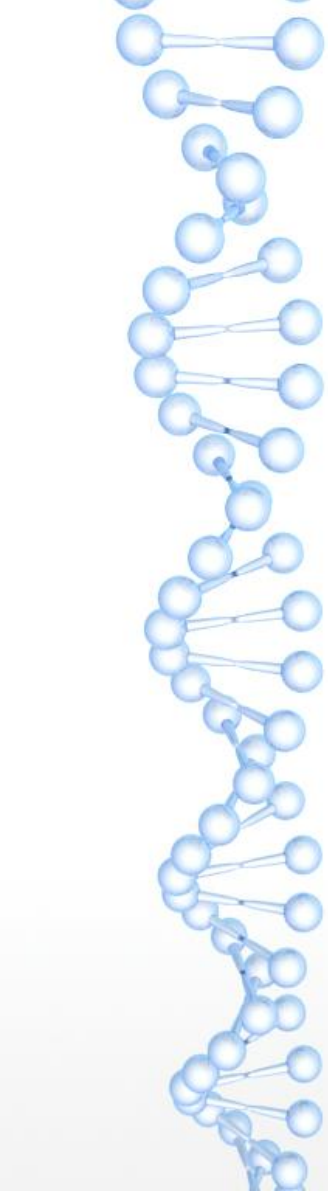
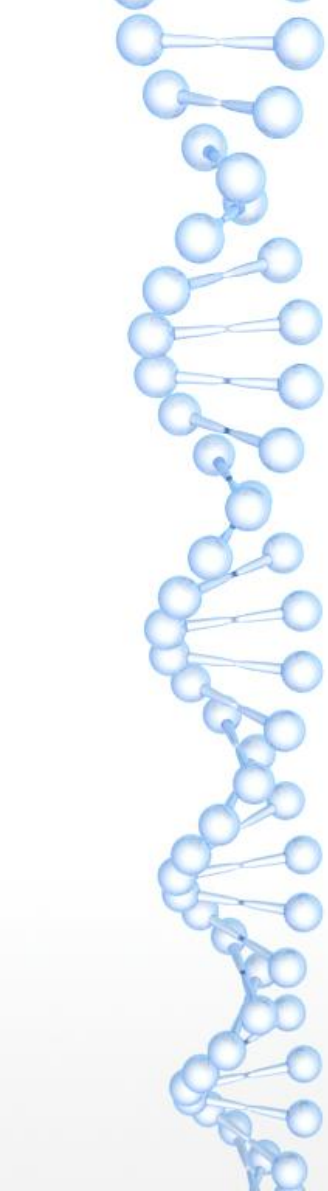


Функциональное и логическое программирование

Лекция 8. Объектно-ориентированное программирование

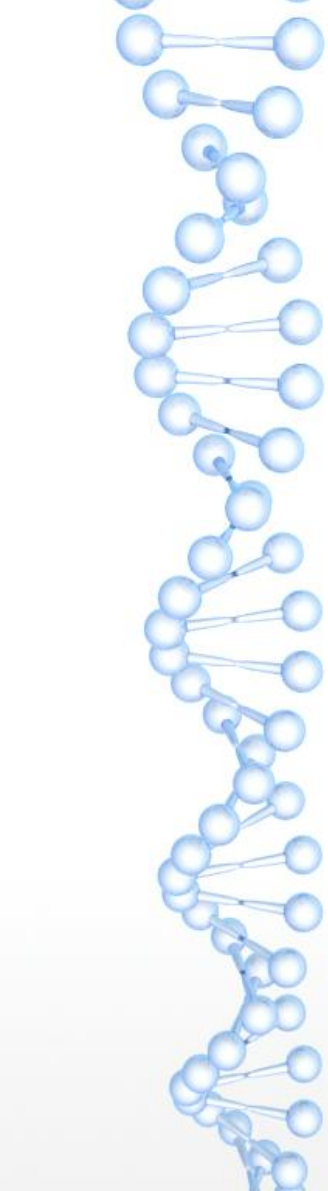


System.Object

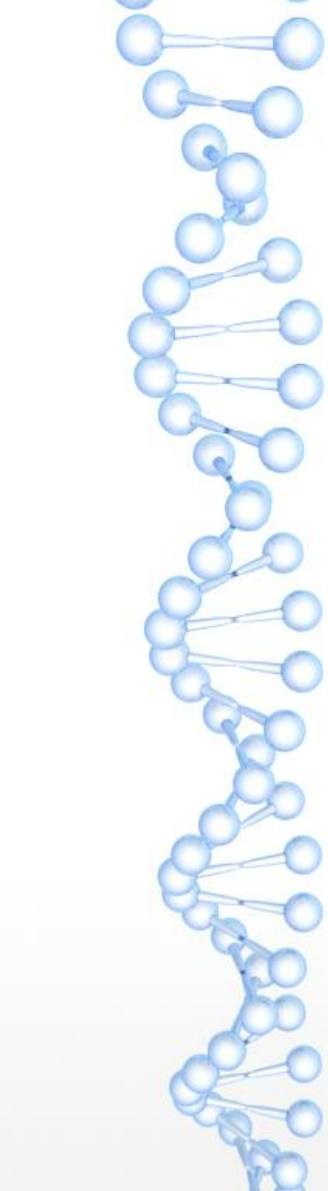
Платформа .NET предлагает богатую систему типов, которая способна проверять идентичность объектов и гарантирует безопасность типов во время выполнения. Благодаря наличию информации о типах, среда времени выполнения платформы .NET не позволит заткнуть круглое отверстие квадратной пробкой.

Идентичность объектов обеспечивается тем, что все компоненты, начиная от целых чисел и заканчивая строками и размеченными объединениями, являются экземплярами типа `System.Object`, который в языке F# обозначается как `obj`. Экземпляры `System.Object` не имеют самостоятельной ценности, потому что они не обладают никакими дополнительными особенностями. Тем не менее важно знать, какие методы есть у типа `System.Object`, потому что они доступны в любых объектах, с которыми вы можете столкнуться в .NET.

