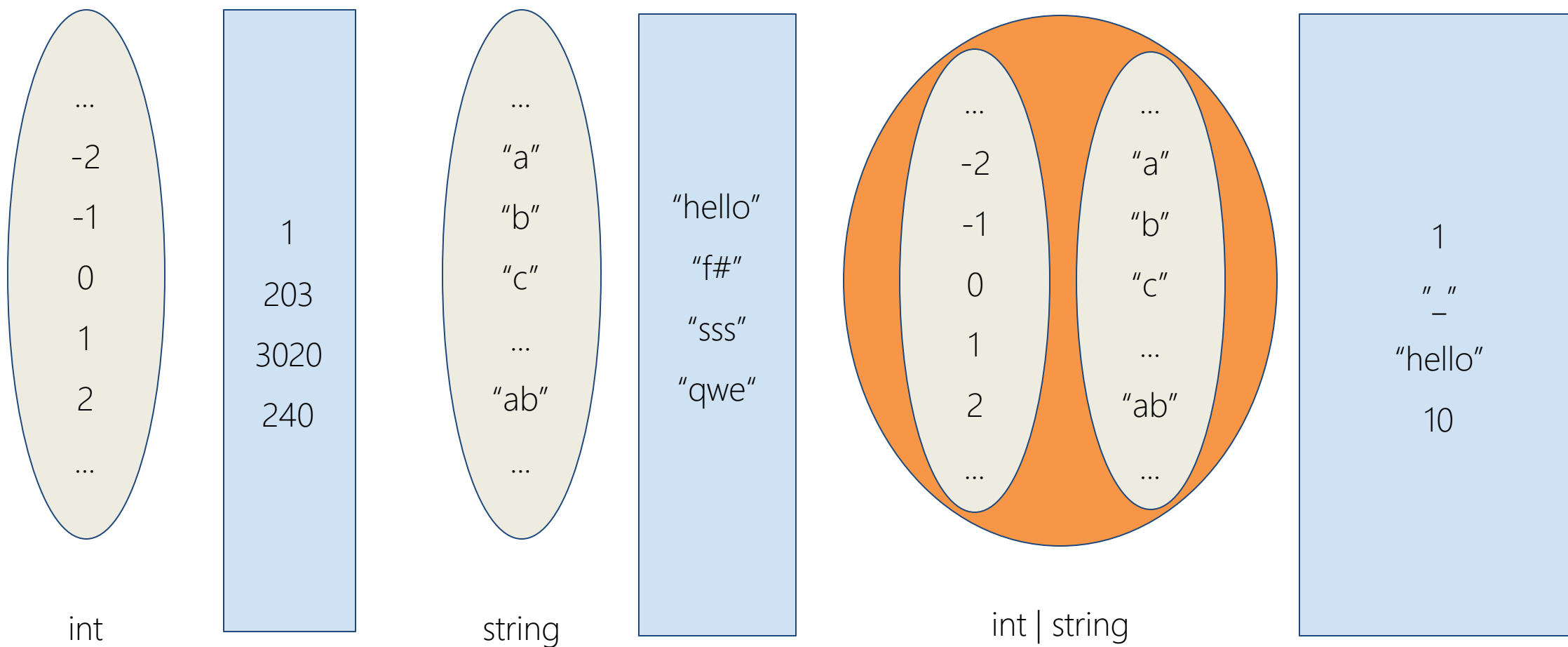
```
>  
val it : bool = false
```

```
= mike = {  
    Name = "Mike";  
    OrdersTotal = 0;  
}
```

```
>  
val it : bool = true
```

F# и функциональное программирование для C# разработчика

Тип-сумма (OR type)



F# и функциональное программирование для C# разработчика

Union

```
type Temperature = F of float | K of float | C of float
```

```
let getTemperatureString temperature =  
    match temperature with  
    | F x -> sprintf "%f °F" x  
    | C x -> sprintf "%f °C" x  
    | K x -> sprintf "%f °K" x
```

```
let temperature = F 10.99
```

```
getTemperatureString temperature
```

F# и функциональное программирование для C# разработчика

Union

```
type BankAccount = string
```

```
type Currency = USD | UAH | RUB
```

```
type Money = {  
    Currency : Currency;  
    Dollars : int;  
    Cents : int;  
}
```

```
type TransferRecord = {Amount : Money; To : BankAccount}
```

```
type Transaction = Income of Money | Expense of Money | Transfer of TransferRecord
```

F# и функциональное программирование для C# разработчика

Union

```
let processTransaction transaction =  
    match transaction with  
    | Income x ->  
        let currency = x.Currency.ToString()  
        printfn "Income transaction for %d.%d %s" x.Dollars x.Cents currency  
    | Expense x ->  
        let currency = x.Currency.ToString()  
        printfn "Expense transaction for %d.%d %s" x.Dollars x.Cents currency  
    | Transfer x ->  
        let amount = x.Amount  
        let currency = amount.Currency.ToString()  
        printfn "Transfer of %d.%d %s to %s" amount.Dollars amount.Cents currency x.To
```

F# и функциональное программирование для C# разработчика

А что с ООП?



F# и функциональное программирование для C# разработчика

Синтаксис F#. Классы

```
2 references
class MyClass
{
    3 references
    public string Name { get; set; }
    3 references
    public int Value { get; set; }

    1 reference
    public MyClass(string name, int value)
    {
        Name = name;
        Value = value;
    }

    1 reference
    public void Print()
    {
        Console.WriteLine($"{Name} : {Value}");
    }

    0 references
    public override string ToString()
    {
        return $"{Name} : {Value}";
    }
}
```

F# и функциональное программирование для C# разработчика

Синтаксис F#. Классы

```
2 references
class MyClass
{
    3 references
    public string Name { get; set; }
    3 references
    public int Value { get; set; }

    1 reference
    public MyClass(string name, int value)
    {
        Name = name;
        Value = value;
    }

    1 reference
    public void Print()
    {
        Console.WriteLine($"{Name} : {Value}");
    }

    0 references
    public override string ToString()
    {
        return $"{Name} : {Value}";
    }
}
```


F# и функциональное программирование для C# разработчика

Синтаксис F#. Классы

2 references

```
class MyClass
```

3 references

```
public string Name { get; set; }
```

3 references

```
public int Value { get; set; }
```

1 reference

```
public MyClass(string name, int value)
```

```
    Name = name;
```

```
    Value = value;
```

1 reference

```
public void Print()
```

```
    Console.WriteLine($"{Name} : {Value}");
```

0 references

```
public override string ToString()
```

```
    return $"{Name} : {Value}";
```

F# и функциональное программирование для C# разработчика

Синтаксис F#. Классы

2 references

`class MyClass`

Так как у классов чаще всего только 1 конструктор, он переносится к определению имени класса.

