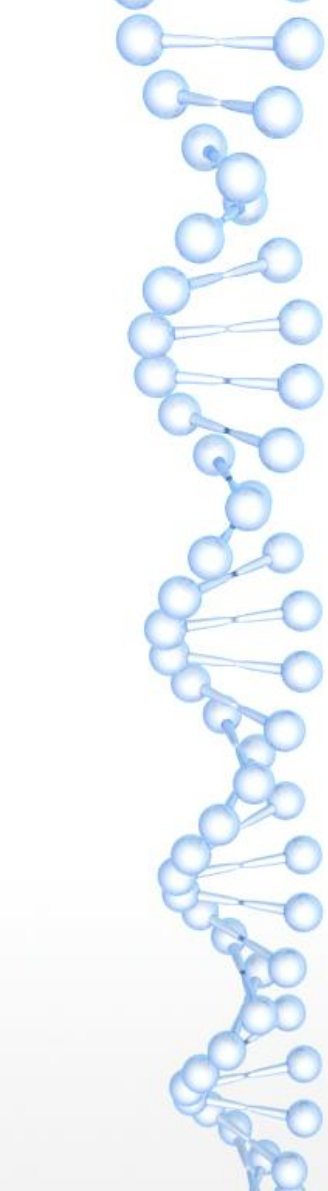
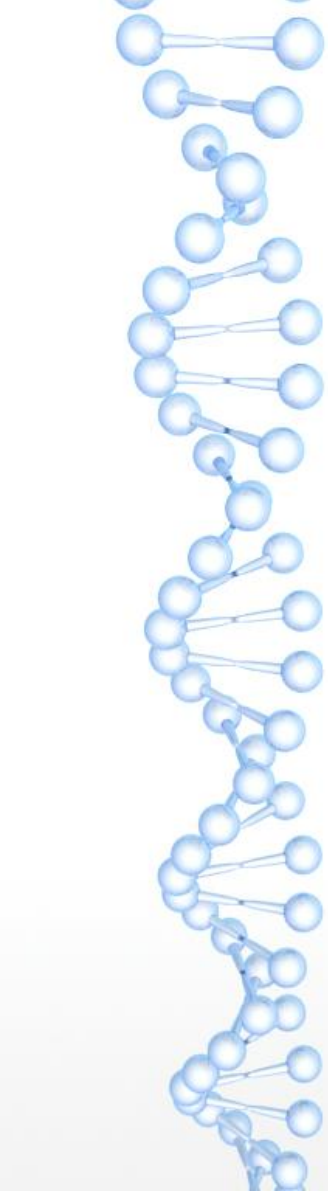


Функциональное и логическое программирование

Лекция 7. Императивное программирование

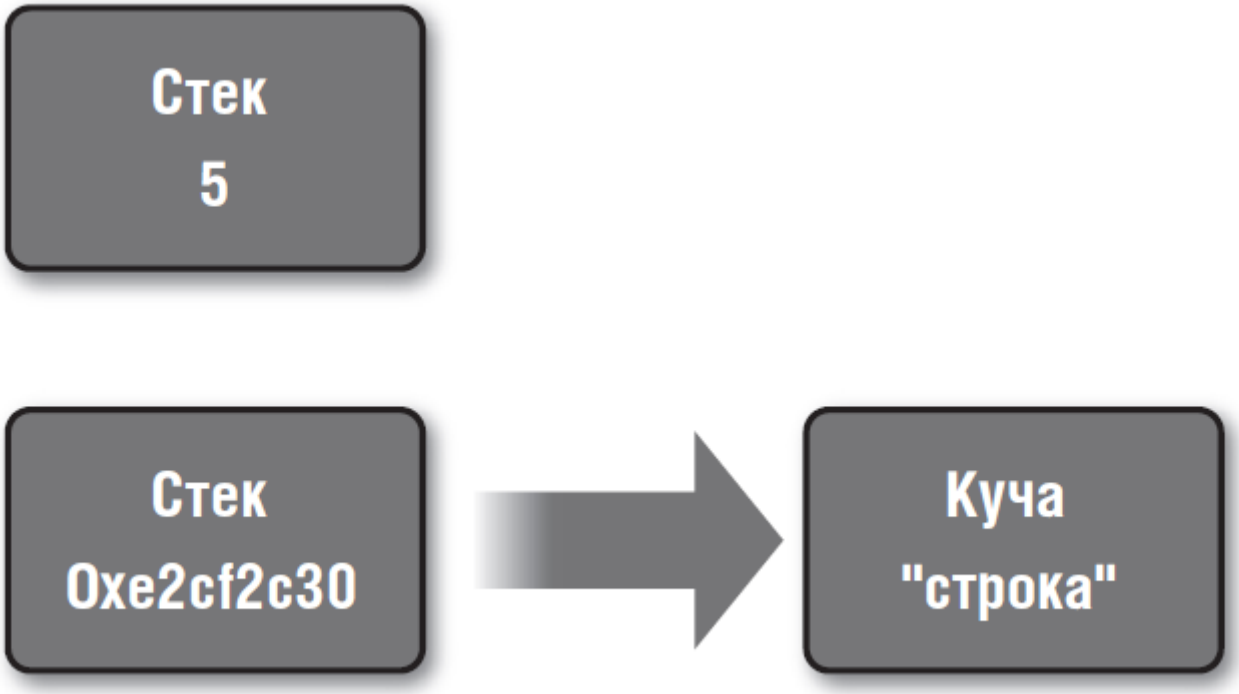


Значимые типы и ссылочные типы

Типы данных, хранящихся в стеке, известны как *значимые типы (value types)*, а типы данных, хранящихся в «куче», известны как *ссылочные типы (reference types)*.