System.Object



```
> // Переопределение метода ToString
type PunctuationMark =
    | Period
    | Comma
    | QuestionMark
    | ExclamationPoint
    override this.ToString() =
        match this with
        | Period -> "Period (.)"
        | Comma -> "Comma (,)"
        | QuestionMark -> "QuestionMark (?)"
        | ExclamationPoint -> "ExclamationPoint (!)";;

type PunctuationMark =
    | Period
    | Comma
    | QuestionMark
    | ExclamationPoint
    with
        override ToString : unit -> string
    end

> let x = Comma;;

val x : PunctuationMark = Comma

> x.ToString();;
val it : string = "Comma (,)"
```



System.Object

Метод GetHashCode возвращает хеш-значение объекта. Переопределение этого метода имеет большое значение для типов, которые будут использоваться в таких коллекциях, как Dictionary, HashSet и Set. Хеш-значение – это псевдоуникальный идентификатор, описывающий определенный экземпляр типа. Это делает операцию определения идентичности двух объектов намного более эффективной.

```
> “alpha”.GetHashCode();;
val it : int = -1898387216
> “bravo”.GetHashCode();;
val it : int = 1946920786
> “bravo”.GetHashCode();;
val it : int = 1946920786
```

Реализация по умолчанию вполне пригодна для большинства приложений, но если вы переопределяете метод Equals, вы должны также переопределить метод GetHashCode.

System.Object

Переопределяя метод `GetHashCode`, вы должны гарантировать соблюдение следующих условий, чтобы обеспечить корректное поведение экземпляров данного типа в коллекциях:

- Если два объекта равны, то для них должны возвращаться одинаковые хеш-значения. Соблюдение обратного условия не требуется – два различных объекта могут иметь одинаковые хеш-значения. Однако вы должны приложить все усилия, чтобы избежать подобных коллизий хеш-значений.
- Хеш-значение объекта должно оставаться постоянным, если этот объект никак не изменялся. Если внутреннее состояние объекта изменилось, хеш-значение также может измениться.

System.Object

Equals

Метод Equals проверяет эквивалентность двух объектов. Понятие эквивалентности в .NET – достаточно сложная тема (обращайтесь к разделу «Эквивалентность объектов» ниже):

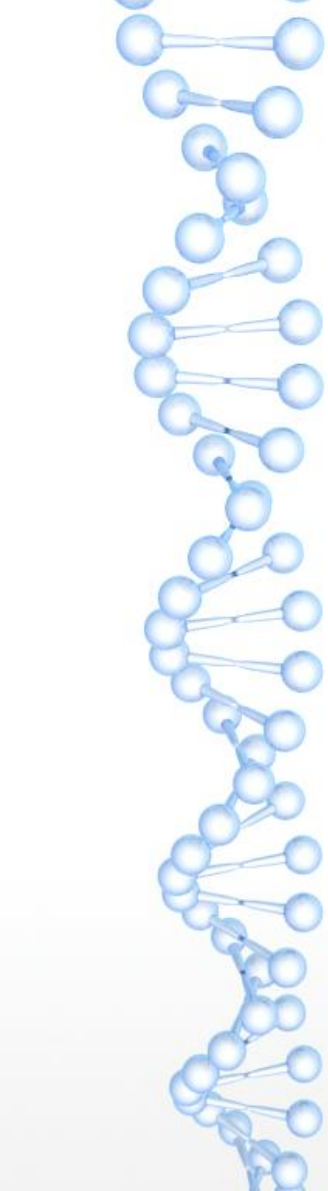
```
> “alpha”.Equals(4);;
val it : bool = false
> “alpha”.Equals(“alpha”);;
val it : bool = true
```

При переопределении метода Equals вы также должны переопределить метод GetHashCode, потому что два эквивалентных объекта должны иметь одинаковые хеш-значения.

GetType

Наконец, метод GetType возвращает экземпляр типа System.Type, который представляет тип фактического объекта. Этот метод имеет большое значение для определения типов во время выполнения, и мы будем говорить об этом гораздо подробнее, когда будем рассматривать механизм рефлексии в главе 12.

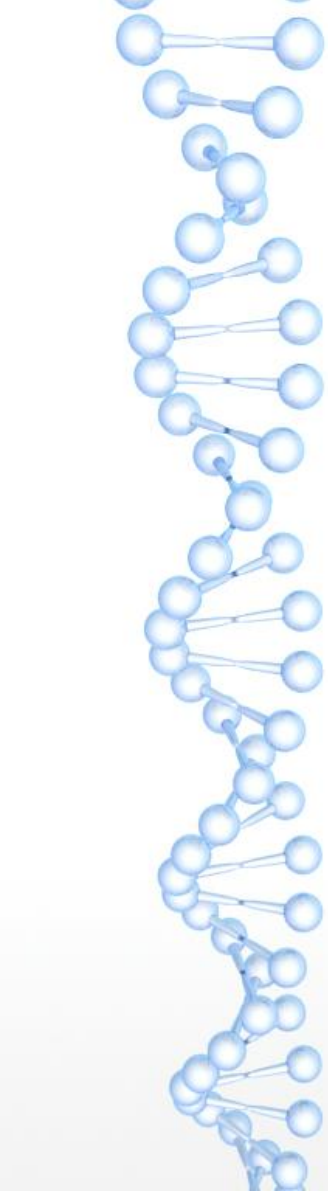
```
> let stringType = "A String".GetType();;

val stringType : System.Type

> stringType.AssemblyQualifiedName;;
val it : string
= "System.String, mscorlib, Version=4.0.0.0, Culture=neutral, Public
KeyToken=b77a5c561934e089"
```



System.Object

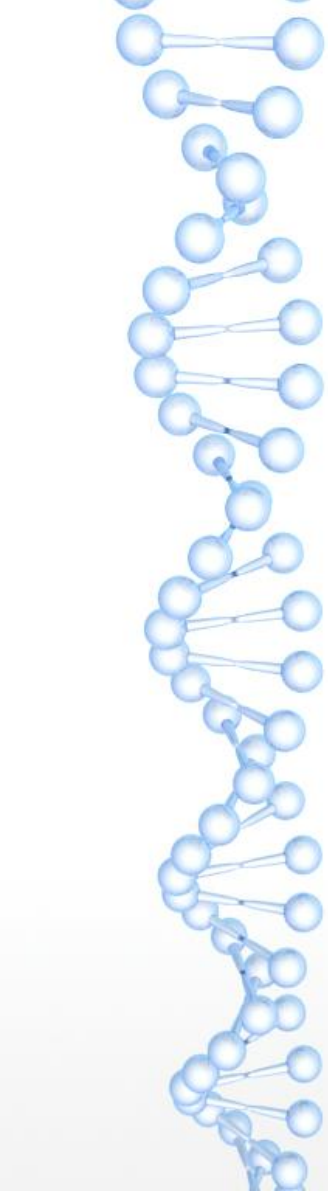
Пример 5.1. Ссылочная эквивалентность

```
> // Ссылочная эквивалентность
type ClassType(x : int) =
    member this.Value = x

let x = new ClassType(42)
let y = new ClassType(42);;

type ClassType =
  class
    new : x:int -> ClassType
    member Value : int
  end
val x : ClassType
val y : ClassType

> x = y;;
val it : bool = false
> x = x;;
val it : bool = true
```