3 references

`public string Name { get; set; }`

3 references

`public int Value { get; set; }`

1 reference

`public MyClass(string name, int value)`

`Name = name;`

`Value = value;`

1 reference

`public void Print()`

`Console.WriteLine($"{Name} : {Value}");`

0 references

`public override string ToString()`

`return $"{Name} : {Value}";`

F# и функциональное программирование для C# разработчика

Синтаксис F#. Классы

2 references

```
class MyClass(Name: string, Value:int)
```

1 reference

```
public void Print()
```

```
    Console.WriteLine($"{Name} : {Value}");
```

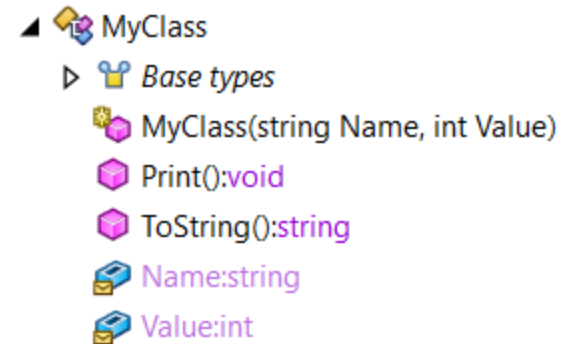
0 references

```
public override string ToString()
```

```
    return $"{Name} : {Value}";
```

Также мы избавились от явного определения свойств.

Аргументы конструктора будут помещены в приватные поля.



F# и функциональное программирование для C# разработчика

Синтаксис F#. Классы

```
2 references
class MyClass(Name: string, Value:int)
1 reference
    public void Print()
        Console.WriteLine($"{Name} : {Value}");
0 references
    public override string ToString()
        return $"{Name} : {Value}";
```

Меняются некоторые ключевые слова:

type - для определения класса

member - для определения члена класса

F# и функциональное программирование для C# разработчика

Синтаксис F#. Классы

2 references

```
type MyClass(Name: string, Value:int) =  
  
    member this.Print() =  
        Console.WriteLine($"{Name} : {Value}");  
  
    override this.ToString() =  
        return $"{Name} : {Value}";
```

F# и функциональное программирование для C# разработчика

Синтаксис F#. Классы

2 references

```
type MyClass(Name: string, Value:int) =  
  
    member this.Print() =  
        Console.WriteLine($"{Name} : {Value}");  
  
    override this.ToString() =  
        return $"{Name} : {Value}";
```

Так как в F# каждая функция должна возвращать значение, от return тоже избавимся.

Также немного отличается API для работы со строками и консолью

F# и функциональное программирование для C# разработчика

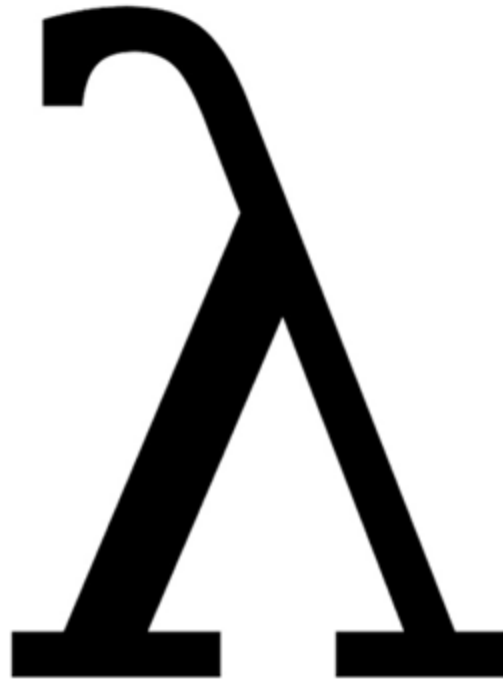
Синтаксис F#. Классы

2 references

```
type MyClass(Name: string, Value:int) =  
  
    member this.Print() =  
        printf "%s : %d" Name Value  
  
    override this.ToString() =  
        sprintf "%s : %d" Name Value
```

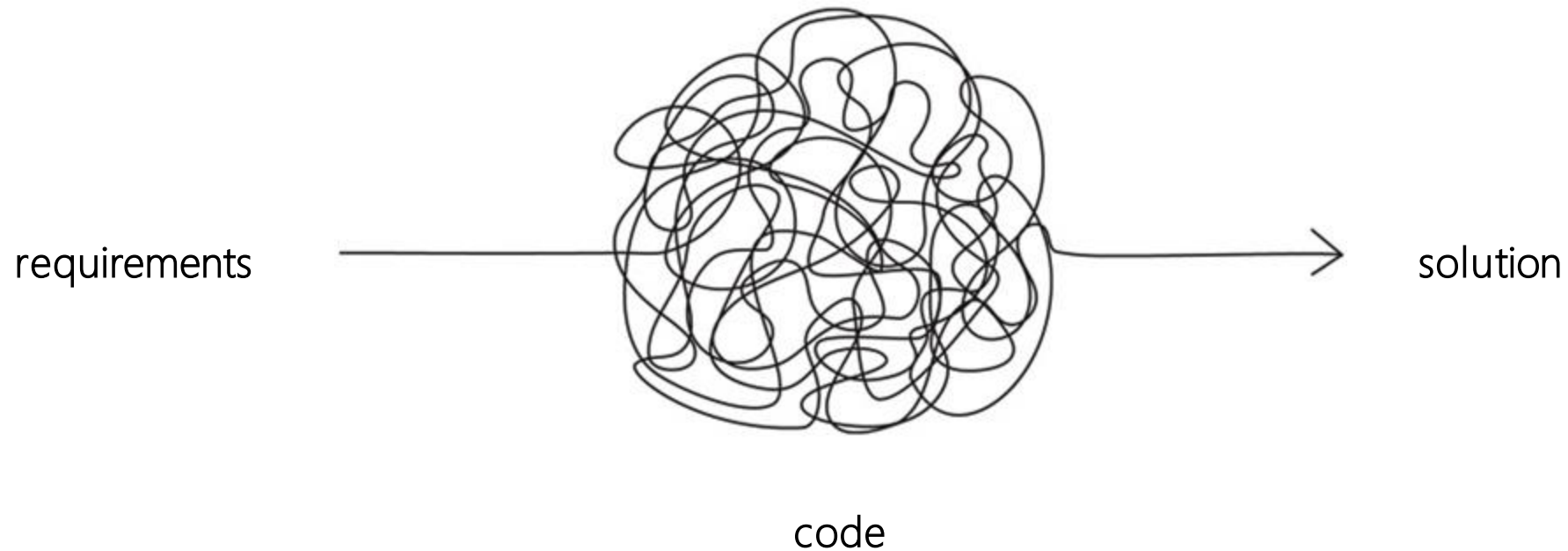
F# и функциональное программирование для C# разработчика

