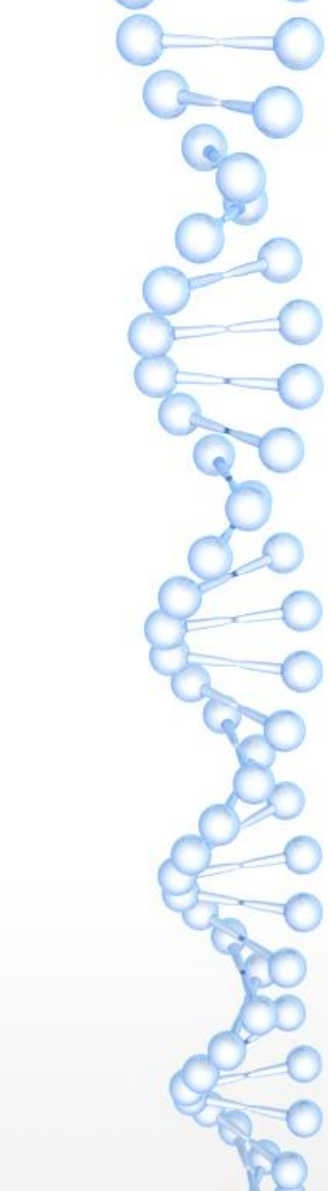
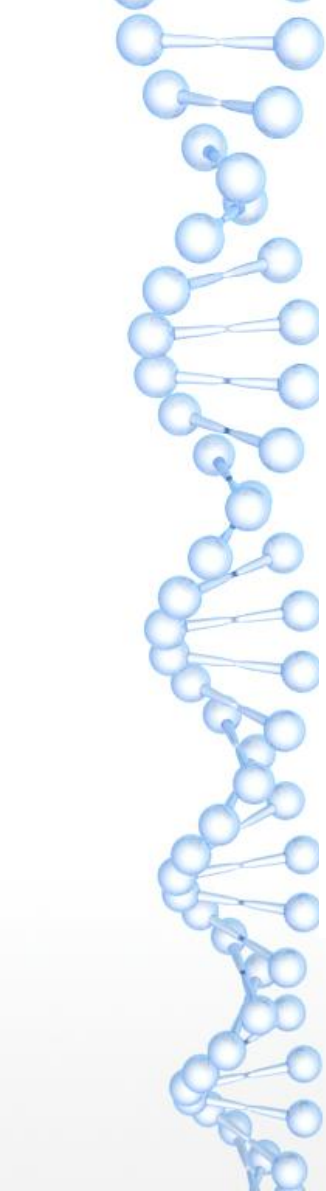


Функциональное и логическое программирование

Лекция 9. Объектно-ориентированное программирование

CLI



Основой платформы .NET является спецификация общеязыковой инфраструктуры (Common Language Infrastructure, CLI). Это спецификация среды времени выполнения, которая фактически выполняет ваш код F#. Компилятор F# генерирует двоичный файл, который называется *сборкой* (*assembly*) и содержит инструкции на высокоуровневом языке ассемблера *MSIL* (Microsoft Intermediate Language – промежуточный язык компании Microsoft).

Реализация CLI компилирует инструкции MSIL в машинный код во время выполнения программы, чтобы обеспечить более высокую скорость работы, чем при интерпретации. Компиляция выполняется динамически, на лету («just-in-time», или JIT).

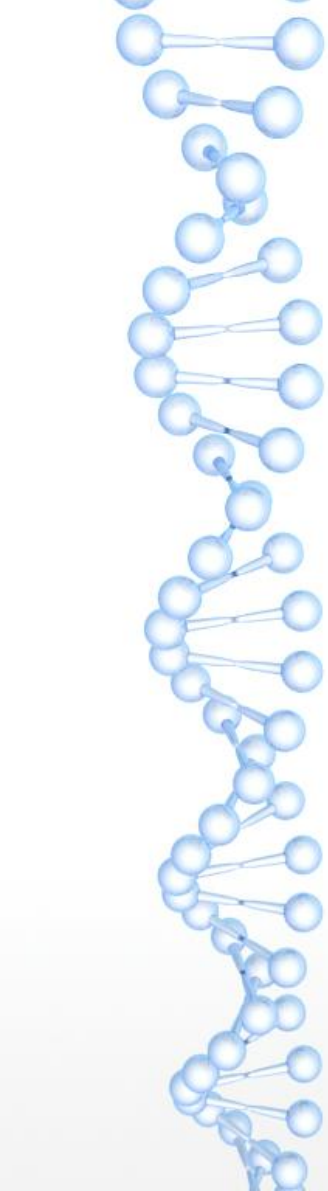
CLI

Код, который компилируется в MSIL и затем выполняется JIT-компилятором, называется *управляемым кодом (managed code)*, в противоположность *неуправляемому коду (unmanaged code)*, который является обычным машинным кодом (как правило, написанным на языке C или C++).

Выполнение управляемого кода с помощью CLI имеет несколько преимуществ по сравнению с компиляцией кода на языке F# непосредственно в машинный код:

Взаимодействие с другими языками

Без обобщенного языка (MSIL) было бы намного сложнее обеспечить возможность совместного использования одного и того же кода в проектах на языках C#, VB.NET и F#.



CLI

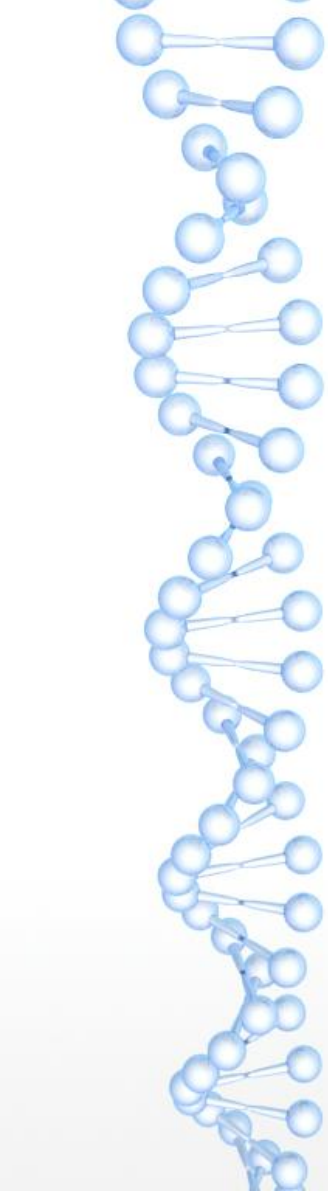
Кроссплатформенность

Благодаря тому что спецификация CLI может быть реализована в любой операционной системе, появляется возможность выполнять код .NET на других платформах, таких как Microsoft Silverlight, на приставках X-Box с установленными XNA и .NET Compact Framework[1] и даже в Linux с помощью проекта Mono.