System.Object

```
> // Переопределение метода Equals
type ClassType2(x : int) =
    member this.Value = x
    override this.Equals(o : obj) =
        match o with
        | :? ClassType2 as other -> (other.Value = this.Value)
        | _ -> false
    override this.GetHashCode() = x

let x = new ClassType2(31)
let y = new ClassType2(31)
let z = new ClassType2(10000);;

type ClassType2 =
  class
    new : x:int -> ClassType2
    override Equals : o:obj -> bool
    override GetHashCode : unit -> int
    member Value : int
  end
val x : ClassType2
val y : ClassType2
val z : ClassType2

> x = y;;
val it : bool = true
> x = z;;
val it : bool = false
```



System.Object

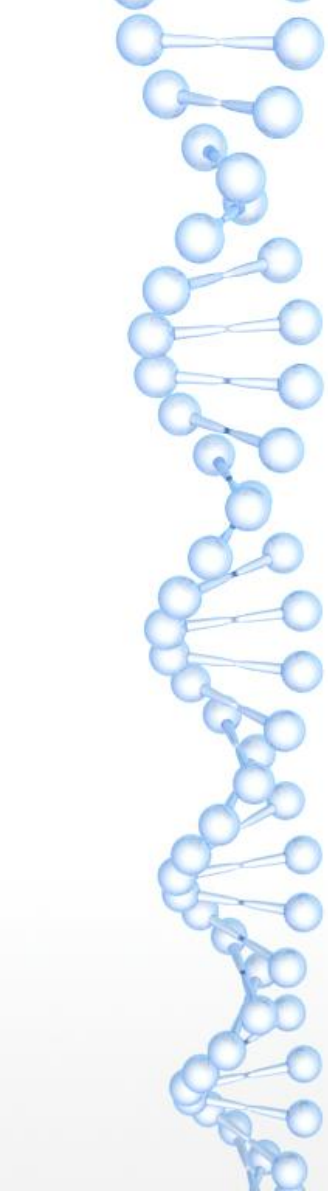
Кортежи считаются эквивалентными, если оба кортежа содержат одинаковое количество элементов и все их элементы эквивалентны:

```
> let x = (1, 'a', "str");;

val x : int * char * string = (1, 'a', "str")

> x = x;;
val it : bool = true
> x = (1, 'a', "different str");;


val it : bool = false
> // Вложенные кортежи
(x, x) = (x, (1, 'a', "str"));;
val it : bool = true
```

System.Object

Записи считаются эквивалентными, если все их поля имеют одинаковые значения:

```
> // Эквивалентность записей
type RecType = { Field1 : string; Field2 : float }

let x = { Field1 = "abc"; Field2 = 3.5 }
let y = { Field1 = "abc"; Field2 = 3.5 }
let z = { Field1 = "XXX"; Field2 = 0.0 };;

type RecType =
  {Field1: string;
   Field2: float;}
val x : RecType = {Field1 = "abc";
                   Field2 = 3.5;}
val y : RecType = {Field1 = "abc";
                   Field2 = 3.5;}
val z : RecType = {Field1 = "XXX";
                   Field2 = 0.0;}

> x = y;;
val it : bool = true
> x = z;;
val it : bool = false
```



System.Object

Размеченные объединения считаются эквивалентными, если оба значения относятся к одному и тому же варианту и кортежи, ассоциированные с вариантами, являются эквивалентными:

```
> // Эквивалентность размеченных объединений
type DUType =
    | A of int * char
    | B

let x = A(1, 'k')
let y = A(1, 'k')
let z = B;;

type DUType =
  | A of int * char
  | B
val x : DUType = A (1,'k')
val y : DUType = A (1,'k')
val z : DUType = B

> x = y;;
val it : bool = true
```

System.Object

Пример 5.2. Переопределение сгенерированной эквивалентности

```
> // Ссылочная эквивалентность в функциональных типах
[<ReferenceEquality>]
type RefDUType =
    | A of int * char
    | B;;

type RefDUType =
  | A of int * char
  | B

> // Объявление двух, концептуально эквивалентных значений
let x = A(4, 'A')
let y = A(4, 'A');;

val x : RefDUType = A (4,'A')
val y : RefDUType = A (4,'A')

> x = y;;
val it : bool = false
> x = x;;
val it : bool = true
```

Классы

Пример 5.3. Явные конструкторы класса

```
type Point =
    val m_x : float
    val m_y : float

// Конструктор 1 - принимает два параметра
new (x, y) = { m_x = x; m_y = y }

// Конструктор 2 - не имеет параметров
new () = { m_x = 0.0; m_y = 0.0 }

member this.Length =
    let sqr x = x * x
    sqrt <| sqr this.m_x + sqr this.m_y

let p1 = new Point(1.0, 1.0)
let p2 = new Point()
```

Классы

Пример 5.4. Выполнение произвольного кода перед вызовом явного конструктора

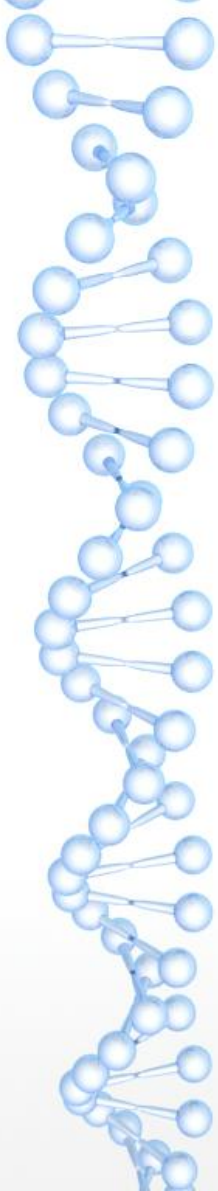
```
open System

type Point2 =
    val m_x : float
    val m_y : float

    // Анализ строки, имеющей вид "1.0, 2.0"
    new (text : string) as this =
        // Выполнить предобработку
        if text = null then
            raise <| new ArgumentException("text")

        let parts = text.Split([| ',' |])
        let (successX, x) = Double.TryParse(parts.[0])
        let (successY, y) = Double.TryParse(parts.[1])

        if not successX || not successY then
            raise <| new ArgumentException("text")

        // Инициализация полей класса
        { m_x = x; m_y = y }
        then
            // Выполнить постобработку
            printfn
                "Initialized to [%f, %f]"
                this.m_x
                this.m_y
```

Классы

Чтобы создать класс с неявным конструктором, достаточно просто добавить круглые скобки с аргументами после имени класса. Эти аргументы будут играть роль параметров *основного конструктора (primary constructor)* класса. Любые операторы связывания `let` и `do`, присутствующие в теле класса, будут выполняться при создании его экземпляров. Любые дополнительные конструкторы должны вызывать основной конструктор, чтобы обеспечить выполнение всех операторов `let`.