Функциональное программирование



F# и функциональное программирование для C# разработчика

Сложность

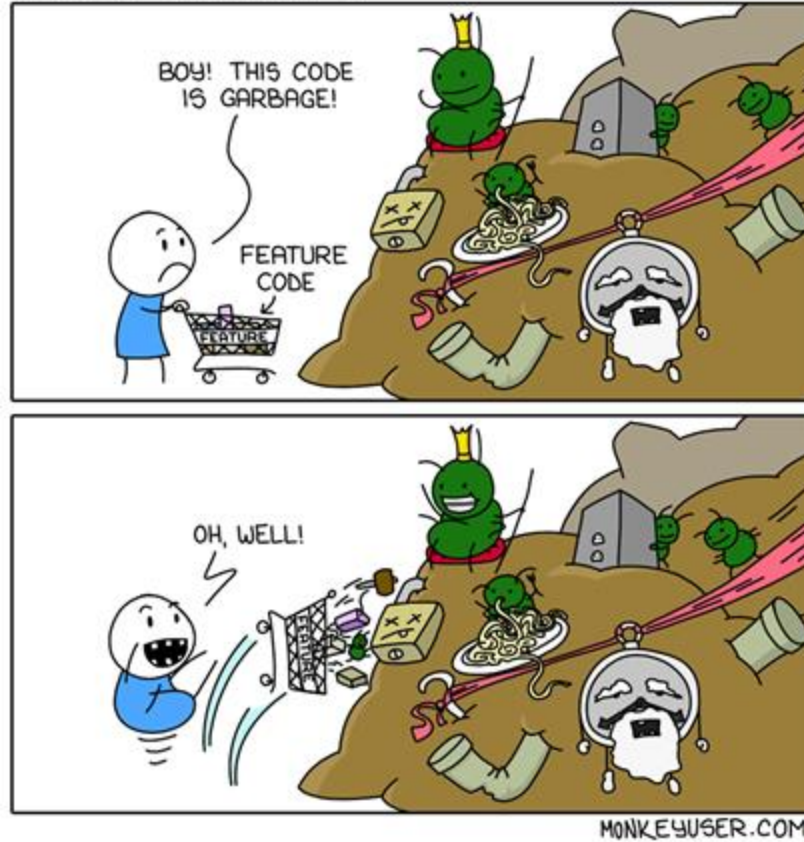


Большое количество кода - это сложно.

F# и функциональное программирование для C# разработчика

Энтропия

CODE ENTROPY



F# и функциональное программирование для C# разработчика

Борьба со сложностью

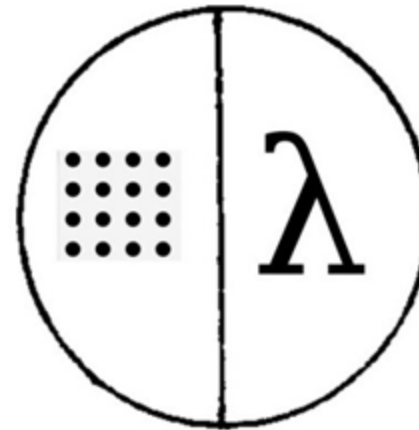
Imperative



Functional



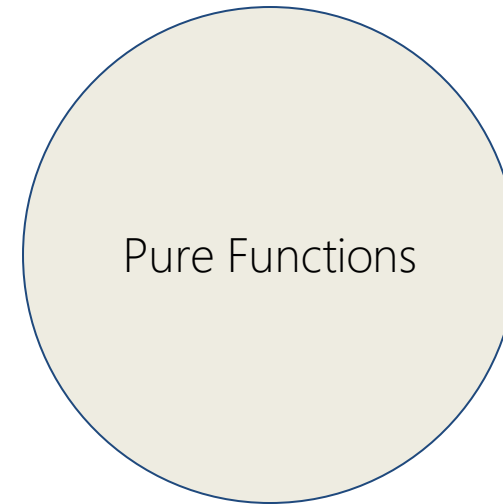
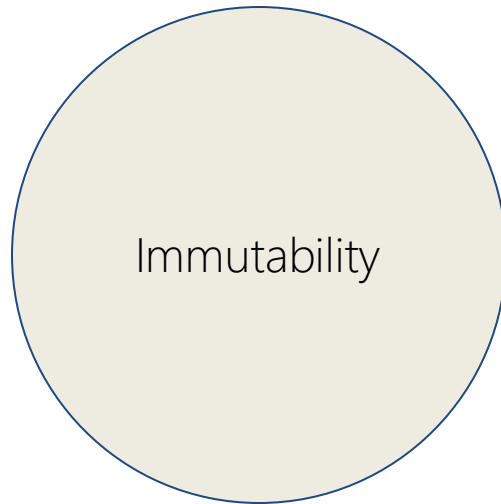
Object-Oriented



Разные парадигмы программирования используют разные подходы к борьбе со сложностью.

F# и функциональное программирование для C# разработчика

Принципы функционального программирования



F# и функциональное программирование для C# разработчика

Неизменяемость

Меньше изменяемых частей в системе - проще управлять её поведением.

F# и функциональное программирование для C# разработчика

Неизменяемость

```
3 references
class Point
{
    1 reference
    public int X { get; set; }
    1 reference
    public int Y { get; set; }

    1 reference
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

```
var point = new Point(1,2);

ProcessPoint(point);

// point.X = ?
// point.Y = ?
```

Какое значение у полей X и Y объекта point после вызова метода ProcessPoint?

F# и функциональное программирование для C# разработчика

Неизменяемость

```
3 references
class Point
{
    1 reference
    public int X { get; }
    1 reference
    public int Y { get; }

    1 reference
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

```
var point = new Point(1,2);

ProcessPoint(point);

// point.X = ?
// point.Y = ?
```

Какое значение у полей X и Y объекта point после вызова метода ProcessPoint?

F# и функциональное программирование для C# разработчика

Неизменяемость



Michael Feathers @mfeathers · 3 нояб. 2010 г.

OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.