Значимые типы значений имеют фиксированный размер. Примерами значимых типов являются типы `int` и `float`, потому что данные этих типов имеют постоянный размер. С другой стороны, при работе со ссылочными типами в стеке хранятся лишь *указатели* на другие области памяти, где находятся фактические данные. Но несмотря на то, что сам указатель имеет фиксированный размер (обычно четыре или восемь байт), данные, на которые он указывает, могут занимать гораздо больший объем. Примерами ссылочных типов могут служить типы `list` и `string`.

Значимые и ссылочные типы



Значимые и ссылочные типы

Чтобы получить значение по умолчанию для любого типа, можно воспользоваться функцией типа `Unchecked.defaultof<'a>`. Она возвращает значение по умолчанию для указанного типа.

Функция типа (type function) – это особый тип функции, которая не принимает никаких аргументов, кроме параметра обобщенного типа. Существует несколько полезных функций типов, которые мы исследуем в следующих главах:

`Unchecked.defaultof<'a>`

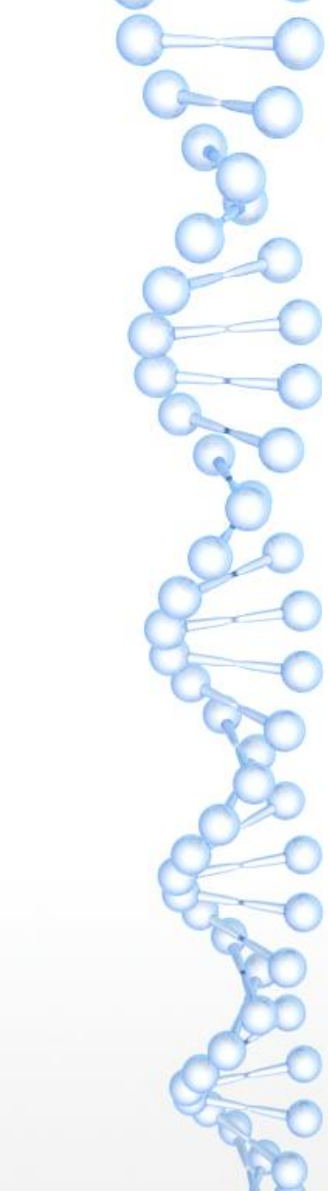
Возвращает значение по умолчанию для типа `'a`.

`typeof<'a>`

Возвращает объект `System.Type`, описывающий тип `'a`.

`sizeof<'a>`

Возвращает размер памяти в стеке, выделяемой для объекта типа `'a`.



Значимые и ссылочные типы

Для значимых типов в качестве значений по умолчанию используется нулевое битовое представление. Поскольку размер экземпляра известен в момент создания, в стеке для него выделяется необходимое количество байтов, каждый байт которого устанавливается в значение `0b00000000`.

Значения по умолчанию для ссылочных типов имеют более сложный вид. До инициализации ссылочный тип указывает на специальный адрес `null`. Он используется, чтобы отличать неинициализированные данные ссылочных типов.

Для проверки равенства ссылочного типа значению `null` в языке F# можно использовать ключевое слово `null`. В следующем примере определяется функция, которая проверяет, равен ли входной аргумент значению `null` или нет, а затем вызывает ее, передавая инициализированную и неинициализированную строки:

Значимые и ссылочные типы

```
> let isNull = function null -> true | _ -> false;;

val isNull : obj -> bool

> isNull "a string";;
val it : bool = false
> isNull (null : string);;
val it : bool = true
```

Однако null в языке F# не является допустимым значением для ссылочных типов, то есть им нельзя присвоить значение null:

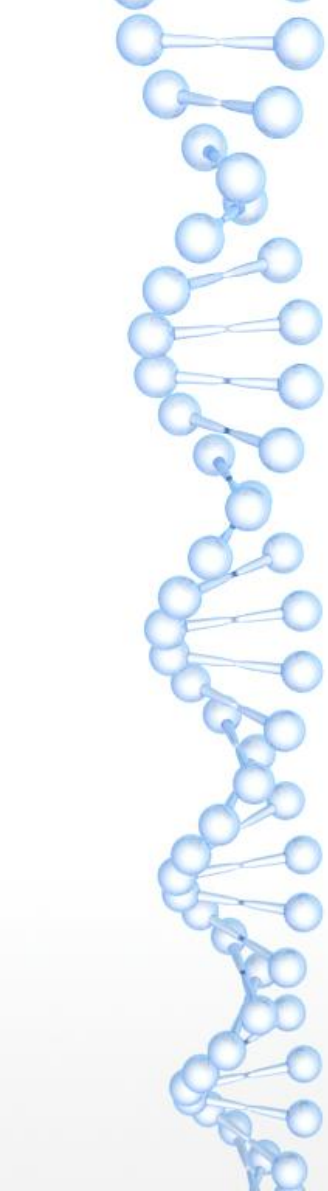
```
> type Thing = Plant | Animal | Mineral;;

type Thing =
    | Plant
    | Animal
    | Mineral

> // Ошибка: Значение Thing не может быть null
let testThing thing =
    match thing with
    | Plant -> "Plant"
    | Animal -> "Animal"
    | Mineral -> "Mineral"
    | null -> "(null)";;

    | null -> "(null)";;
------^^^^
stdin(9,7): error FS0043: The type 'Thing' does not have 'null' as a proper
value.
```

