Лекция 5: Объектно-ориентированное программирование в F#

Юрий Литвинов
y.litvinov@spbu.ru

1. Зачем объектно-ориентированное программирование в F#

F# — функциональный язык, а функциональные языки прекрасно обходятся и без ООП. Однако F# по мнению его автора на самом деле объектно-ориентированный язык с некоторыми функциональными возможностями, и рекомендует на нём программировать в основном в объектно-ориентированном стиле. Почему это для F# важно:

- Люди привыкли к объектно-ориентированным языкам, поэтому наличие объектно-ориентированных возможностей по задумке должно было сделать F# более популярным и облегчить переход на него. Но не то чтобы на F# кинулись программировать миллионы C#-программистов, так что это не то чтобы самый главный пункт «за» ООП.

- Что реально нужно на практике — это возможность интеграции с кодом, написанным на C# (и других .NET-языках). C# знать ничего не знает про каррирование и размеченные объединения, поэтому нужно иметь возможность писать на F# код, который бы всё это не использовал и напрямую транслировался в C#-овые языковые конструкции. Вместе с тем, в F# требуются языковые механизмы, позволяющие естественным образом вызывать код из библиотек, написанных на C# (например, стандартной библиотеки .NET).

- Вообще, ООП позволяет делать приложениям аккуратную архитектуру, применять паттерны и т.п. Большие системы в целом удобнее проектировать в объектно-ориентированном стиле, поэтому объектно-ориентированный язык с некоторыми функциональными возможностями для написания красивых и выразительных тел методов пришёлся бы очень кстати. Аналогичным образом, кстати, появилась и Scala — «гибридный» язык для Java-мира, любимый в своё время даже в production, но так и не сумевший занять значимую часть рынка (собственно как и F#).

Однако использование ООП в F# также не лишено недостатков:

- ООП не очень дружит с системой вывода типов — ограничения вида «любой тип, у которого есть такой-то метод» F# обрабатывать не умеет, что требует много лишних аннотаций типов, делает код более громоздким.

- Все стандартные генерик-операции типа сравнения, печати и т.п. определены только для встроенных типов, для произвольных классов работать не будут и требуют ручного переопределения.

2. Методы у типов

2.1. Методы у записей

Хорошая новость в том, что F# не требует выбирать между чистым ООП и чистым ФП — даже у встроенных типов могут быть объявлены методы. Например, записи с методами:

```
type Vector = {x : float; y : float} with
    member v.Length = sqrt(v.x * v.x + v.y * v.y)

let vector = {x = 1.0; y = 1.0}
let length = vector.Length

type Vector with
    member v.Scale k = {x = v.x * k; y = v.y * k}

let scaled = vector.Scale 2.0
```

Тут у записи Vector определяется метод (на самом деле свойство) Length. «v» перед именем свойства означает ссылку на объект, от которого оно вызывается, как «this» в C#, но тут можно самому выбирать имя переменной, по которой мы ссылаемся на объект. Обычно используют что-нибудь вроде v (от «Value»), x (просто переменная в математическом смысле), this (чтобы было похоже на нормальные языки), _ — если передаваемый объект в методе не используется (справедливости ради, тогда стоит проверить, может ли метод быть статическим).

Дальше — пример вызова, вроде ничего необычного. А дальше — тот факт, что методы к типам можно добавлять и после их объявления. Метод Scale (на сей раз полноценный метод, с одним целочисленным параметром) дописан к той же записи Vector, что мы только что объявили и уже успели попользовать. Тут Scale возвращает новый вектор (записи сами по себе немутабельны, поэтому в F# практически всё, что по смыслу должно менять свой объект, возвращает изменённую копию). Работает это как методы-расширения в C#.

Кстати, записи с методами по смыслу близки к ООПшным классам, но при этом с ними работают все встроенные в язык вещи типа структурного сравнения. Правда, наследоваться они не могут. А вот реализовывать интерфейсы — вполне.