8



457

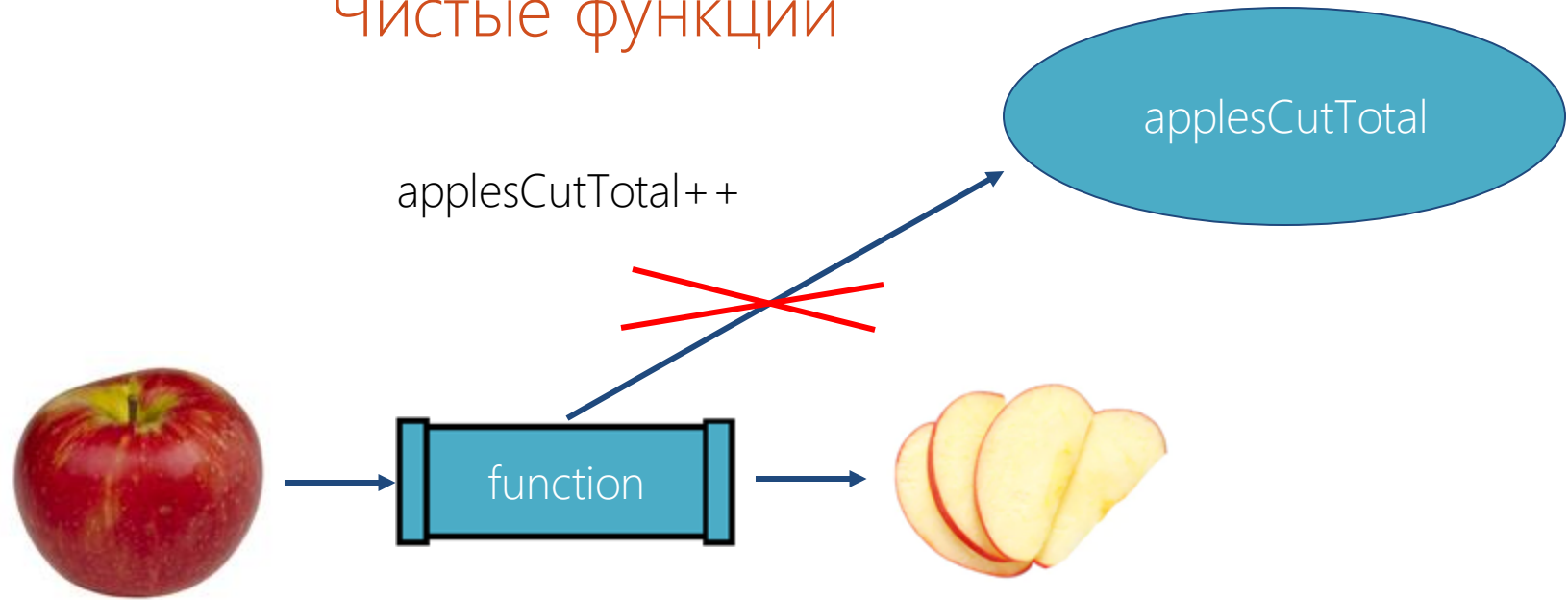


438



F# и функциональное программирование для C# разработчика

Чистые функции

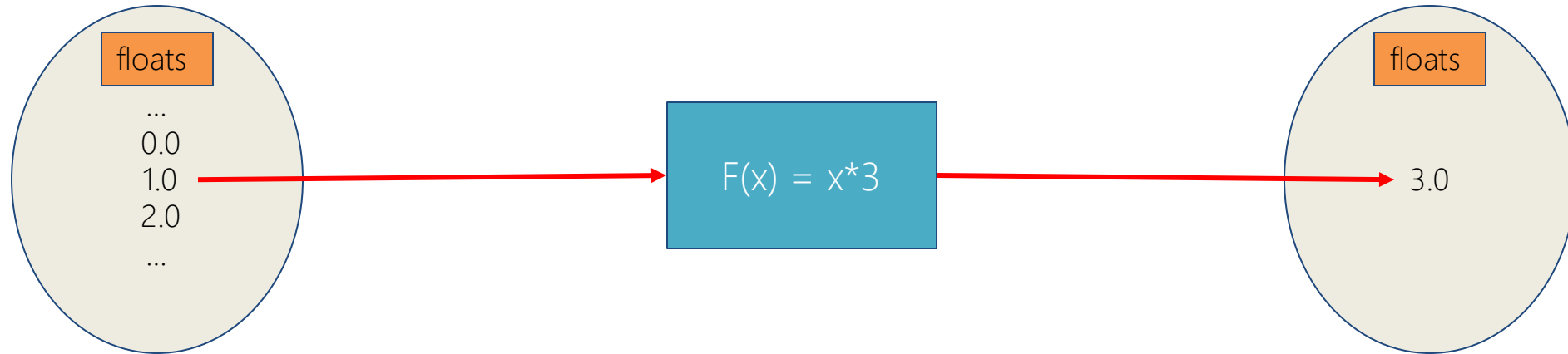


Чистая функция (pure function):

- является детерминированной: при одинаковых входных данных возвращает одинаковые исходящие данные;
- не обладает побочными эффектами.

F# и функциональное программирование для C# разработчика

Чистые функции



Чистые функции можно представлять как функции в математике - соответствие элемента одного множества элементу другого множества.

F# и функциональное программирование для C# разработчика

Чистые функции



Более простым языком - чистая функция преобразует одно значение в другое значение. Больше чистая функция не оказывает никакого влияния на внешний мир.

F# и функциональное программирование для C# разработчика

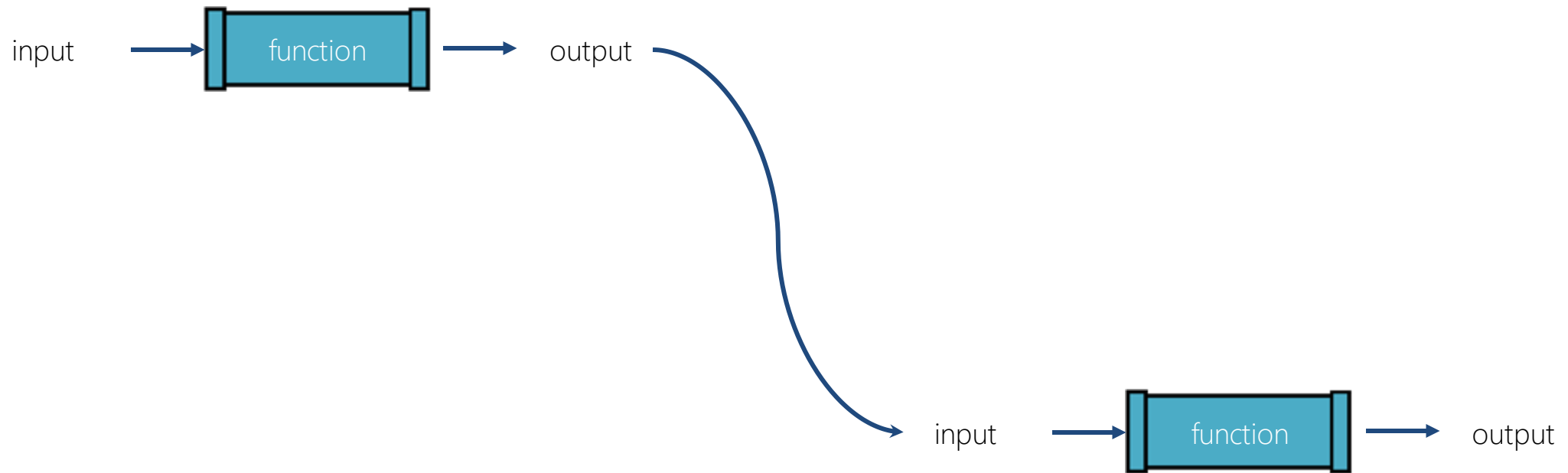
Чистые функции

```
let sliceApple apple =  
    ...  
    slicedApple
```

Чистая функция **полностью** определяется своей сигнатурой.