Пример 5.5. Неявный конструктор класса

```
type Point3(x : float, y : float) =

    let length =
        let sqr x = x * x
        sqrt <| sqr x + sqr y
    do printfn "Initialized to [%f, %f]" x y

    member this.X = x
    member this.Y = y
    member this.Length = length

    // Определяем дополнительные конструкторы, которые должны
```

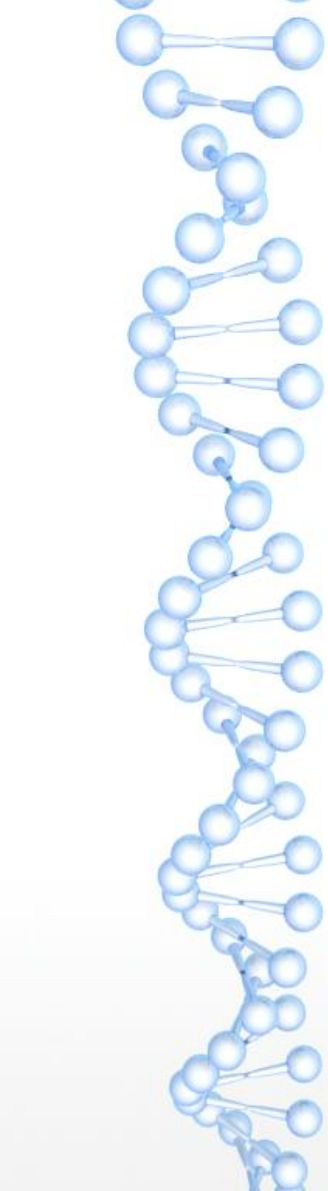


Классы

```
// Определяем дополнительные конструкторы, которые должны
// вызывать ‘основной’ конструктор
new() = new Point3(0.0, 0.0)

// Второй конструктор.
new(text : string) =
    if text = null then
        raise <| new ArgumentException(“text”)


let parts = text.Split([| ‘,’ |])
let (successX, x) = Double.TryParse(parts.[0])
let (successY, y) = Double.TryParse(parts.[1])
if not successX || not successY then
    raise <| new ArgumentException(“text”)
// Вызов основного конструктора класса
new Point3(x, y)
```

Обобщенные классы

```
> // Определение обобщенного класса
type Arrayify<'a>(x : 'a) =

  member this.EmptyArray : 'a[] = [| |]
  member this.ArraySize1 : 'a[] = [| x |]
  member this.ArraySize2 : 'a[] = [| x; x |]
  member this.ArraySize3 : 'a[] = [| x; x; x |];;

type Arrayify<'a> =
  class
    new : x:'a -> Arrayify<'a>
    member ArraySize1 : 'a []
    member ArraySize2 : 'a []
    member ArraySize3 : 'a []
    member EmptyArray : 'a []
  end

> let arrayifyTuple = new Arrayify<int * int>( (10, 27) );;

val arrayifyTuple : Arrayify<int * int>

> arrayfiyTuple.ArraySize3;;
val it : (int * int) [] = [|(10, 27); (10, 27); (10, 27)|]
```



Обобщенные классы

Важно отметить, что в вызове конструктора обобщенный параметр можно опустить, заменив его групповым символом <_>, и положиться на автоматический вывод типа параметра компилятором F#. В следующем фрагменте на основе значения параметра x, передаваемого конструктору класса Arrayify, компилятор определил, что обобщенный параметр относится к типу string:

```
> let inferred = new Arrayify<_>( "a string" );;

val inferred : Arrayify<string>
```

Записи и размеченные объединения также могут быть обобщенными. В следующем фрагменте приводится определение обобщенного размеченного объединения:

```
> // Обобщенное размеченное объединение
type GenDU<'a> =
    | Tag1 of 'a
    | Tag2 of string * 'a list;;

type GenDU<'a> =
  | Tag1 of 'a
  | Tag2 of string * 'a list

> Tag2("Primary Colors", [ 'R'; 'G'; 'B' ]);;
val it : GenDU<char> = Tag2 ("Primary Colors",['R'; 'G'; 'B'])
```

Пример 5.7. Именование указателя this

```
open System

type Circle =
    val m_radius : float

    new(r) = { m_radius = r }
    member foo.Radius = foo.m_radius
    member bar.Area = Math.PI * bar.Radius * bar.Radius
```

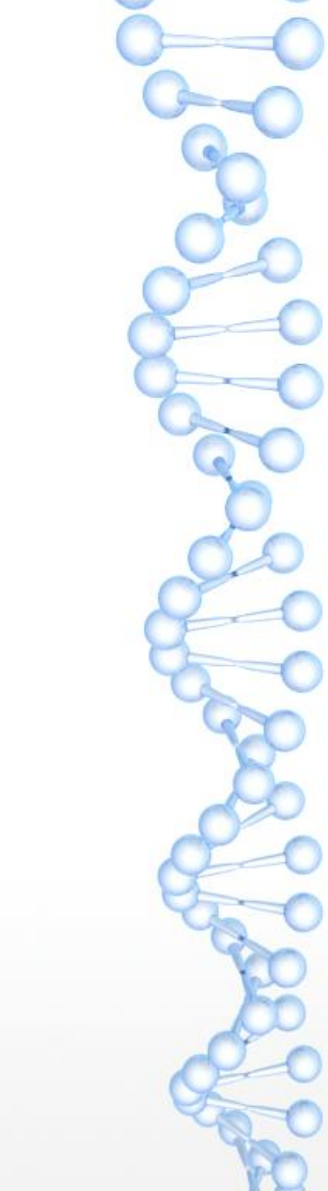
Безусловно, имя собственного идентификатора не должно конфликтовать с именами членов класса.

В примерах, которые приводятся в этой книге, обычно используется имя this из соображений соответствия синтаксису языка C#; при этом в типичном коде на языке F# редко возникает необходимость использования собственного идентификатора, поэтому ему обычно дается более короткое имя, такое как x или даже __.