Независимость от аппаратной архитектуры

Компиляция в машинный код производится JIT-компилятором уже во время работы программы, поэтому вам не нужно заботиться о том, чтобы скомпилировать исходный код под определенную аппаратную архитектуру, такую как x86 или x64.

Кроме того, платформа .NET предлагает механизм автоматической сборки мусора. Это освобождает программиста от необходимости следить за выделением памяти и своевременным ее освобождением, благодаря чему (практически полностью) ликвидируется целый класс ошибок.



Сборка мусора

Если по каким-то причинам вам приходится вручную выделять и освобождать ресурсы, рекомендуется использовать интерфейс `IDisposable`. (Интерфейсы будут рассматриваться ниже в этой главе.) Интерфейс `IDisposable` содержит всего один метод, `Dispose`, который освобождает ресурсы. В своей реализации метода `Dispose` вы можете предусмотреть выполнение всех необходимых операций по освобождению ресурсов, таких как закрытие файла:

```
type IDisposable =
    interface
        abstract member Dispose : unit -> unit
    end
```

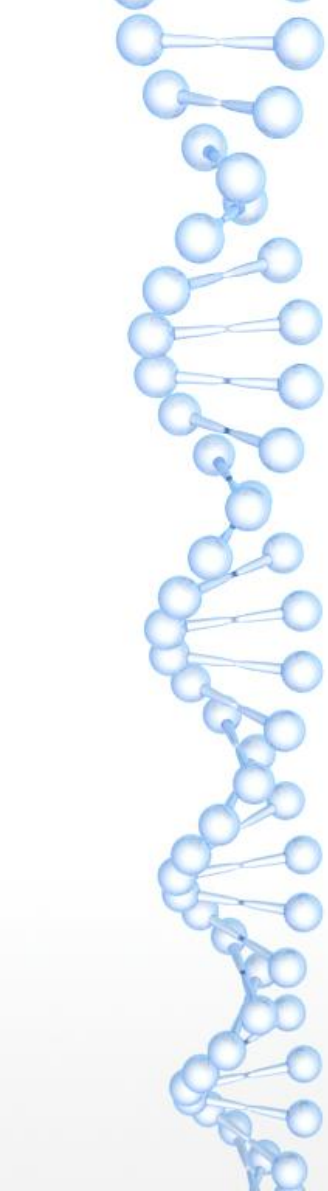
Но необходимость помнить о вызове метода `Dispose` представляет собой определенную проблему. Фактически этот способ точно так же может приводить к ошибкам, как ручное управление памятью. К счастью, в языке F# есть синтаксический сахар, который может придти на помощь.

При использовании ключевого слова `use` компилятор F# автоматически будет освобождать ресурсы, как только работа с ними будет закончена, то есть после выхода из области видимости. Синтаксически ключевое слово `use` эквивалентно ключевому слову `let`.

Сборка мусора

```
> // Реализация интерфейса IDisposable
open System
open System.IO
open System.Collections.Generic

type MultiFileLogger() =
    do printfn "Constructing..."
    let m_logs = new List<StreamWriter>()

    member this.AttachLogFile file =
        let newLogFile = new StreamWriter(file, true)
        m_logs.Add(newLogFile)

    member this.LogMessage (msg : string) =
        m_logs |> Seq.iter (fun writer -> writer.WriteLine(msg))

    interface IDisposable with
        member this.Dispose() =
            printfn "Cleaning up..."
            m_logs |> Seq.iter (fun writer -> writer.Close())
            m_logs.Clear();;
```

```
> // Некоторый программный код, использующий MultiFileLogger
let task1() =
    use logger = new MultiFileLogger()
    // ...
    printfn "Exiting the function task1.."
    ();;

val task1 : unit -> unit

> task1();;
Constructing...
Exiting the function task1..
Cleaning up...
val it : unit = ()
```



Интерфейсы

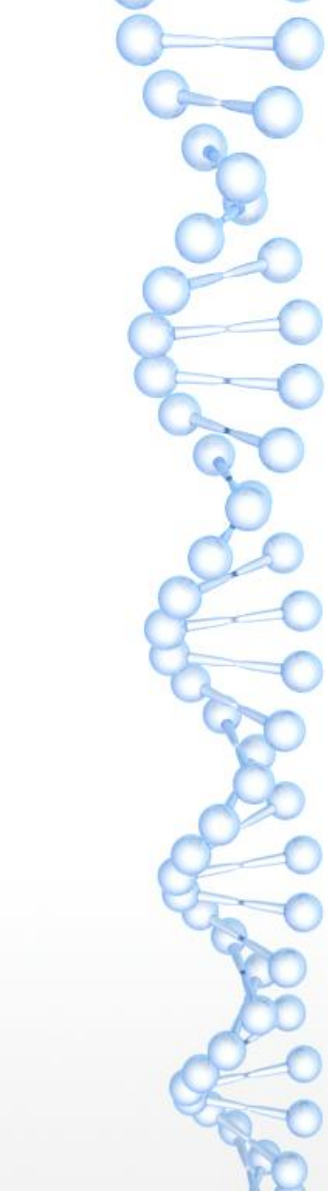
```
type Tastiness =
    | Delicious
    | SoSo
    | TrySomethingElse

type IConsumable =
    abstract Eat : unit -> unit
    abstract Tastiness : Tastiness

// Совет: ешьте по одному каждый день
type Apple() =
    interface IConsumable with
        member this.Eat() = printfn "Tastey!"
        member this.Tastiness = Delicious

// Это не так вкусно, но голод, как известно, не тетка
type CardboardBox() =
    interface IConsumable with
        member this.Eat() = printfn "Yuck!"
        member this.Tastiness = TrySomethingElse
```