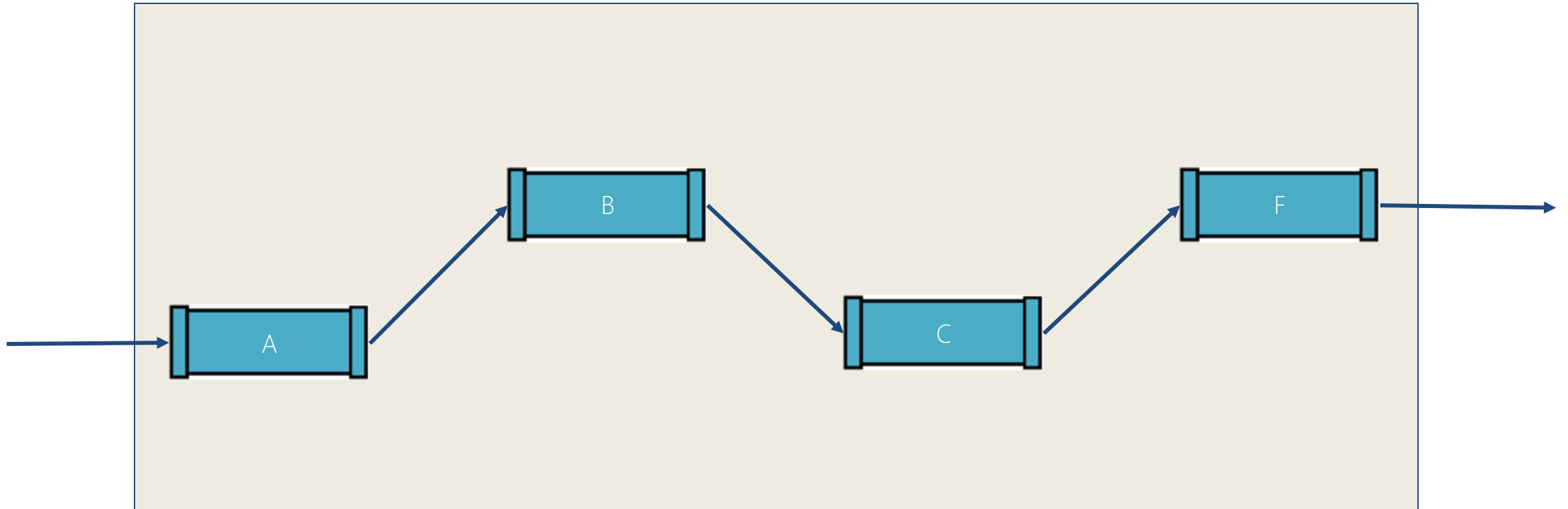
F# и функциональное программирование для C# разработчика

Композиция функций



F# и функциональное программирование для C# разработчика

Композиция функций



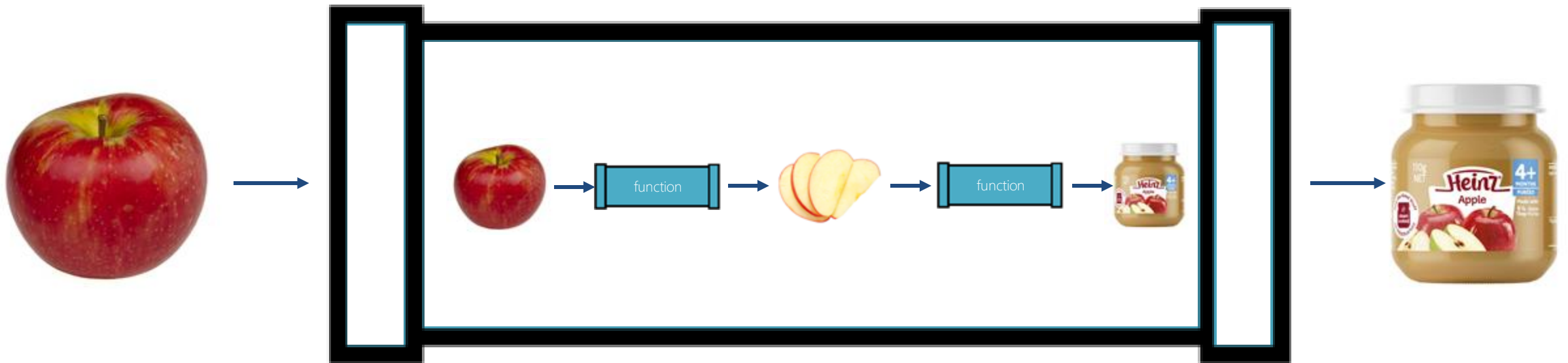
F# и функциональное программирование для C# разработчика

Композиция функций



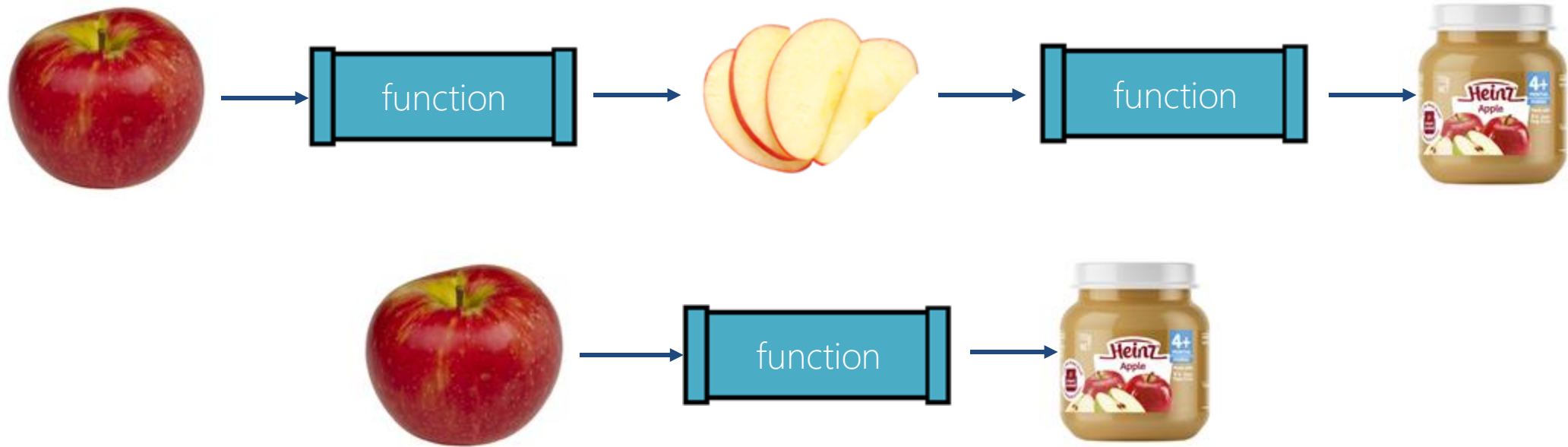
F# и функциональное программирование для C# разработчика

Композиция функций



F# и функциональное программирование для C# разработчика

Композиция функций



```
let makeApplesauce = sliseApple >> boilApple
```

```
let applesauce = makeApplesauce apple
```

F# и функциональное программирование для C# разработчика

Применение принципов ФП в C#



F# и функциональное программирование для C# разработчика

Функциональное программирование в C#

LINQ

Immutable
members

Pattern matching

Delegates

Tuples

Nullable reference
types

F# и функциональное программирование для C# разработчика

Чистые функции



Чистые функции:

- при одинаковых входных данных всегда возвращают один и тот же результат;
- не оказывают побочных эффектов;
- определяют все входные данные и все возможные результаты в сигнатуре.