(Тип "Thing" не содержит собственное значение "null".)

Значимые и ссылочные типы

Такое ограничение на первый взгляд выглядит достаточно странным, но благодаря этому отпадает необходимость в излишних проверках на равенство значению `null`. (Если вызвать метод на неинициализированном объекте ссылочного типа, программа сгенерирует исключение `NullReferenceException`. Такая защитная проверка всех параметров функций на равенство значению `null` является обычным делом.) Если вам потребуется представить неинициализированное состояние в программе на языке F#, то вместо ссылочного типа со значением `null` можно использовать тип `Option`, где значение `None` представляет неинициализированное состояние, а значение `Some('a)` – инициализированное.

В языке F# к типу можно добавить специальный атрибут, чтобы обеспечить возможность присваивания данным этих типов значения `null` и упростить взаимодействие с другими языками. NET. Подробнее об этом рассказывается в приложении B.

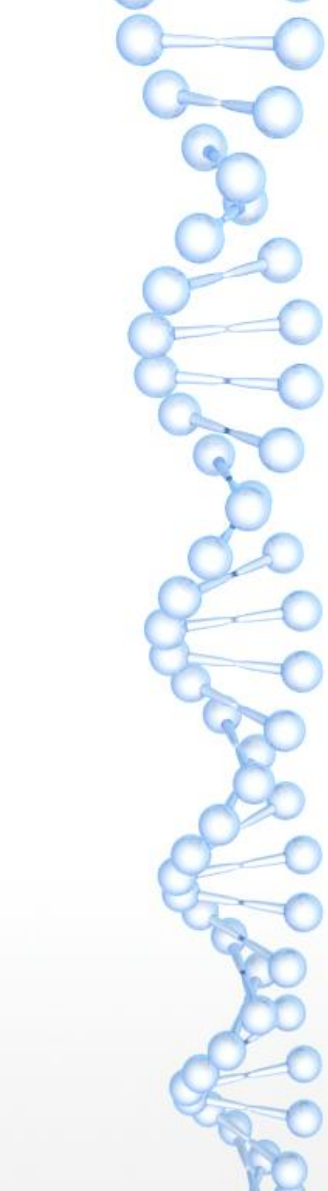


Значимые и ссылочные типы

```
> // Определение двух классов, один – с атрибутом [<AllowNullLiteral>],
// а другой – без атрибута
type Widget() =
    override this.ToString() = "A Widget"

[<AllowNullLiteral>]
type Sprocket() =
    override this.ToString() = "A Sprocket";;

type Widget =
  class
    new : unit -> Widget
    override ToString : unit -> string
  end
type Sprocket =
  class
    new : unit -> Sprocket
    override ToString : unit -> string
  end

> // ОШИБКА: экземпляр типа Widget не может иметь значение null
let x : Widget = null;;

  let x : Widget = null;;
  ----------------^^^^

stdin(12,18): error FS0043: The type 'Widget' does not have 'null' as a proper
value
```

(Тип "Widget" не содержит собственное значение "null".)

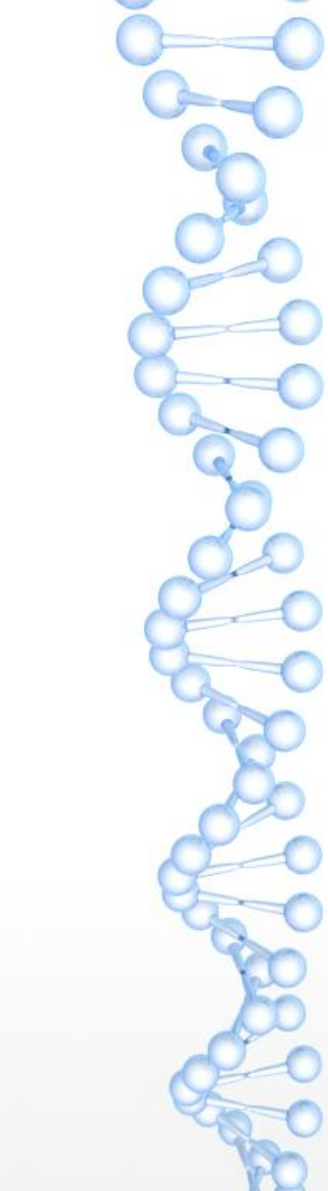
```
> // Нет ошибки
let x : Sprocket = null;;

val x : Sprocket = null
```



Значимые и ссылочные типы

Существует возможность двум ссылочным типам ссылаться на одну и ту же область памяти.[1] Этот прием известен как *совмещение имен (aliasing)*. В этом случае изменение одного значения приведет к изменению другого, потому что оба они ссылаются на одну и ту же область памяти. При неосторожном обращении такие ситуации могут приводить к ошибкам.