Интерфейсы

```
> let apple = new Apple();;

val apple : Apple

> apple.Tastiness;;

  apple.Tastiness;;
  ------^^^^^^^^^^
stdin(81,7): error FS0039: The field, constructor or member 'Tastiness' is not
defined.
```

(Поле, конструктор или элемент "Tastiness" не определен.)

```
> let iconsumable = apple :> IConsumable;;

val iconsumable : IConsumable
```



Интерфейсы

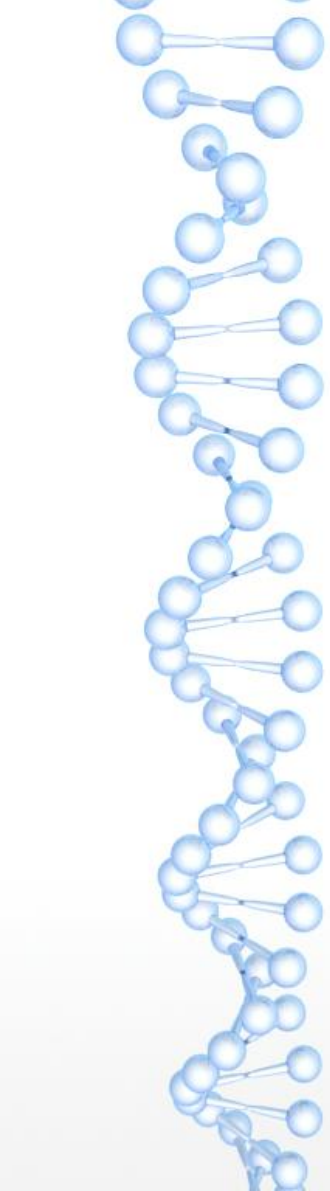
```
> // Объявление типа, который автоматически определяется как интерфейс
type IDoStuff =
    abstract DoThings : unit -> unit;;

// Явное определение интерфейса
type IDoStuffToo =
    interface
        abstract member DoThings : unit -> unit
    end;;

type IDoStuff =
  interface
    abstract member DoThings : unit -> unit
  end

type IDoStuffToo =
  interface
    abstract member DoThings : unit -> unit
  end
```

Интерфейсы могут наследовать друг друга, при этом производный интерфейс расширяет перечень методов и свойств базового интерфейса. Однако классы, реализующие этот интерфейс, должны реализовать всю цепочку наследования, реализуя каждый интерфейс полностью.

Интерфейсы

```
> // Наследование интерфейсов
type IDoStuff =
    abstract DoStuff : unit -> unit

type IDoMoreStuff =
    inherit IDoStuff

    abstract DoMoreStuff : unit -> unit;;

... вырезано ...
```

```
> // Работающий вариант
type Bar() =
    interface IDoStuff with
        override this.DoStuff() = printfn "Stuff getting done..."

    interface IDoMoreStuff with
        override this.DoMoreStuff() = printfn "More stuff getting done...";;
```

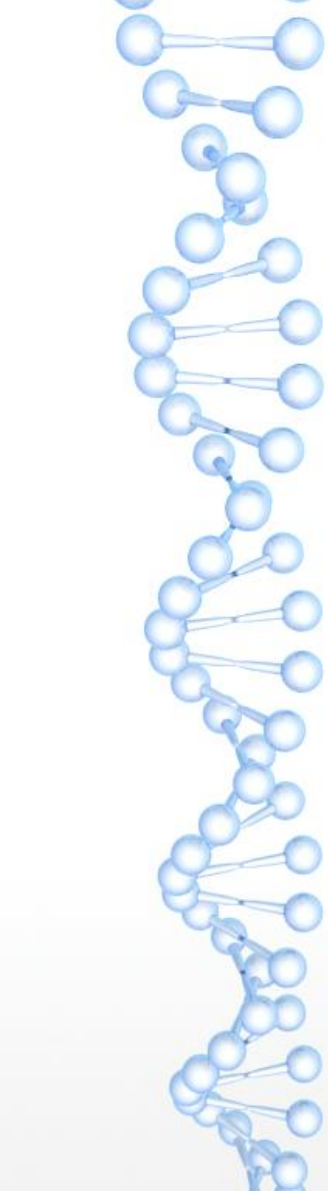


Интерфейсы

Интерфейсы играют важную роль в .NET, но иногда бывает необходимо просто реализовать некоторый интерфейс без определения нового класса. Например, чтобы отсортировать элементы коллекции типа `List<_>`, необходимо определить тип, реализующий интерфейс `IComparer<'a>`. Как только начнет появляться необходимость сортировать эту коллекцию типа `List<_>` различными способами, вы быстро обнаружите, что код загромождается типами, которые не делают ничего, кроме реализации единственного метода сравнения двух объектов.

В языке F# существует возможность использовать *объектные выражения (object expressions)*, которые создают анонимные классы и возвращают их экземпляры. (Термин «анонимный класс» означает лишь то, что компилятор сгенерирует класс, имя которого для вас останется неизвестным.) Это упрощает создание одноразовых типов аналогично тому, как лямбда-выражения упрощают создание одноразовых функций.

Синтаксис создания объектного выражения представляет собой пару фигурных скобок `{ }`, начинается с ключевого слова `new`, за которым следует имя интерфейса, объявляемого точно так же, как при использовании в классах. Результатом объектного выражения является экземпляр анонимного класса.