F# и функциональное программирование для C# разработчика

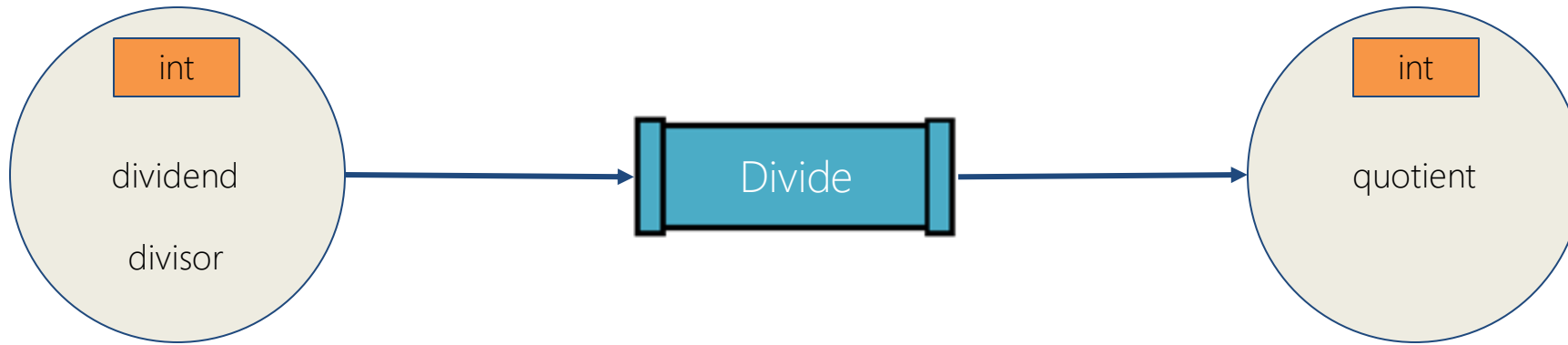
Чистые функции

0 references

```
public int Divide(int dividend, int divisor)
{
    return dividend / divisor;
}
```

F# и функциональное программирование для C# разработчика

Чистые функции



Сигнатура метода утверждает, что передав в функцию 2 аргумента из множества целых чисел, мы получим значение из множества целых чисел.

F# и функциональное программирование для C# разработчика

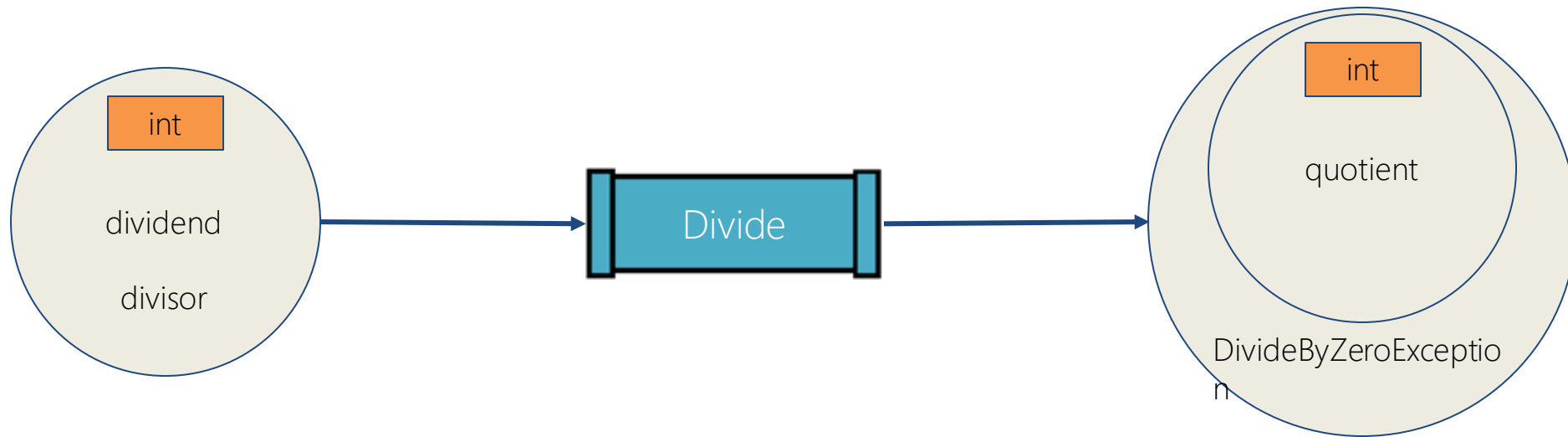
Чистые функции

```
0 references  
public int Divide(int dividend, int divisor)  
{  
    return dividend / divisor;  
}
```

Если в качестве divisor будет передано значение "0", то результатом выполнения метода будет не int, а DivideByZeroException.

F# и функциональное программирование для C# разработчика

Чистые функции

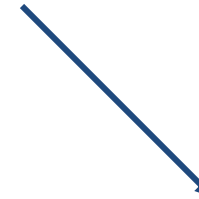
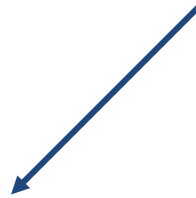