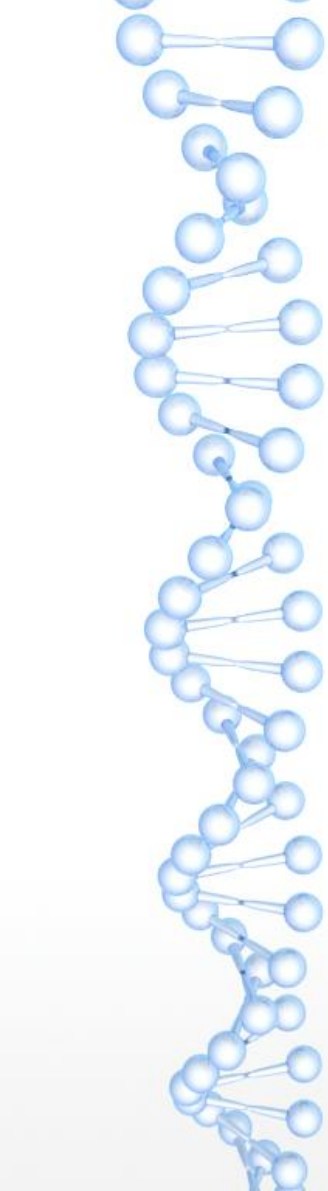
В примере 4.2 создается единственный экземпляр типа `array` (с которым мы скоро познакомимся поближе), но имеется два значения, указывающие на один и тот же экземпляр. При изменении значения `x` также изменяется и значение `y`, и наоборот.



Значимые и ссылочные типы

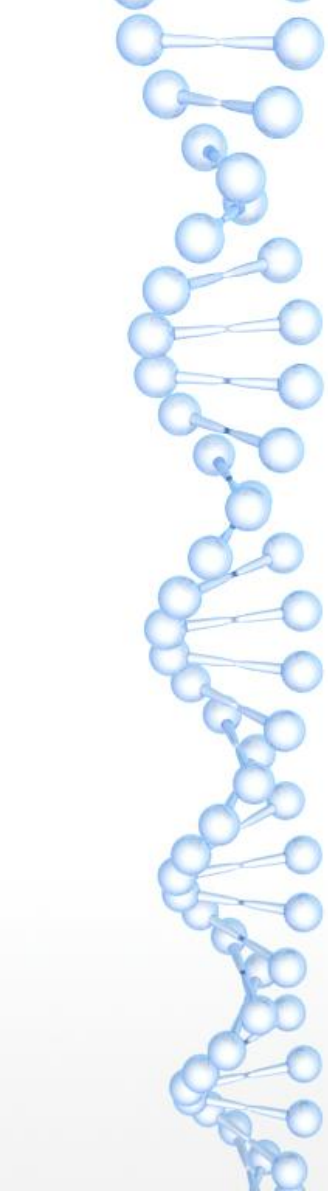
Пример 4.2. Совмещение имен ссылочных типов

```
> // Значение x указывает на массив, значение y указывает
// на тот же самый массив, что и значение x
let x = [| 0 |]
let y = x;;

val x : int [] = [|0|]
val y : int [] = [|0|]

> // Если изменить значение x...
x.[0] <- 3;;
val it : unit = ()
> // ... x изменится ...
x;;
val it : int [] = [|3|]
> // ... но при этом изменится и значение y ...
y;;
val it : int [] = [|3|]
```



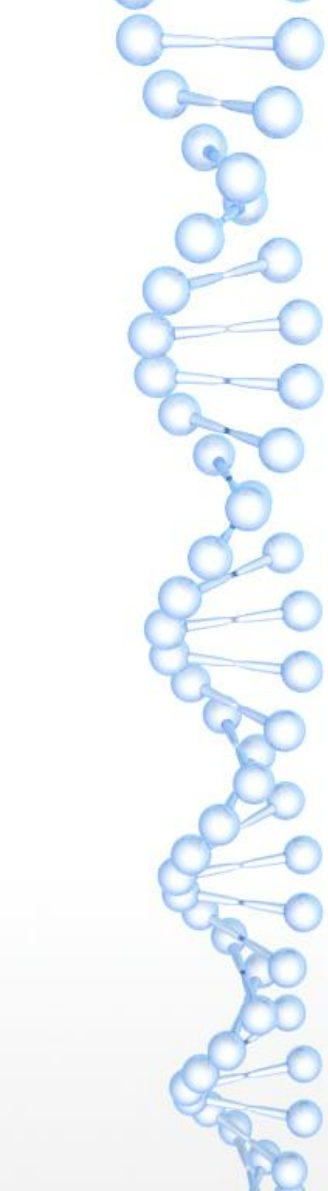
Изменение значений

```
> let mutable message = “World”;;

val mutable message : string = “World”

> printfn “Hello, %s” message;;
Hello, World
val it : unit = ()

> message <- “Universe”;;
val it : unit = ()
> printfn “Hello, %s” message;;
Hello, Universe
val it : unit = ()
```