Методы и свойства

Пример 5.8. Определение свойств класса

```
> // Определение класса WaterBottle с двумя свойствами
[<Measure>]
type ml

type WaterBottle() =
    let mutable m_amount = 0.0<ml>

    // Свойство, доступное только для чтения
    member this.Empty = (m_amount = 0.0<ml>)

    // Свойство, доступное для чтения и записи
    member this.Amount with get ()    = m_amount
                       and set newAmt = m_amount <- newAmt;;

[<Measure>]
type ml
type WaterBottle =
  class
    new : unit -> WaterBottle
    member Amount : float<ml>
    member Empty : bool
    member Amount : float<ml> with set
  end

> let bottle = new WaterBottle();;

val bottle : WaterBottle

> bottle.Empty;;
val it : bool = true
> bottle.Amount <- 1000.0<ml>;;
val it : unit = ()

> bottle.Empty;;
val it : bool = false
```



Методы и свойства

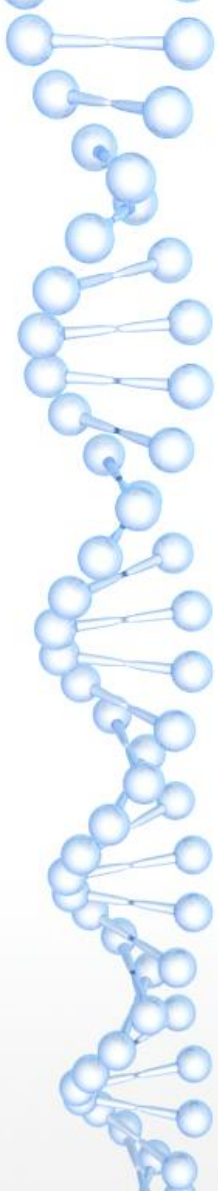
Пример 5.9. Установка значений свойств после вызова конструктора

```
open System.Windows.Forms

// Попытка первая - сложный способ
let f1 = new Form()
f1.Text    <- "Window Title"
f1.TopMost <- true
f1.Width   <- 640
f1.Height  <- 480
f1.ShowDialog()

// Попытка вторая - простой способ
let f2 = new Form(Text    = "Window Title",
                  TopMost = true,
                  Width   = 640,
                  Height  = 480)
f2.ShowDialog()
```

Методы и свойства

```
type Television =

    val mutable m_channel : int
    val mutable m_turnedOn : bool

    new() = { m_channel = 3; m_turnedOn = true }

    member this.TurnOn () =
        printfn "Turning on..."
        this.m_turnedOn <- true

    member this.TurnOff () =
        printfn "Turning off..."
        this.m_turnedOn <- false

    member this.ChangeChannel (newChannel : int) =
        if this.m_turnedOn = false then
            failwith "Cannot change channel, the TV is not on."

        printfn "Changing channel to %d..." newChannel
        this.m_channel <- newChannel

    member this.CurrentChannel = this.m_channel
```

Методы и свойства



```
> // Каррируемые методы класса
type Adder() =
    // Каррируемые аргументы метода
    member this.AddTwoParams x y = x + y
    // Обычные аргументы
    member this.AddTwoTupledParams (x, y) = x + y;;

type Adder =
  class
    new : unit -> Adder
    member AddTwoParams : x:int -> y:int -> int
    member AddTwoTupledParams : x:int * y:int -> int
  end

> let adder = new Adder();;

val adder : Adder


> let add10 = adder.AddTwoParams 10;;

val add10 : (int -> int)

> adder.AddTwoTupledParams(1, 2);;
val it : int = 3
```

Статические методы, свойства и поля

Пример 5.10. Статические методы

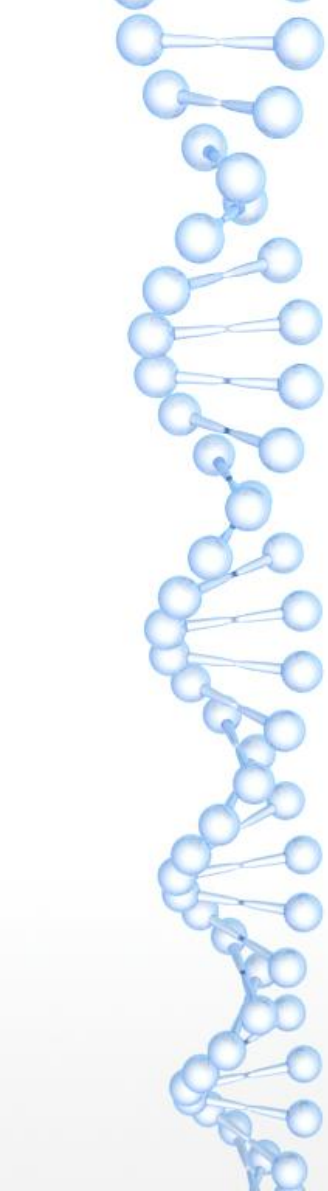
```
> // Объявление статического метода
type SomeClass() =
    static member StaticMember() = 5;;

type SomeClass =
  class
    new : unit -> SomeClass

    static member StaticMember : unit -> int
  end

> SomeClass.StaticMember();;
val it : int = 5
> let x = new SomeClass();;

val x : SomeClass

> x.StaticMember();;

x.StaticMember();;
^^^^^^^^^^^^^^^^^
stdin(39,1): error FS0809: StaticMember is not an instance method
```

(StaticMember не является экземплярным методом)

Статические методы, свойства и поля

Пример 5.11. Создание и использование статических полей

```
// Статические поля
type RareType() =

    // Все экземпляры класса RareType имеют одно общее значение m_numLeft
    static let mutable m_numLeft = 2

    do
        if m_numLeft <= 0 then
            failwith "No more left!"
        m_numLeft <- m_numLeft - 1
        printfn " Инициализация экземпляра RareType. Осталась возможность
                  создания только %d экземпляра(ов)" m_numLeft

    static member NumLeft = m_numLeft
```



Статические методы, свойства и поля

Следующий сеанс FSI показывает действие этого счетчика:

```
> let a = new RareType();;

val a : RareType

Initialized a rare type, only 1 left!
> let b = new RareType();;

val b : RareType

Initialized a rare type, only 0 left!
> let c = new RareType();;

val c : RareType

System.Exception: No more left!
  at FSI_0012.RareType..ctor() in C:\Users\chrsmith\Desktop\Ch05.fsx:line 18
  at <StartupCode$FSI_0015>.$FSI_0015._main()
stopped due to error
> RareType.NumLeft;;
val it : int = 0
```