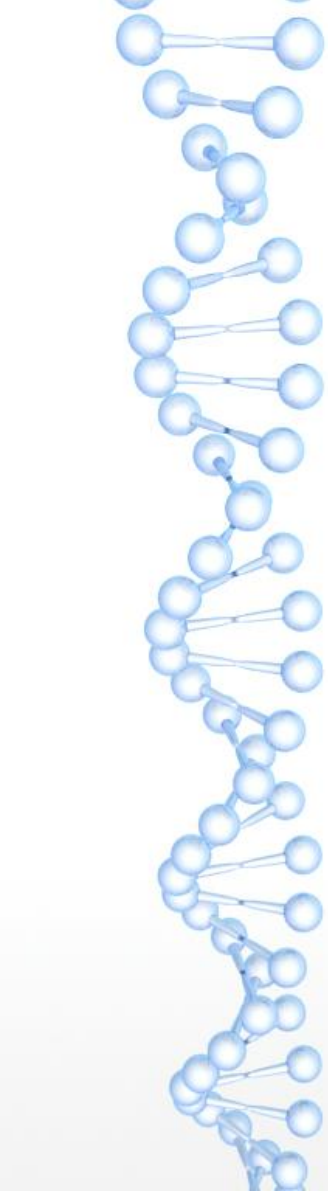
Объектные выражения

```
type Person =
    { First : string; Last : string }
    override this.ToString() = sprintf "%s, %s" this.Last this.First

let people =
    new List<_>(
        [|
            { First = "Jomo";  Last = "Fisher" }
            { First = "Brian"; Last = "McNamara" }
            { First = "Joe";   Last = "Pamer" }
        |] )

let printPeople () =
    Seq.iter (fun person -> printfn "\t %s" (person.ToString())) people

// Теперь отсортировать по фамилии

printfn "Initial ordering:"
printPeople()
```

Объектные выражения

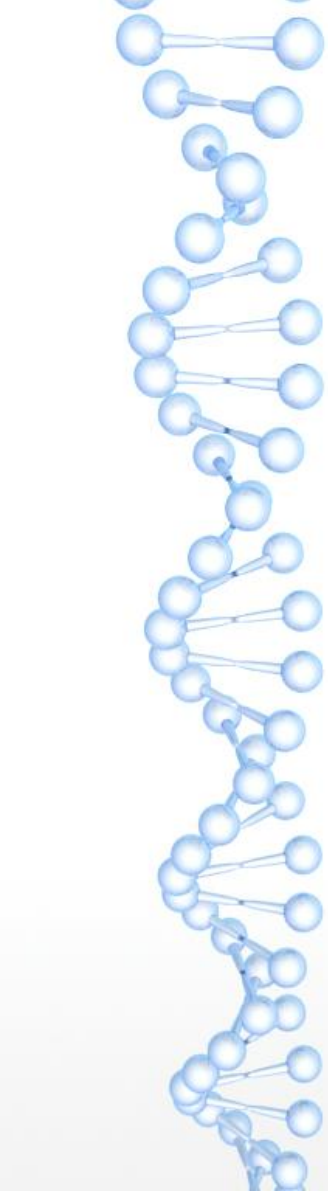
```
// Сортировка по имени
people.Sort(
    {
        new IComparer<Person> with
            member this.Compare(l, r) =
                if   l.First > r.First then  1
                elif l.First = r.First then  0
                else                        -1
    } )

printfn "After sorting by first name:"
printPeople()

// Сортировать по фамилии
people.Sort(
    {
        new IComparer<Person> with
        member this.Compare(l, r) =
            if   l.Last > r.Last then  1
            elif l.Last = r.Last then  0
            else                      -1
    } )

printfn "After sorting by last name:"
printPeople()
```

```
Initial ordering:
        Fisher, Jomo
        McNamara, Brian
        Pamer, Joe
After sorting by first name:

        McNamara, Brian
        Pamer, Joe
        Fisher, Jomo
After sorting by last name:
        Fisher, Jomo
        McNamara, Brian
        Pamer, Joe
```



Объектные выражения

```
> // Абстрактный класс
[<AbstractClass>]
type Sandwich() =
    abstract Ingredients : string list
    abstract Calories : int

// Объектное выражение производного класса
let lunch =
    {
        new Sandwich() with
        member this.Ingredients = [“Peanutbutter”; “Jelly”]
        member this.Calories = 400
    };;

type Sandwich =
  class
    abstract member Calories : int
    abstract member Ingredients : string list
    new : unit -> Sandwich
  end
val lunch : Sandwich

> lunch.Ingredients;;
val it : string list = [“Peanutbutter”; “Jelly”]
```



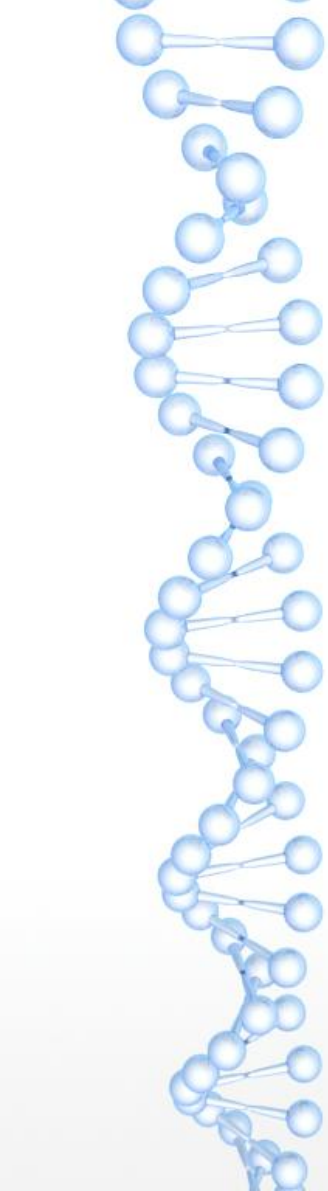
Методы расширения

Иногда при работе с некоторым типом отсутствует возможность изменить его, либо потому что нет доступа к исходному коду, либо потому что приходится использовать старую версию. В подобных ситуациях бывает необходимо хоть как-то расширить некоторый тип.