Изменение значений

```
> // ОШИБКА: Изменяемые значения можно использовать только в функции,
//где они объявлены
let invalidUseOfMutable() =
    let mutable x = 0

    let incrementX() = x <- x + 1
    incrementX()

    x;;

    let incrementX() = x <- x + 1
----------------------^^^^^^^^^^^
stdin(16,24): error FS0407: The mutable variable 'x' is used in an invalid way.
Mutable variables may not be captured by closures. Consider eliminating this use
of mutation or using a heap-allocated mutable reference cell via 'ref' and '!'.
```

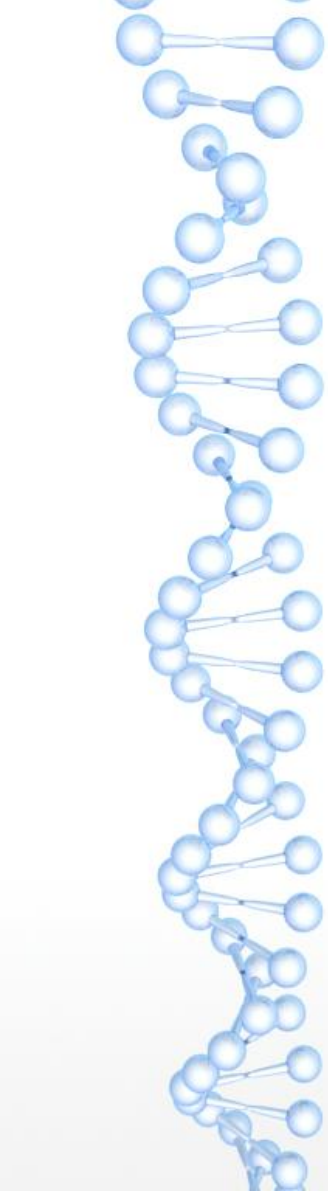


Изменение значений

Полный перечень ограничений, связанных с изменяемыми значениями:

- Изменяемые значения не могут возвращаться из функции (вместо этого возвращается копия).
- Изменяемые значения не могут использоваться вложенными функциями (замыканиями).

Если вы когда-либо столкнетесь с одной из этих проблем, самый простой способ решить ее состоит в том, чтобы сохранить изменяемые данные в «куче» с помощью ссылочной ячейки `ref`.



Ссылочные ячейки

Тип `ref`, который иногда называют *ссылочной ячейкой (ref cell)*, позволяет хранить изменяемые данные в «куче», что дает возможность обойти ограничения, связанные с изменяемыми значениями, хранящимися в стеке. Для получения значения типа `ref` используется символический оператор `!`, а для изменения – оператор `:=`.

`ref` – это не только название типа, но также имя функции, которая создает значения типа `ref` и имеет сигнатуру:

```
val ref: 'a -> 'a ref
```

Функция `ref` принимает значение и возвращает его копию, заключенную в ссылочную ячейку `ref`. Пример 4.4 демонстрирует передачу списка планет функции `ref` и последующее изменение возвращенного значения.

Ссылочные ячейки

Пример 4.4. Использование ссылочных ячеек для изменения данных

```
let planets =
    ref [
        “Mercury”; “Venus”; “Earth”;
        “Mars”; “Jupiter”; “Saturn”;
        “Uranus”; “Neptune”; “Pluto”
    ]

// Опа! Плутон уже не считается планетой...

// Оставить в списке планеты, кроме “Pluto”
// Получить значение planets типа ref cell с помощью оператора (!),
// затем присвоить ему новое значение с помощью оператора (:=)
planets := !planets |> List.filter (fun p -> p <> “Pluto”)
```

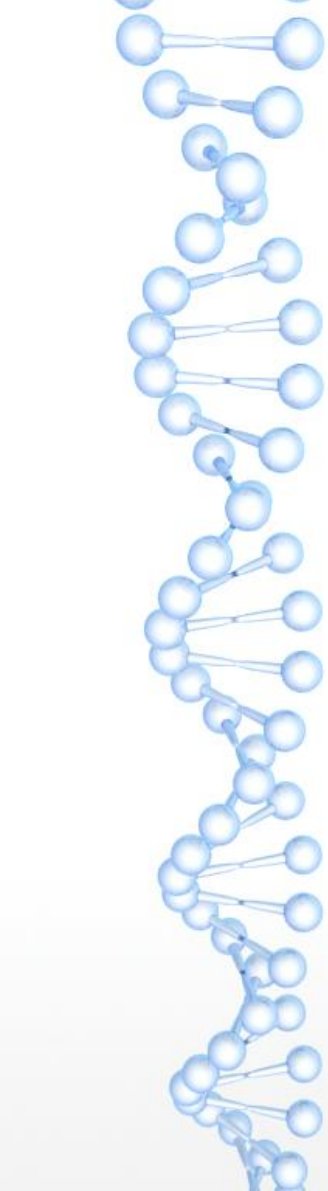
Ссылочные ячейки

Программисты на языке C# должны быть особенно внимательными при использовании данных типа `ref` и булевых значений. Запись `!x` означает получение значения `x`, а не логическое отрицание `x`:

```
> let x = ref true;;

val x : bool ref

> !x;;

val it : bool = true
```