Перегрузка методов

Пример 5.12. Перегрузка методов в F#

```
type BitCounter =

    static member CountBits (x : int16) =

        let mutable x' = x
        let mutable numBits = 0
        for i = 0 to 15 do
            numBits <- numBits + int (x' &&& 1s)
            x' <- x' >>> 1
        numBits

    static member CountBits (x : int) =
        let mutable x' = x
        let mutable numBits = 0
        for i = 0 to 31 do
            numBits <- numBits + int (x' &&& 1)
            x' <- x' >>> 1
        numBits

    static member CountBits (x : int64) =
        let mutable x' = x
        let mutable numBits = 0
        for i = 0 to 63 do
            numBits <- numBits + int (x' &&& 1L)
            x' <- x' >>> 1
        numBits
```

Модификаторы доступа

Пример 5.13. Модификаторы доступа

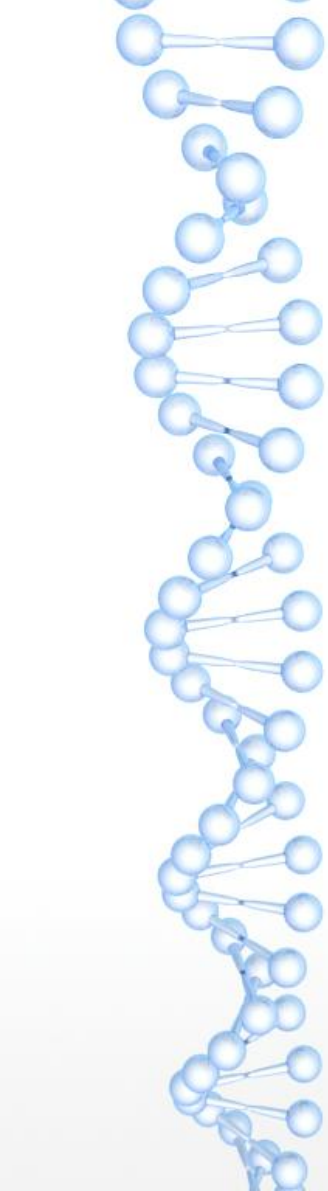
```
type internal Ruby private(shininess, carats) =

    let mutable m_size = carats
    let mutable m shininess = shininess
  // Полировка увеличивает отражающую способность, но уменьшает размер
  member this.Polish() =
      this.Size   <- this.Size - 0.1
      m_shininess <- m_shininess + 0.1

  // Открытый метод чтения и закрытый метод записи
  member public this.Size with get () = m_size
  member private this.Size with set newSize = m_size <- newSize

  member this.Shininess = m_shininess

  public new() =
      let rng = new Random()
      let s = float (rng.Next() % 100) * 0.01
      let c = float (rng.Next() % 16) + 0.1
      new Ruby(s, c)

  public new(carats) =
      let rng = new Random()
      let s = float (rng.Next() % 100) * 0.01
      new Ruby(s, carats)
```

Модификаторы доступа

Таблица 5.1. Модификаторы доступа

Модификатор доступа	Видимость
`public`	Модификатор `public` означает, что метод или свойство будет доступно откуда угодно. Все классы, значения и функции в языке F# по умолчанию являются открытыми.
`private`	Модификатор `private` ограничивает область видимости значения данным классом. Закрытые значения недоступны ни внешнему коду, ни производным классам. Все поля классов по умолчанию являются закрытыми.
`internal`	Модификатор `internal` имеет то же значение, что и модификатор `public`, но он распространяется только на текущую сборку. Внутренние (`internal`) классы недоступны в других сборках, как если бы они были объявлены закрытыми (`private`).

В отличие от языка C#, в F# нет модификатора `protected`. Однако F# будет соблюдать правила видимости защищенных членов при наследовании классов, реализованных на других языках, где поддерживается этот модификатор доступа.



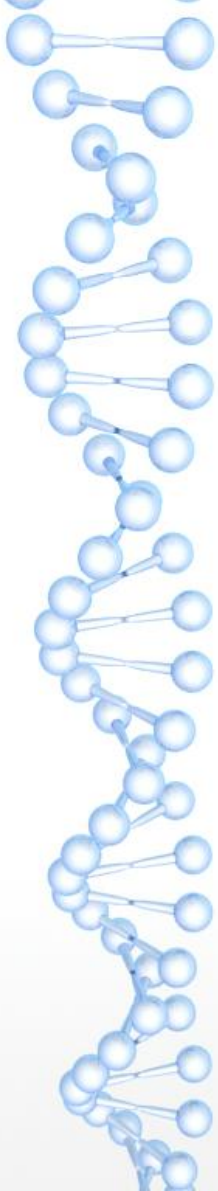
Модификаторы доступа

Модификаторы могут применяться не только к классам – они могут также использоваться для значений, определенных в модулях.

В примере 5.14 определяется модуль `Logger`, в котором имеется закрытое изменяемое значение `m_filesToWriteTo`. Это значение доступно только внутри модуля, поэтому оно может использоваться методами `AddLogFile` и `LogMessage` без дополнительной проверки `m_filesToWriteTo` на равенство значению `null`. (Если бы значение `m_filesToWriteTo` было открытым, вызывающий код мог бы по ошибке изменить это значение.)

Пример 5.14. Модификаторы доступа в модулях

```
open System.IO
open System.Collections.Generic

module Logger =

    let mutable private m_filesToWriteTo = new List<string>()

    let AddLogFile(filePath) = m_filesToWriteTo.Add(filePath)

    let LogMessage(message : string) =
        for logFile in m_filesToWriteTo do
            use file = new StreamWriter(logFile, true)
            file.WriteLine(message)
            file.Close()
```



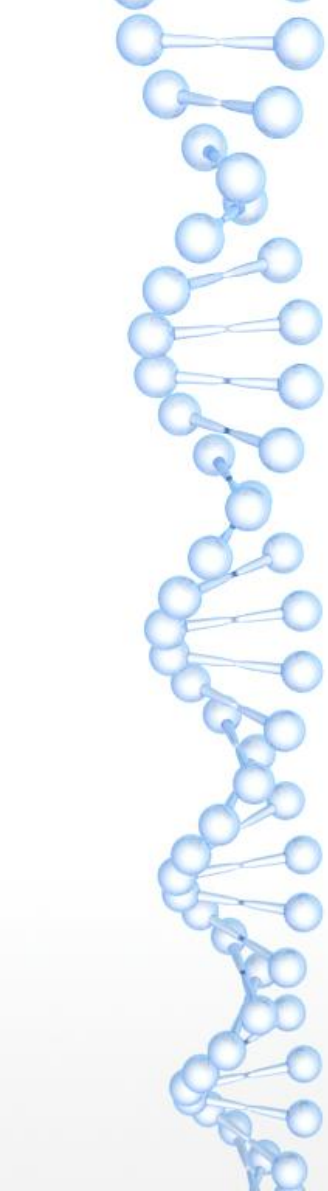
Наследование

```
// Базовый класс
type BaseClass =
    val m_field1 : int

    new(x) = { m_field1 = x }
    member this.Field1 = this.m_field1

// Производный класс с неявным конструктором
type ImplicitDerived(field1, field2) =
    inherit BaseClass(field1)

    let m_field2 : int = field2

    member this.Field2 = m_field2
```