F# и функциональное программирование для C# разработчика

Чистые функции

0 references

```
public int Divide(int dividend, int divisor)
```



0 references

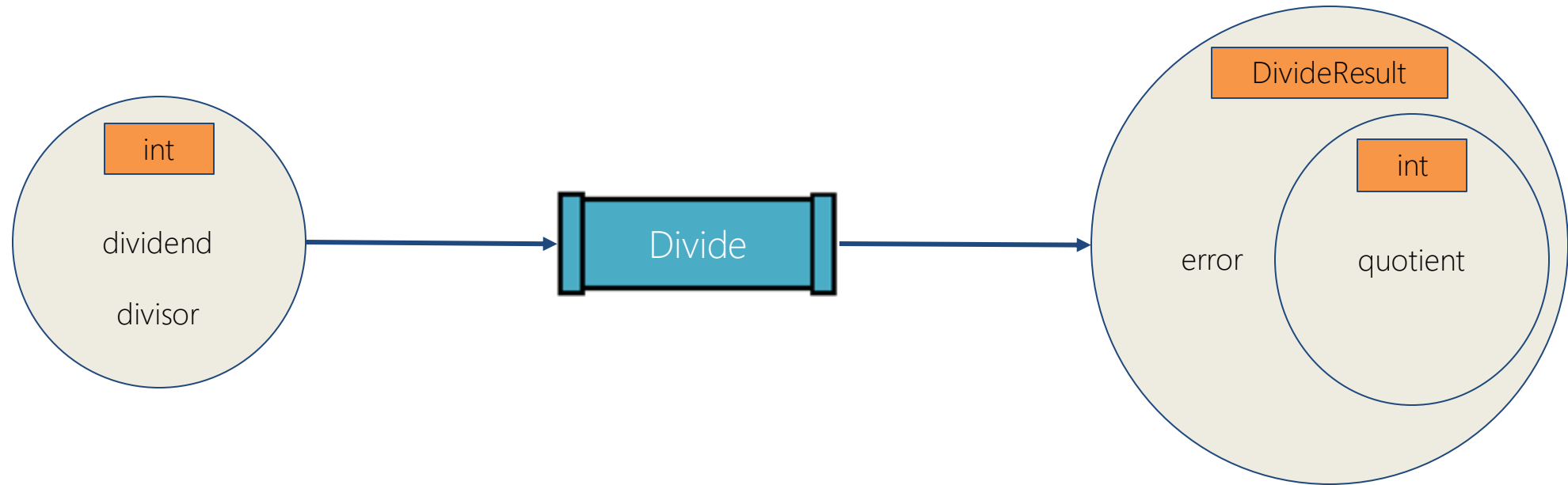
```
public DivideResult Divide(int dividend, int divisor)
```

0 references

```
public int Divide(int dividend, NonZeroInt divisor)
```

F# и функциональное программирование для C# разработчика

Чистые функции

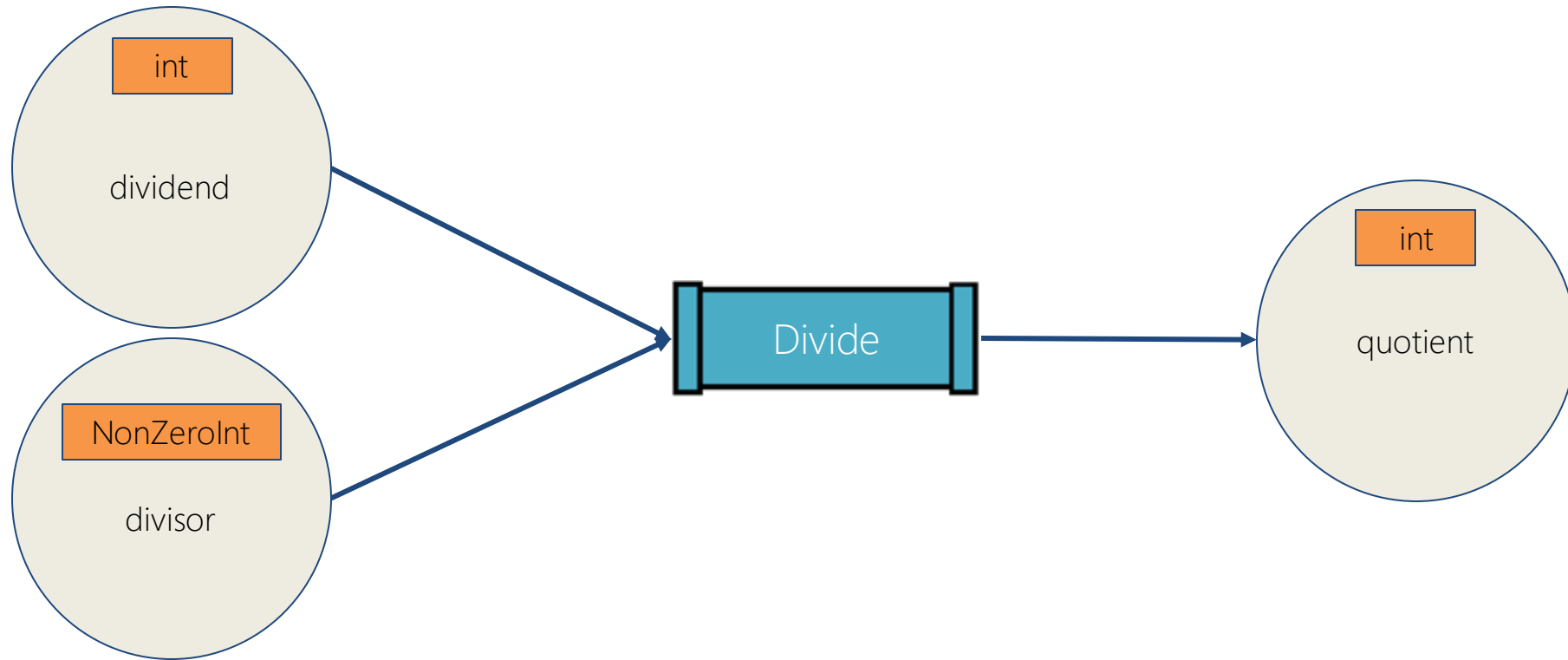


0 references

```
public DivideResult Divide(int dividend, int divisor)
```

F# и функциональное программирование для C# разработчика

Чистые функции



0 references

```
public int Divide(int dividend, NonZeroInt divisor)
```

F# и функциональное программирование для C# разработчика

Чистые функции в C#

Преимущества при использовании чистых функций:

- повышается читаемость кода;
- упрощается Unit тестирование;
- появляется возможность применения мемоизации.

F# и функциональное программирование для C# разработчика

Чистые функции в C#

Программирование с помощью чистых функций в C# приносит наибольшую пользу когда используется для написания логики предметной области.

F# и функциональное программирование для C# разработчика

Неизменяемость в C#

```
1 reference
class CreateIncomeTransactionCommand
{
    1 reference
    public DateTime Created { get; }
    1 reference
    public Guid TransactionId { get; }
    1 reference
    public Guid AccountId { get; }
    1 reference
    public Guid UserId { get; }
    1 reference
    public string Currency { get; }
    1 reference
    public decimal Amount { get; }

    0 references
    public CreateIncomeTransactionCommand(
        DateTime created, Guid transactionId, Guid accountId,
        Guid userId, string currency, decimal amount)
    {
        Created = created;
        TransactionId = transactionId;
        AccountId = accountId;
        UserId = userId;
        Currency = currency;
        Amount = amount;
    }
}
```

F# и функциональное программирование для C# разработчика

Неизменяемость в C#

```
1 reference
class Account
{
    1 reference
    public User Owner { get; }
    1 reference
    public Currency Currency { get; }
    2 references
    public List<TransactionBase> Transactions { get; }
    2 references
    public decimal Balance { get; set; }
    1 reference
    public string Name { get; set; }

    0 references
    public Account(User owner, Currency currency, string name)
    {
        Balance = 0m;
        Transactions = new List<TransactionBase>();
        Owner = owner;
        Currency = currency;
        Name = name;
    }

    0 references
    public void AddTransaction(TransactionBase transaction)
    {
        Transactions.Add(transaction);
        Balance += transaction.AbsoluteAmount;
    }
}
```

F# и функциональное программирование для C# разработчика

Неизменяемость в C#

Чем у объекта больше изменяемого состояния, тем за большим количеством деталей нужно следить при работе с этим объектом.