В стандартной библиотеке языка F# имеется две функции, `decr` и `incr`, упрощающие увеличение и уменьшение данных типа `int ref`:

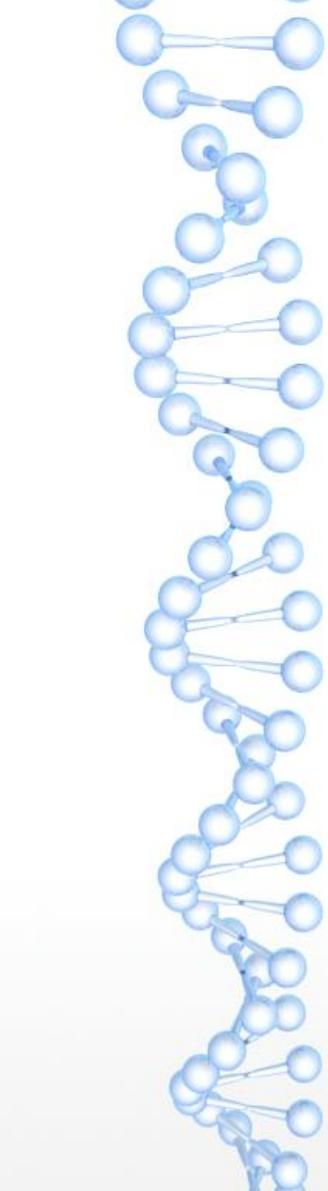
```
> let x = ref 0;;

val x : int ref = {contents = 0;}

> incr x;;
val it : unit = ()
> x;;
val it : int ref = {contents = 1;}
> decr x;;
val it : unit = ()
> x;;
val it : int ref = {contents = 0;}
```



Изменяемые записи

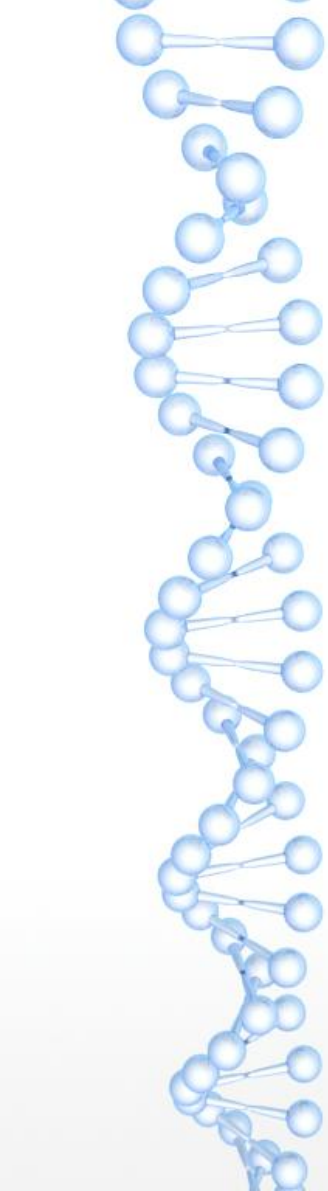
Изменяемыми могут быть не только простые значения – поля записей также могут быть помечены как изменяемые. Это позволяет использовать записи в императивном стиле. Чтобы сделать поле записи изменяемым, просто добавьте ключевое слово `mutable` перед его именем.

В следующем примере создается запись с изменяемым полем `Miles`, которое может модифицироваться, как обычная изменяемая переменная. Теперь у вас появилась возможность изменять поля записи без необходимости клонировать запись целиком:



Изменяемые записи

```
> // Изменяемые записи
open System

type MutableCar = { Make : string; Model : string; mutable Miles : int }

let driveForASeason car =
    let rng = new Random()
    car.Miles <- car.Miles + rng.Next() % 10000;;

type MutableCar =
  {Make: string;
   Model: string;
   mutable Miles: int;}
val driveForASeason : MutableCar -> unit

> // Изменение поля записи
let kitt = { Make = "Pontiac"; Model = "Trans Am"; Miles = 0 }

driveForASeason kitt
driveForASeason kitt
driveForASeason kitt
driveForASeason kitt;;

val kitt : MutableCar = {Make = "Pontiac";
                         Model = "Trans Am";
                         Miles = 4660;}
```