2.2. Методы у других типов

У размеченных объединений тоже могут быть методы:

```
type Tree<'a> =
    | Tree of 'a * Tree<'a> * Tree<'a>
    | Tip of 'a
```

```
    with
        member t.Size =
            match t with
            | Tree(_, l, r) -> 1 + l.Size + r.Size
            | Tip _ -> 1
```

Тут всё так же, как и с записями — мы имеем доступ с «своему» объекту, можем быть свойством или честным методом, и если надо что-то поменять — возвращаем копию.

Методы-расширения F# тоже поддерживает, поэтому можно делать методы для вообще любого типа, хоть библиотечных, хоть Int32:

```
type System.Int32 with
    member i.IsPrime =
        let limit = i |> float |> sqrt |> int
        let rec check j =
            j > limit or (i % j <> 0 && check (j + 1))
        check 2

printfn "%b" (5).IsPrime
printfn "%b" (8).IsPrime
```

Конкретно для чисел, чтобы вызывать их методы-расширения, числа надо писать в скобках, иначе лексер решит, что точка относится к числу (и оно вещественное). Но в остальном всё как обычно. Увлекаться методами-расширениями не стоит, потому что бывает сложно понять, откуда взялся тот или иной метод, но если это концептуально целостно и оправданно (например, как в LINQ), то почему нет.

2.3. Технические детали

Статические методы описываются, что неудивительно, с помощью ключевого слова static:

```
type Vector = {x : float; y : float} with
    static member Create x y = {x = x; y = y}

let vector = Vector.Create 1.0 1.0

type System.Int32 with
    static member IsEven x = x % 2 = 0

printfn "%b" <| System.Int32.IsEven 10
```

Работают они так же, как в C# — не принимают ссылку на объект, поэтому не требуют указания имени объекта (никаких странных букв с точкой перед именем метода).

Методы часто используются со «свободными» функциями:

```
type Vector = {x : float; y : float} with
    static member Create x y = {x = x; y = y}
```

```
let length (v : Vector) = sqrt(v.x * v.x + v.y * v.y)

type Vector with
    member v.Length = length v

printfn "%f" <| (Vector.Create 1.0 1.0).Length
printfn "%f" <| (length (Vector.Create 1.0 1.0))
```

Тут полезную работу делает функция, а метод просто делегирует вызов в функцию. Это даёт возможность пользователю выбирать, использовать ему метод или соответствующую свободную функцию. Функции более «функциональны», с ними хорошо работает вывод типов, функции высших порядков и т.п., но методы в целом удобней (хотя бы тем, что среда разработки покажет список методов, если введёте точку после имени объекта).

Ещё методы могут использовать каррирование при делегировании запроса функции:

```
open Operators

type Vector = {x : float; y : float} with
    static member Create x y = {x = x; y = y}

let transform v rotate scale =
    let r = System.Math.PI * rotate / 180.0
    { x = scale * v.x * cos r - scale * v.y * sin r;
      y = scale * v.x * sin r + scale * v.y * cos r }

type Vector with
    member v.Transform = transform v

printfn "%A" <| (Vector.Create 1.0 1.0).Transform 45.0 2.0
```

Transform — метод с двумя параметрами, хотя технически это свойство типа «функция». При вызове методов каррирование, разумеется, тоже вполне можно применять.

Ничто не мешает методу с несколькими аргументами принимать эти аргументы либо в каррированной форме, либо в форме кортежа:

```
type Vector with
    member v.TupledTransform (r, s) = transform v r s
    member v.CurriedTransform r s = transform v r s

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.CurriedTransform 45.0 2.0
```

Кортежная форма не позволяет применять каррирование при вызове, что несколько против идеологии функционального программирования (потому что с точки зрения синтаксиса у метода один аргумент, просто он пара).

Однако кортежная форма метода генерируется компилятором в честный метод .NET, так что будет самым обычным способом понята из кода на других языках. Плюс к тому, компилятор позволяет для кортежной формы ещё разные удобные штуки, которые для каррированной формы не очень осмысленны. Например, именованные аргументы:

```
type Vector with
    member v.TupledTransform (r, s) =
       transform v r s

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (r = 45.0, s = 2.0)
printfn "%A" <| v.TupledTransform (s = 2.0, r = 45.0)
```

Именованные аргументы работают как в C# — могут идти в любом порядке. Они делают код несколько более документированным и избавляют от ошибок типа «перепутали аргументы местами», но это громоздко, поэтому ни в F#, ни в C# именованные аргументы не прижились. Напомним, что в Аде можно было только в именованной форме аргументы передавать, и это считалось важной особенностью для повышения надёжности.