Неизменяемость позволяет упростить работу с объектом.

F# и функциональное программирование для C# разработчика

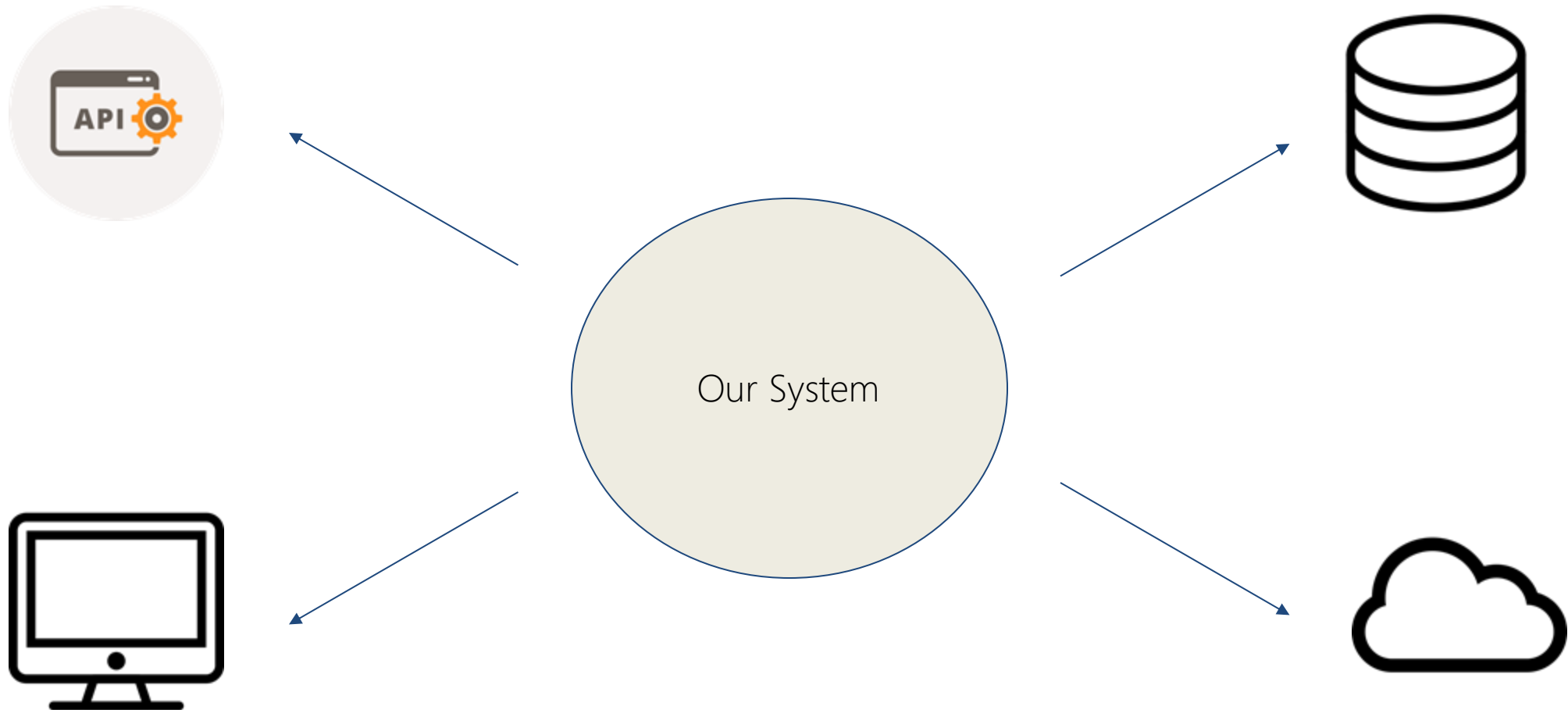
Неизменяемость в C#

Чистые функции и неизменяемость это связанные понятия.

Если у объекта нет изменяемого состояния и в системе нет глобального изменяемого состояния - то у методов этого объекта не будет побочных эффектов.

F# и функциональное программирование для C# разработчика

Побочные эффекты



F# и функциональное программирование для C# разработчика

Command–query separation

Методы разделяются на command и query.

Command - не возвращает значений, но приводит к возникновению побочных эффектов.

Query - чистые функции, возвращают значение, не оказывают побочных эффектов, при одинаковых входных данных возвращают одинаковые значения.

F# и функциональное программирование для C# разработчика

Command–query separation

0 references

```
public void AddTransaction(TransactionBase transaction)
{
    Transactions.Add(transaction);
    Balance += transaction.AbsoluteAmount;
}

public CashFlow CalculateCashFlowForPeriod(Period period)
{
    var transactionsForPeriod = Transactions
        .Where(x => x.Created > period.From && x.Created < period.To)
        .ToList()
        .AsReadOnly();

    return Statistics.CalculateCashFlow(transactionsForPeriod);
}
```

F# и функциональное программирование для C# разработчика

CQS Исключения

```
var stack = new Stack<int>();
```

```
stack.Push(1);
```

```
var topValue = stack.Pop();
```

0 references

```
public interface IUserRepository
```

```
{
```

```
    // adds entity to DB, returns Id of created entity
```

0 references

```
    int AddUser(User user);
```

```
}
```

F# и функциональное программирование для C# разработчика

Применение функциональных подходов

На уровне кода

- чистые функции
- неизменяемые объекты-сообщения
- избежание null значений
- избежание исключений
- более широкое использование системы типов

На архитектурном уровне

- Immutable Architecture
- Event Sourcing
- CQRS
- Pipes and Filters
- ...

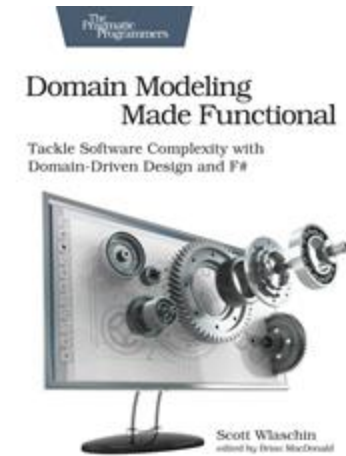
F# и функциональное программирование для C# разработчика

Дальнейшее изучение

- <https://fsharpforfunandprofit.com/>
- <https://fsharp.org/>
- <http://tomasp.net/blog/types-and-math.aspx/>
- <http://www.fsharpworkshop.com/>
- <https://enterprisecraftsmanship.com>

F# и функциональное программирование для C# разработчика

Дальнейшее изучение



Q&A

Смотрите наши уроки в видео формате

.NET Developer



На ITVDN вы найдете подборку видео курсов и вебинаров по специальности .NET Developer. Заходите на сайт [ITVDN.com](http://itvdn.com) и смотрите наши видео уроки прямо сейчас!

Информационный видеосервис для разработчиков программного обеспечения