Массивы

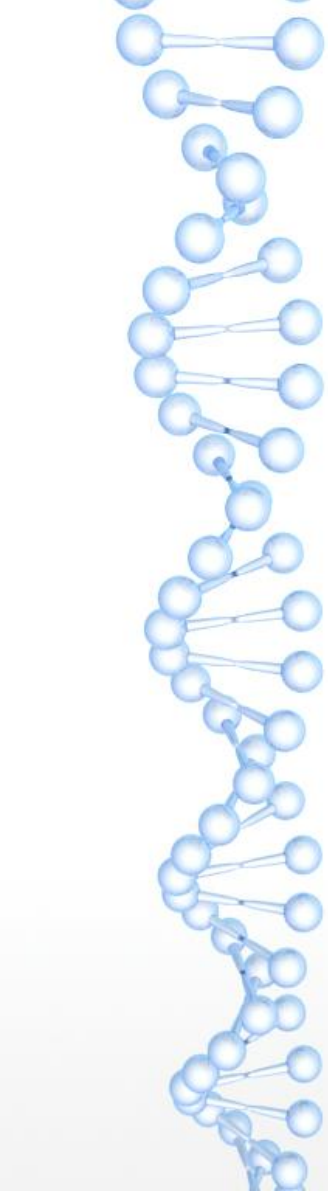
```
> // Использование синтаксиса генератора массивов
let perfectSquares = [| for i in 1 .. 7 -> i * i |];;

val perfectSquares : int [] = [|1; 4; 9; 16; 25; 36; 49|]

> perfectSquares;;
val it : int []= [|1; 4; 9; 16; 25; 36; 49|]

> // Объявление вручную
let perfectSquares2 = [| 1; 4; 9; 16; 25; 36; 49; 64; 81 |];;

val perfectSquares2 : int [] = [|1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

Массивы

Для получения элемента массива необходимо использовать оператор индексирования .[], где в квадратных скобках указывается индекс элемента (счет элементов начинается с нуля):

```
> // Обращение к элементам массива
printfn
    "The first three perfect squares are %d, %d, and %d"
    perfectSquares.[0]
    perfectSquares.[1]
    perfectSquares.[2];;
The first three perfect squares are 1, 4, and 9
val it : unit = ()
```

В примере 4.5 показан примитивный алгоритм шифрования ROT13, в котором с помощью оператора индексирования выполняется изменение отдельных элементов массива символов. Суть алгоритма ROT13 заключается в том, что он просто извлекает каждый символ и замещает его символом, отстоящим в алфавите на 13 позиций дальше. Для этого в примере каждый символ преобразуется в целое число, к этому числу добавляется 13, а затем получившееся значение преобразуется обратно в символ.



Массивы

```
> let alphabet = [| 'a' .. 'z' |];;

val alphabet : char [] =
  [|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'; 'i'; 'j'; 'k'; 'l'; 'm'; 'n'; 'o';
    'p'; 'q'; 'r'; 's'; 't'; 'u'; 'v'; 'w'; 'x'; 'y'; 'z'|]

> // Первый символ
alphabet.[0];;
val it : char = 'a'

> // Последний символ
alphabet.[alphabet.Length - 1];;
val it : char = 'z'

> // Некоторый несуществующий символ
alphabet.[10000];;
System.IndexOutOfRangeException: Index was outside the bounds of the array.
  at <StartupCode$FSI_0012>.$FSI_0012._main()
stopped due to error
```

(Индекс находился вне границ массива. Остановлено из-за ошибки.)

Массивы

Пример 4.6. Получение среза массива

```
open System
let daysOfWeek = Enum.GetNames( typeof<DayOfWeek> )

// Стандартный способ получения среза, элементы со 2 по 4
daysOfWeek.[2..4]

// Указана только нижняя граница, элементы с 4 по последний
daysOfWeek.[4..]

// Указана только верхняя граница, элементы с 0 по 2
daysOfWeek.[..2]

// Границы не указаны, возвращаются все элементы (копия всего массива)
daysOfWeek.[*]
```

Массивы

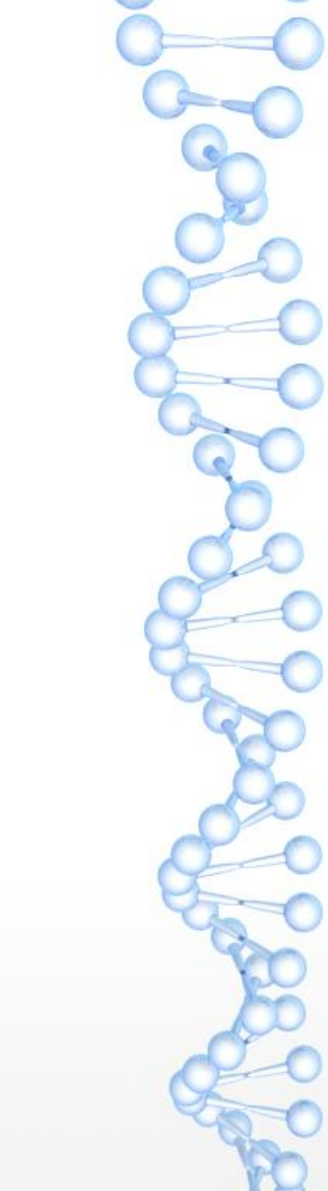
Пример 4.7. Инициализация массивов с помощью Array.init