Наследование

В классах с явными конструкторами вызов конструктора базового класса осуществляется путем указания `inherit` *тип* **в разделе инициализации полей класса:**

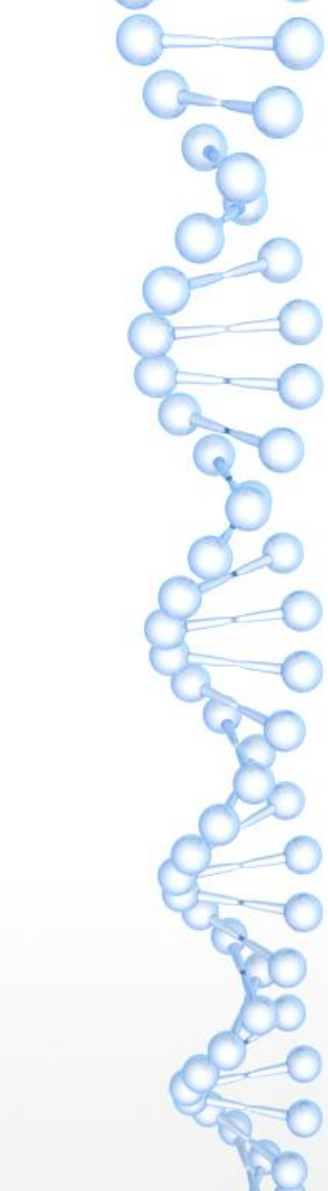
```
// Производный класс с явным конструктором
type ExplicitDerived =
    inherit BaseClass

    val m_field2 : int

    new(field1, field2) =
        {
            inherit BaseClass(field1)
            m_field2 = field2
        }

    member this.Field2 = this.m_field2
```



Наследование

Пример 5.16. Переопределение методов в языке F#

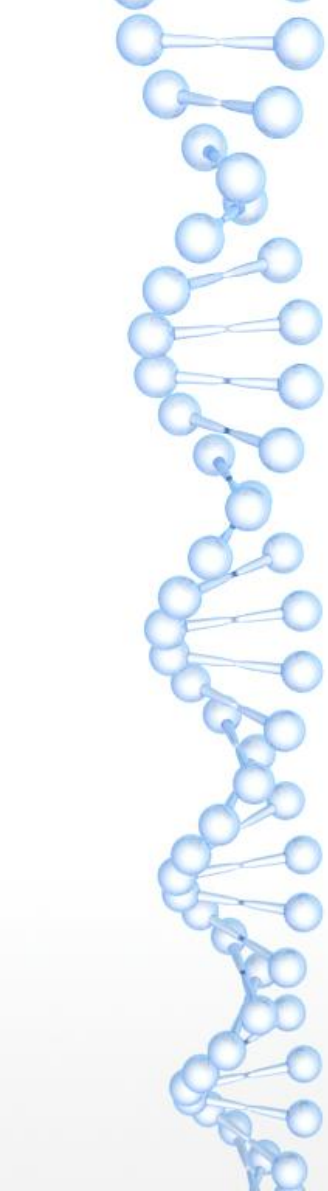
```
type Sandwich() =
    abstract Ingredients : string list
    default this.Ingredients = []

    abstract Calories : int
    default this.Calories = 0
type BLTSandwich() =
    inherit Sandwich()

    override this.Ingredients = ["Bacon"; "Lettuce"; "Tomato"]
    override this.Calories = 330

type TurkeySwissSandwich() =
    inherit Sandwich()

    override this.Ingredients = ["Turkey"; "Swiss"]
    override this.Calories = 330
```



Наследование

Пример 5.17. Обращение к базовому классу

```
> // Сэндвич бекон-салат-помидор (BLT) с добавлением соленого огурца
type BLTWithPickleSandwich() =
    inherit BLTSandwich()

    override this.Ingredients = "Pickles" :: base.Ingredients
    override this.Calories = 50 + base.Calories;;

type BLTWithPickleSandwich =
  class
    inherit BLTSandwich
    new : unit -> BLTWithPickleSandwich
    override Calories : int
    override Ingredients : string list
  end

> let lunch = new BLTWithPickleSandwich();;

val lunch : BLTWithPickleSandwich

> lunch.Ingredients;;
val it : string list = ["Pickles"; "Bacon"; "Lettuce"; "Tomato"]
```



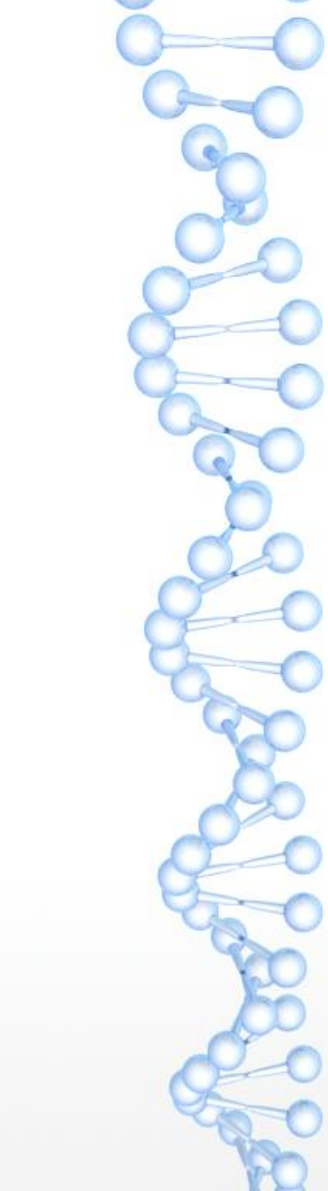
Абстрактные классы

Абстрактные классы (abstract classes) обычно являются корнем иерархии наследования и не являются самодостаточными. Фактически вы не можете создавать экземпляры классов, помеченных как абстрактные, так как в противном случае у вас появилась бы возможность вызвать метод, который объявлен, но не имеет реализации.