Ещё в кортежной форме можно передавать опциональные аргументы:

```
type Vector with
    member v.TupledTransform (r, ?s) =
        match s with
        | Some scale -> transform v r scale
        | None -> transform v r 1.0

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.TupledTransform (90.0)
```

Если аргумент помечен вопросиком, то вместо аргумента принимается Option, где внутри лежит переданный аргумент, если он был передан, или Option равен None, если не был. Для распаковки опционального аргумента можно использовать match, как в примере, или специально для этого созданную функцию defaultArg, как тут:

```
type Vector with
    member v.TupledTransform (r, ?s) =
       transform v r <| defaultArg s 1.0

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.TupledTransform (90.0)
```

Она принимает опциональный аргумент и значение, которое возвращается, если аргумент не установлен (то есть None). Больше ничего интересного она не делает, и легко пишется вручную.

Обратите внимание, что опциональные аргументы в каррированной форме в принципе невозможны, потому что компилятор никак не может отличить непереданный опциональный аргумент от частичного применения функции. У кортежной формы нет такой неоднозначности, потому что частично применить её нельзя, у неё и так один аргумент.

По тем же причинам методы в каррированной форме перегружать нельзя, а в кортежной — можно:

```
type Vector with
    member v.TupledTransform (r, s) =
        transform v r s
    member v.TupledTransform r =
        transform v r 1.0

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.TupledTransform (90.0)
```

В целом и опциональные аргументы, и перегрузка — это сомнительная практика и для F#, и для объектно-ориентированных языков тоже, особенно их комбинирование, поскольку это создаёт у программиста, пользующегося такими методами, неуверенность в том, что же именно будет вызвано. Иногда опциональные аргументы оправданны — когда они редко используются, имеют разумное умолчание, и всё-таки иногда требуют каких-то отличных от умолчаний значений. Однако часто проще объявить метод с другим названием — это хотя бы сделает явным в месте вызова, какой именно метод вызывается. То же касается перегрузок — часто осмысленно сделать два независимых метода, чтобы при вызове было понятно, что именно вызовется, и избежать дурацких ошибок типа «забыли указать параметр — вызвался совсем другой код».

Если суммаризовать, кортежная и каррированная формы записи метода отличаются следующим:

Кортежная:

- метод можно вызывать из кода на любом языке .NET, он транслируется в самый обычный метод;
- можно использовать опциональные и именованные аргументы, перегрузки.

Каррированная:

- позволяет использовать частичное применение метода;
- хорошо работает с функциями высших порядков и всякими чисто функциональными штуками.

В целом правило таково, что кортежную форму применяют для всех public-типов сборки (прежде всего классов, поскольку более хитрые типы другие языки всё равно не поймут), каррированную форму, поддержанную свободными функциями, принимают внутри F# кода, там, где всё равно нельзя их вызвать извне.

Свободные функции имеют ряд преимуществ по сравнению с методами, поэтому их применение обычно предпочтительнее. Например, вывод типов:

```
type Vector = {x : float; y : float} with
    member v.Length = v.x * v.x + v.y * v.y |> sqrt

let length v = v.x * v.x + v.y * v.y |> sqrt

let compareWrong v1 v2 =
    v1.Length < v2.Length

let compareRight v1 v2 =
    length v1 < length v2
```

compareWrong не скомпилируется, потому что в месте объявления функции нет информации о типах v1 и v2, так что проверить корректность обращения к Length компилятор не может. compareRight скомпилируется, потому что функция length знает, что принимает аргумент типа Vector (кстати, знает по обращению к полям записи — для классов это не работает, а для записей вполне, потому что записи просты по своей структуре, не могут наследоваться от непонятно чего и не могут получить полей непонятно откуда).