Методы расширения (extension methods) предоставляют простой механизм, который позволяет расширять возможности типов без изменения иерархии типов или изменения существующих типов. Методы расширения – это специализированная разновидность методов, которые можно добавлять к существующим типам и использовать их, как если бы они были реальными членами этих типов. Благодаря этому отпадает необходимость пересобирать библиотеки или создавать новые производные типы ради добавления нескольких методов. После добавления методов расширения появляется возможность использовать типы со всеми дополнительными методами и свойствами, даже если они изначально отсутствовали в этих типах.

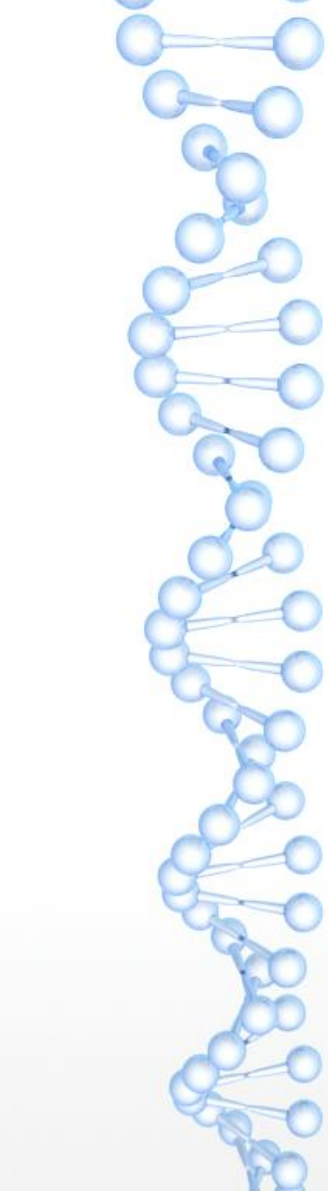
Объявление метода расширения начинается с конструкции `type` *идентификатор* `with` с последующей реализацией метода, где *идентификатор* – это полное квалифицированное имя класса.

Обратите внимание, что методы расширения в действительности не являются частью класса, поэтому они не имеют доступа к закрытым или внутренним данным. Это всего лишь методы, которые могут вызываться, как если бы они были членами класса.



Методы расширения

```
> // Расширяем тип Int32 (он же тип int)
type System.Int32 with
    member this.ToHexString() = sprintf "0x%x" this;;

type Int32 with
  member ToHexString : unit -> string

> (1094).ToHexString();;
val it : string = "0x446"
```



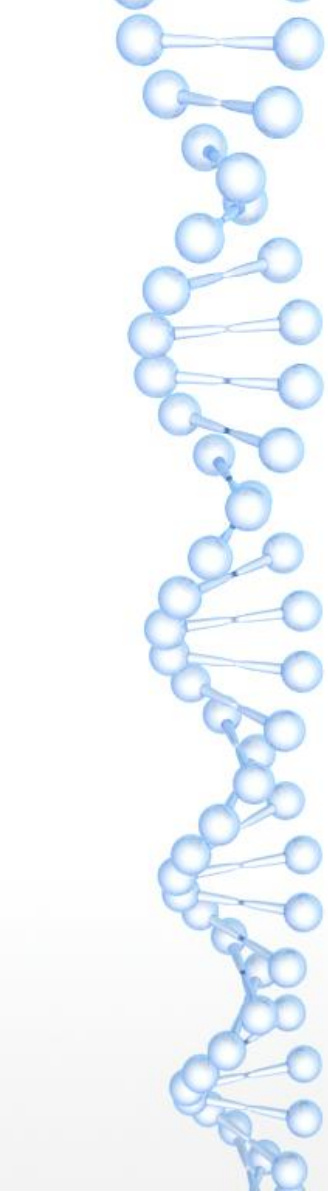
Методы расширения

```
namespace FSCollectionExtensions

open System.Collections.Generic

module List =

    /// Пропустить первые n элементов списка
    let rec skip n list =
        match n, list with
        | _, []       -> []
        | 0, list     -> list
        | n, hd :: tl -> skip (n - 1) tl

module Seq =

    /// Изменить порядок следования элементов последовательности на обратный
    let rec rev (s : seq<'a>) =
        let stack = new Stack<'a>()
        s |> Seq.iter stack.Push
        seq {
            while stack.Count > 0 do
                yield stack.Pop()
        }
```

Перечисления

```
type ChessPiece =
    | Empty  = 0
    | Pawn   = 1
    | Knight = 3
    | Bishop = 4
    | Rook   = 5
    | Queen  = 8
    | King   = 1000000
```

```
/// Создать 2-мерный массив со значениями перечисления ChessPiece
let createChessBoard() =
    let board = Array2D.init 8 8 (fun __ -> ChessPiece.Empty)

    // Расстановка пешек
    for i = 0 to 7 do
        board.[1,i] <- ChessPiece.Pawn
        board.[6,i] <- enum<ChessPiece> (-1 * int ChessPiece.Pawn)

    // Расстановка черных фигур
    [| ChessPiece.Rook; ChessPiece.Knight; ChessPiece.Bishop;
       ChessPiece.Queen; ChessPiece.King; ChessPiece.Bishop;
       ChessPiece.Knight; ChessPiece.Rook |]
    |> Array.iteri(fun idx piece -> board.[0,idx] <- piece)

    // Расстановка белых фигур
    [| ChessPiece.Rook; ChessPiece.Knight; ChessPiece.Bishop; ChessPiece.King;
       ChessPiece.Queen; ChessPiece.Bishop; ChessPiece.Knight; ChessPiece.Rook
    |]
    |> Array.iteri(fun idx piece ->
                       board.[7,idx] <- enum<ChessPiece> (-1 * int piece))
    // Вернуть шахматную доску с фигурами
    Board
```