Для объявления абстрактного класса достаточно добавить атрибут `[<AbstractClass>]` к объявлению класса. В противном случае при наличии абстрактных методов, не имеющих реализации по умолчанию, компилятор выведет сообщения об ошибке.

```
> // ОШИБКА: Определение класса с членами, не имеющими реализации
type Foo() =
    member this.Alpha() = true
    abstract member Bravo : unit -> bool;;

  type Foo() =
  -----^^^

stdin(2,6): error FS0365: No implementation was given for 'abstract member
Foo.Bravo : unit -> bool'
```

(Для "abstract member Foo.Bravo : unit -> bool" не было дано реализации)



```
> // Правильно объявленный абстрактный класс
[<AbstractClass>]
type Bar() =
    member this.Alpha() = true
    abstract member Bravo : unit -> bool;;

type Bar =
  class
    new : unit -> Bar
    abstract member Bravo : unit -> bool
    member Alpha : unit -> bool
  end
```

Запечатанные классы

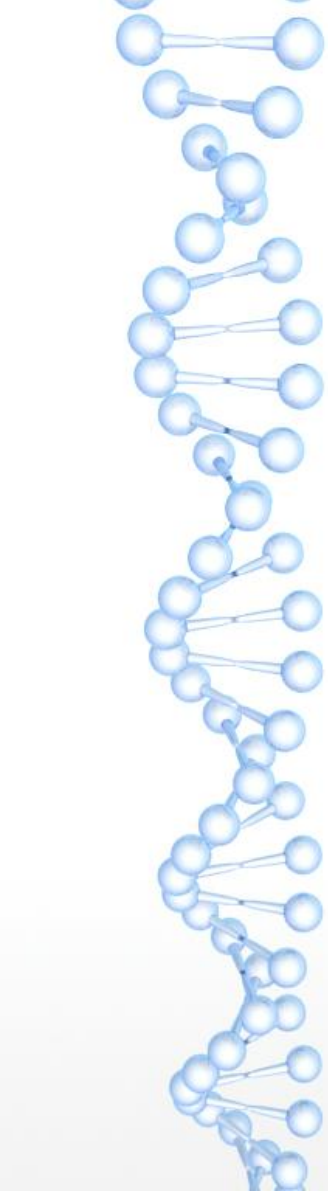
```
> // Определение запечатанного класса
[<Sealed>]
type Foo() =
    member this.Alpha() = true;;

type Foo =
  class
    new : unit -> Foo
    member Alpha : unit -> bool
  end

> // ОШИБКА: Попытка наследовать от запечатанного класс
type Bar() =
    inherit Foo()
    member this.Barvo() = false;;

      inherit Foo()
  ----^^^^^^^^^^^^^^

stdin(19,5): error FS0945: Cannot inherit a sealed type
```

(Не удается реализовать наследование от запечатанного типа)

Приведение типа

Пример 5.18. Статическое приведение типа

```
> // Определение иерархии классов
[<AbstractClass>]
type Animal() =
    abstract member Legs : int

[<AbstractClass>]
type Dog() =
    inherit Animal()
    abstract member Description : string
    override this.Legs = 4

type Pomeranian() =
    inherit Dog()
    override this.Description = "Furry";;

... вырезано ...

> let steve = new Pomeranian();;

val steve : Pomeranian

> // Приведение экземпляра steve к различным типам
let steveAsDog    = steve :> Dog
let steveAsAnimal = steve :> Animal
let steveAsObject = steve :> obj;;

val steveAsDog : Dog
val steveAsAnimal : Animal
val steveAsObject : obj
```



Динамическое приведение типа

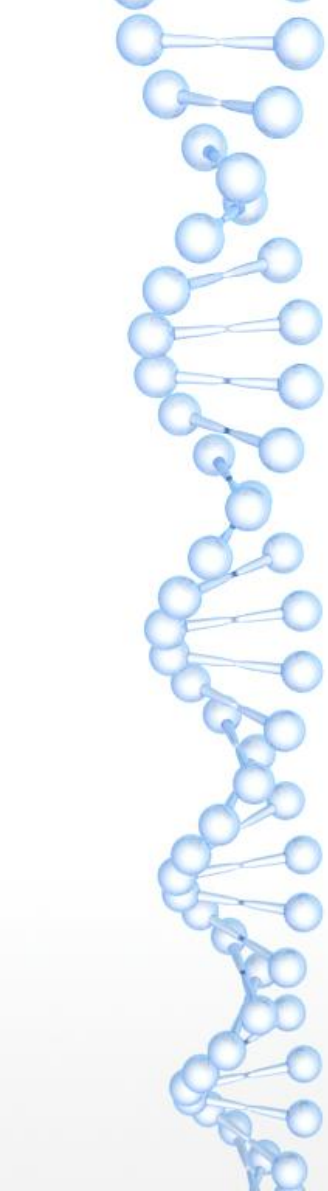
Чаще всего динамическое приведение типа производится, когда имеется экземпляр класса `System.Object`, но при этом известно, что в действительности он является экземпляром некоторого другого, производного класса. Для динамического приведения типа используется оператор динамического приведения типа `:?>`. Продолжая предыдущий пример, мы можем привести значение `steveAsObj` типа `obj` к типу `Dog`, выполнив динамическое приведение типа:

```
> let steveAsObj = steve :> obj;;

val steveAsObj : obj

> let steveAsDog = steveAsObj :?> Dog;;

val steveAsDog : Dog

> steveAsDog.Description;;
val it : string = "Furry"
```

Если экземпляр преобразуется к типу, которым он в действительности не является, во время выполнения мы получим исключение `System.InvalidCastException`:

```
> let _ = steveAsObj :?> string;;
System.InvalidCastException: Unable to cast object of type 'Pomeranian' to type
'System.String'.
  at <StartupCode$FSI_0022>.$FSI_0022._main()
stopped due to error
```

(Не удалось привести тип объекта "Pomeranian" к типу "System.String". Остановлено из-за ошибки.)



Динамическое приведение типа

Пример 5.19. Проверка типа с помощью выражения сопоставления с образцом

```
> // Сопоставление с типами
let whatIs (x : obj) =
    match x with
    | :? string    as s -> printfn "x is a string \"%s\"" s
    | :? int       as i -> printfn "x is an int %d" i
    | :? list<int> as l -> printfn "x is an int list '%A'" l
    | _ -> printfn "x is a '%s'" <| x.GetType().Name;;

val whatIs : obj -> unit

> whatIs [1 .. 5];;
x is an int list '[1; 2; 3; 4; 5]'
val it : unit = ()
> whatIs "Rosebud";;
x is a string "Rosebud"
val it : unit = ()
> whatIs (new System.IO.FileInfo(@"C:\config.sys"));;
x is a 'FileInfo'
val it : unit = ()
```