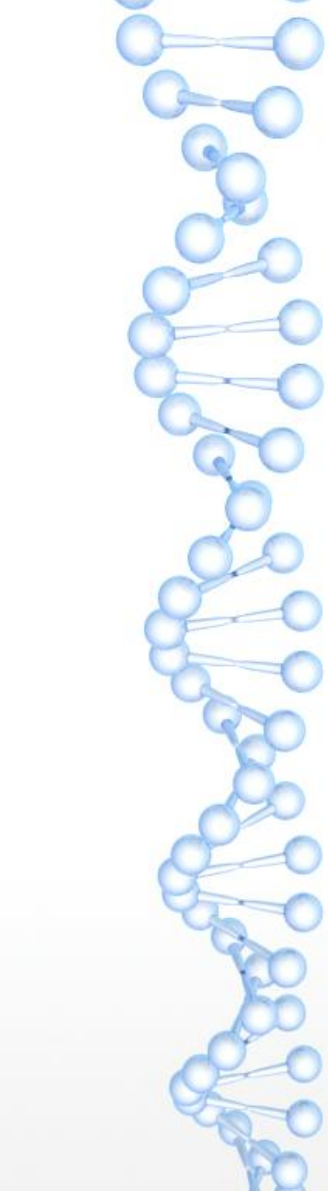
Перечисления

Ниже демонстрируется преобразование значения 42 в экземпляр `ChessPiece`:

```
let invalidPiece = enum<ChessPiece>(42)
```

Чтобы получить значение перечисления, достаточно использовать обычную функцию преобразования типа, такую как `int`:

```
let materialValueOfQueen = int ChessPiece.Queen
```



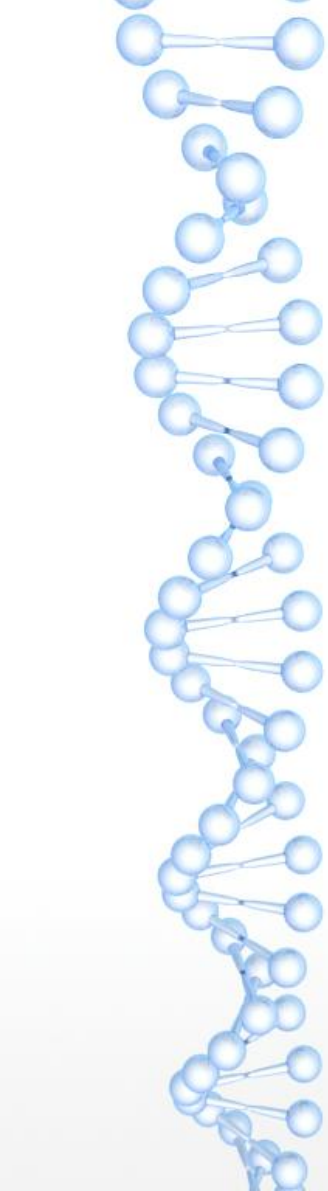
Перечисления

Ниже демонстрируется преобразование значения 42 в экземпляр `ChessPiece`:

```
let invalidPiece = enum<ChessPiece>(42)
```

Чтобы получить значение перечисления, достаточно использовать обычную функцию преобразования типа, такую как `int`:

```
let materialValueOfQueen = int ChessPiece.Queen
```



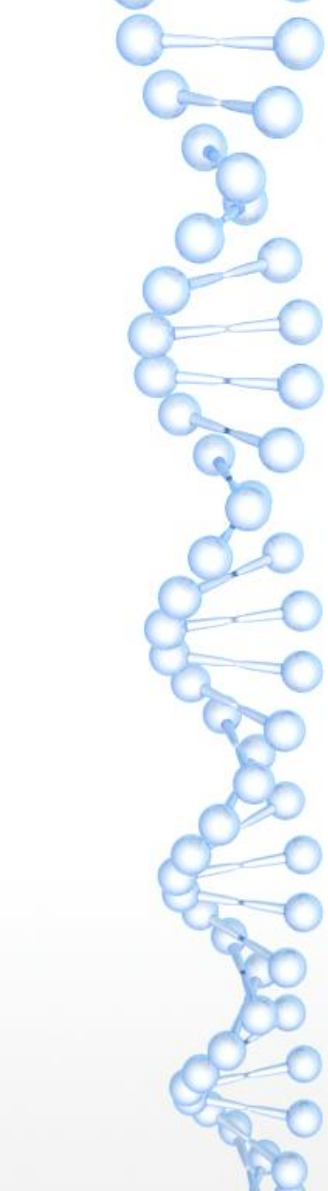
Когда использовать перечисления, а когда размеченные объединения

Перечисления и размеченные объединения имеют два существенных отличия. Во-первых, перечисления не дают гарантий безопасности. Размеченные объединения, напротив, гарантируют, что экземпляры могут иметь только одно из допустимых значений. Перечисления – это всего лишь синтаксический сахар над простыми типами. Поэтому получая экземпляр перечисления, нельзя быть полностью уверенным, что он имеет значение, присутствующее в перечислении.

Например, если значение –711 преобразовать в экземпляр `ChessPiece`, этот экземпляр не будет иметь какой-либо смысл в терминах этого перечисления и наверняка станет источником ошибок. Когда вы получаете экземпляр перечисления из внешнего источника, обязательно проверьте его с помощью метода `Enum.IsDefined`:

```
> System.Enum.IsDefined(typeof<ChessPiece>, int ChessPiece.Bishop);;
val it : bool = true
> System.Enum.IsDefined(typeof<ChessPiece>, -711);;
val it : bool = false
```

Во-вторых, перечисления могут содержать только свое значение, тогда как каждый элемент данных размеченного объединения может хранить уникальный кортеж с данными.



Когда использовать перечисления, а когда размеченные объединения

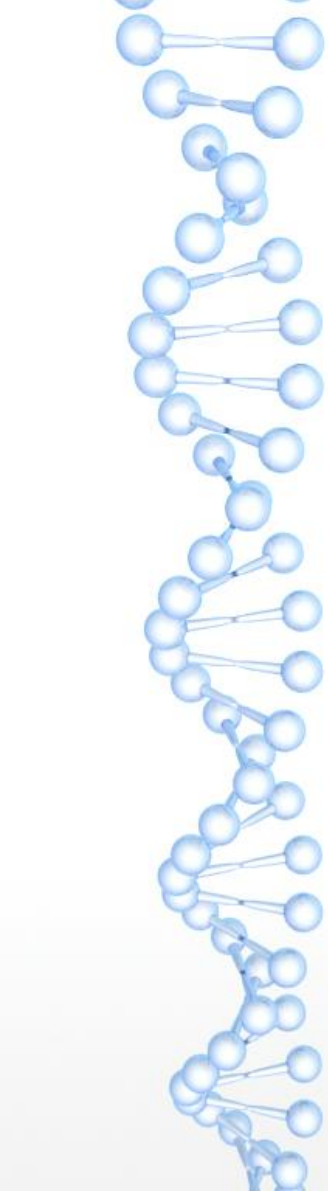
С другой стороны, перечисления представляют общую идиому .NET и обеспечивают более высокую производительность по сравнению с размеченными объединениями. Экземпляр перечисления – это всего лишь несколько байтов в стеке, тогда как размеченное объединение – это ссылочный тип, и для размещения его экземпляров требуется выделение отдельного блока памяти. Поэтому заполнение массива большим количеством перечислений будет выполняться намного быстрее, чем заполнение того же массива размеченными объединениями.

Одной из удобных особенностей перечислений является возможность определять битовые флаги. Взгляните на пример 6.8. Определив значения перечисления в виде степеней двойки, вы можете интерпретировать перечисление как набор битовых флагов. Затем вы можете использовать оператор битового «ИЛИ», `|||`, для объединения значений и использовать маски для проверки установленных битов. Используя этот прием, можно определить до 32 флагов для перечислений на основе `int` и до 64 флагов для перечислений на основе `int64`.

Когда использовать перечисления, а когда размеченные объединения

```
open System

// Перечисление с флаговыми значениями
type FlagsEnum =
    | OptionA = 0b0001
    | OptionB = 0b0010
    | OptionC = 0b0100
    | OptionD = 0b1000

let isFlagSet (enum : FlagsEnum) (flag : FlagsEnum) =
    let flagName = Enum.GetName(typeof<FlagsEnum>, flag)
    if enum &&& flag = flag then
        printfn "Flag [%s] is set." flagName
    else
        printfn "Flag [%s] is not set." flagName
```

```
> // Проверка флага
let customFlags = FlagsEnum.OptionA ||| FlagsEnum.OptionC

isFlagSet customFlags FlagsEnum.OptionA
isFlagSet customFlags FlagsEnum.OptionB
isFlagSet customFlags FlagsEnum.OptionC
isFlagSet customFlags FlagsEnum.OptionD
;;
Flag [OptionA] is set.
Flag [OptionB] is not set.
Flag [OptionC] is set.
Flag [OptionD] is not set.

val customFlags : FlagsEnum = 5
```

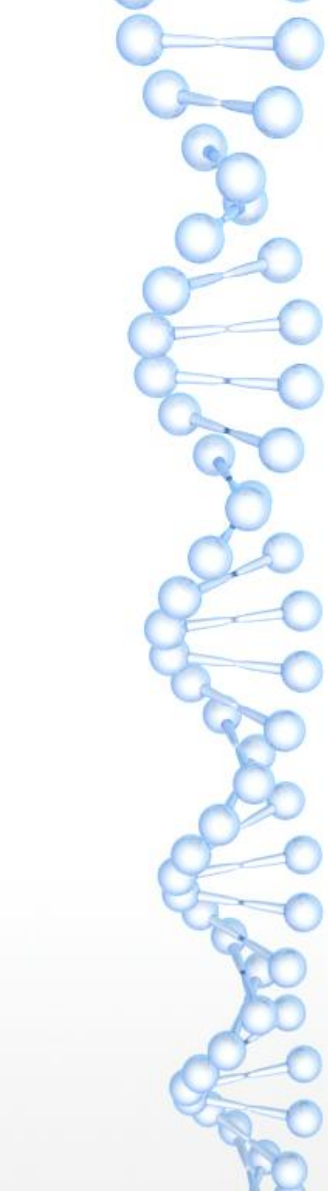


Структуры

Чтобы создать структуру, необходимо определить тип, как это обычно делается, и добавить атрибут [<Struct>] или вставить ключевые слова struct и end до и после тела структуры:

```
[<Struct>]
type StructPoint(x : int, y : int) =
    member this.X = x
    member this.Y = y

type StructRect(top : int, bottom : int, left : int, right : int) =
    struct
        member this.Top = top
        member this.Bottom = bottom
        member this.Left = left
        member this.Right = right

        override this.ToString() =
            sprintf "[%d, %d, %d, %d]" top bottom left right
    end
```

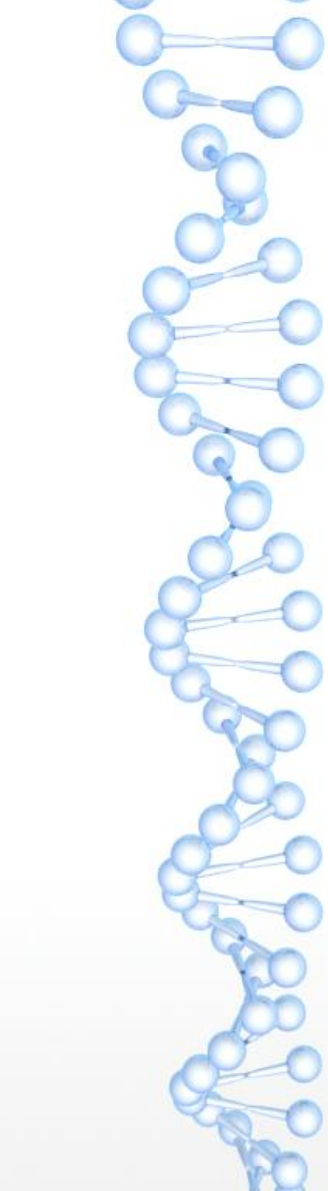
Структуры обладают многими особенностями классов, но при этом могут храниться в стеке, а не в куче. Благодаря этому операция проверки эквивалентности (по умолчанию) выполняется в виде сравнения значений на стеке, а не в виде сравнения ссылок:



Структуры

Одно из различий между структурами и классами заключается в способе их создания. Классы содержат только те конструкторы, которые вы определите вручную, тогда как структуры автоматически получают *конструктор по умолчанию*, который присваивает значения по умолчанию каждому полю структуры. (Если вы помните, экземпляры значимых типов по умолчанию инициализируются нулями, а экземпляры ссылочных типов – значениями `null`.)

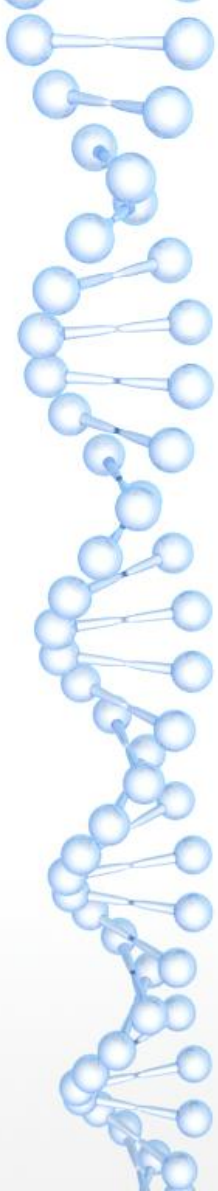
```
> // Структура с описанием книги
[<Struct>]
type BookStruct(title : string, pages : int) =
    member this.Title = title
    member this.Pages = pages

    override this.ToString() =
        sprintf "Title: %A, Pages: %d" this.Title this.Pages;;

type BookStruct =
  struct
    new : title:string * pages:int -> BookStruct
    override ToString : unit -> string
    member Pages : int
    member Title : string
  end

> // Создать экземпляр структуры с помощью конструктора
let book1 = new BookStruct("Philosopher's Stone", 309);;

val book1 : BookStruct = Title: "Philosopher's Stone", Pages: 309

> // Создать экземпляр с помощью конструктора по умолчанию
let namelessBook = new BookStruct();;

val namelessBook : BookStruct = Title: <null>, Pages: 0
```



Структуры

```
> // Определение структуры с изменяемыми полями
[<Struct>]
type MPoint =
    val mutable X : int

    val mutable Y : int

    override this.ToString() =
        sprintf "{%d, %d}" this.X this.Y;;

type MPoint =
  struct
    val mutable X: int
    val mutable Y: int
    override ToString : unit -> string
  end

> let mutable pt = new MPoint();;

val mutable pt : MPoint = {0, 0}
> // Изменение полей
pt.X <- 10
pt.Y <- 7;;
> pt;;
val it : MPoint = {10, 7}
> let nonMutableStruct = new MPoint();;

val nonMutableStruct : MPoint = {0, 0}

> // ОШИБКА: Попытка изменить поле неизменяемого экземпляра структуры
nonMutableStruct.X <- 10;;

  nonMutableStruct.X <- 10;;
  ^^^^^^^^^^^^^^^^^

stdin(54,1): error FS0256: A value must be mutable in order to mutate the
contents of a value type, e.g. 'let mutable x = ...'
```

(Чтобы изменить содержимое или получить адрес типа значения, значение должно быть изменяемым, например, "let mutable x = ...")

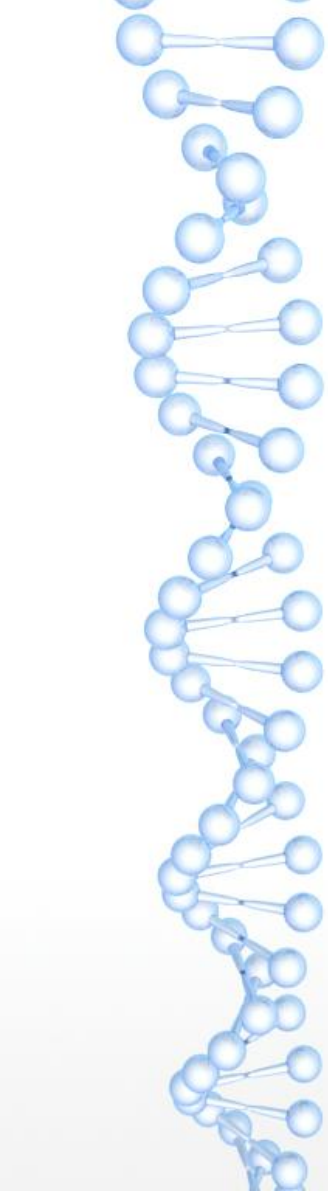


Структуры

Ограничения

Структуры имеют несколько ограничений, связанных с их характеристиками:

- Структуры не могут наследоваться – они неявно помечены атрибутом `[<Sealed>]`.
- Структуры не могут переопределять конструктор по умолчанию (потому что он существует всегда и всегда обнуляет память, занимаемую экземпляром структуры).
- Структуры не могут использовать операторы связывания `let` в определениях полей, как это делается в неявных конструкторах классов.



Структуры

Язык F# предлагает два легковесных контейнера данных – структуры и записи, поэтому иногда могут возникать затруднения в выборе того или иного типа варианта.

Главное преимущество структур перед записями такое же, что и перед классами, а именно – иные показатели производительности. При работе с большим количеством небольших объектов выделение памяти в стеке для создания новых структур выполняется намного быстрее, чем выделение памяти в куче для того же количества объектов. Кроме того, экземпляры структур не нуждаются в сборке мусора, потому что стек очищается автоматически при завершении работы функции.

Однако нередко выигрыш в производительности при использовании структур оказывается незначительным. Фактически необдуманное использование структур может даже приводить к снижению производительности из-за необходимости копировать большие объемы при передаче их функциям в виде параметров. Кроме того, сборщик мусора и менеджер памяти в .NET оптимизированы для использования в высокопроизводительных приложениях. Если вам кажется, что у вас есть проблемы с производительностью, воспользуйтесь инструментами профилирования кода, имеющимися в составе Visual Studio, чтобы выявить узкие места, прежде чем преобразовывать все свои классы и записи в структуры.