```
> // Инициализация массива значениями синусоиды
let divisions = 4.0
let twoPi = 2.0 * Math.PI;;

val divisions : float = 4.0
val twoPi : float = 6.283185307

> Array.init (int divisions) (fun i -> float i * twoPi / divisions);;
val it : float []= [|0.0; 1.570796327; 3.141592654; 4.71238898|]
> // Конструирование пустых массивов
let emptyIntArray : int []= Array.zeroCreate 3
let emptyStringArray : string []= Array.zeroCreate 3;;

val emptyIntArray : int array = [|0; 0; 0|]
val emptyStringArray : string array = [|null; null; null|]
```

Массивы

Пример 4.8. Использование массивов с сопоставлением с образцом

```
> // Получить описание массива
let describeArray arr =
    match arr with
    | null       -> “The array is null”
    | [| |]      -> “The array is empty”
    | [| x |]    -> sprintf “The array has one element, %A” x
    | [| x; y |] -> sprintf “The array has two elements, %A and %A” x y
    | a          -> sprintf “The array had %d elements, %A” a.Length a;;

val describeArray : ‘a array -> string

> describeArray [| 1 .. 4 |];;
val it : string = “The array had 4 elements, [|1; 2; 3; 4|]”
> describeArray [| (“tuple”, 1, 2, 3) |];;
val it : string = “The array has one element, (“tuple”, 1, 2, 3)”
```



Массивы

Эквивалентность массивов

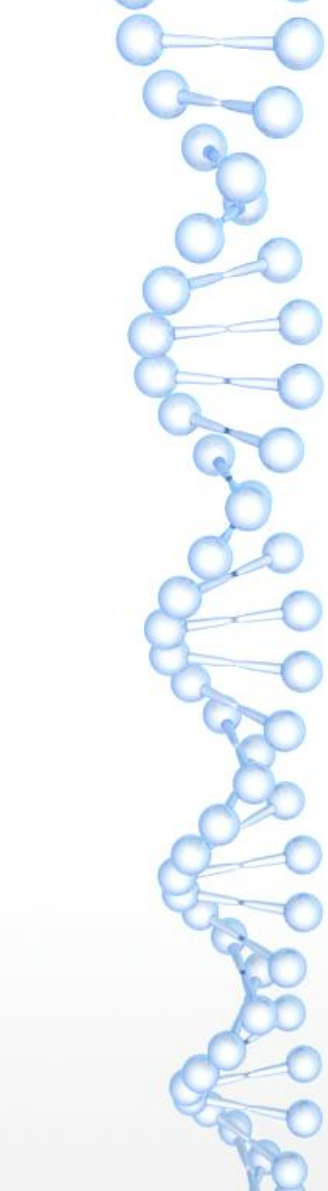
В языке F# массивы сравниваются с учетом структурной эквивалентности. Два массива считаются равными, если они имеют одинаковую размерность, длину и элементы. (Понятие размерности массива мы будем рассматривать в следующем разделе.)

```
> [| 1 .. 5 |] = [| 1; 2; 3; 4; 5 |];;
val it : bool = true
> [| 1 .. 3 |] = [| |];;
val it : bool = false
```

Массивы

Таблица 4.1. Часто используемые методы модуля Array

Функция	Описание
`Array.length` `'a[] -> int`	Возвращает длину массива. Массивы уже имеют свойство `Length`, но эту функцию удобно использовать для композиции функций.
`Array.init` `int -> (int -> 'a) -> 'a[]`	Создает новый массив с указанным числом элементов – каждый элемент инициализируется значением, возвращаемым указанной функцией.
`Array.zeroCreate` `int -> 'a[]`	Создает массив указанной длины, где каждый элемент массива получает значение по умолчанию
`Array.exists` `('a -> bool) -> 'a[] -> bool`	Проверяет наличие элемента массива, удовлетворяющего заданной функции.



Массивы

Array.partition

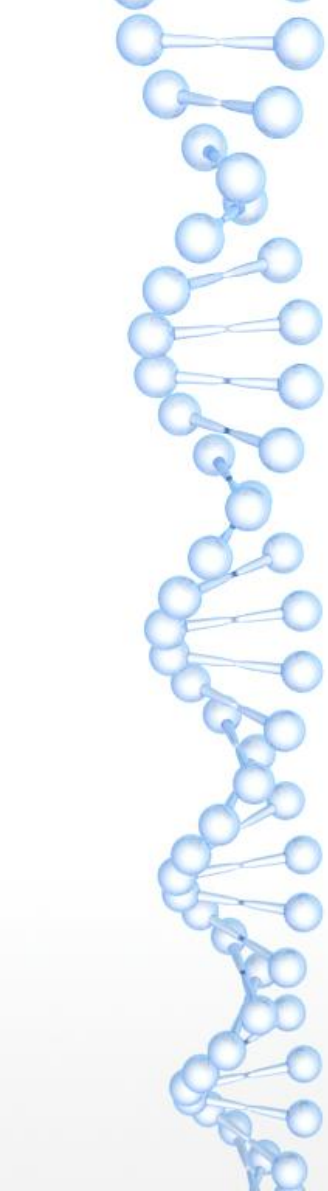
Функция `Array.partition` делит указанный массив на два новых массива. Первый массив получает элементы, для которых указанная функция возвратит `true`, а второй массив – элементы, для которых указанная функция возвратит `false`:

```
> // Простая булевая функция
let isGreaterThanTen x =
    if x > 10
    then true
    else false;;

val isGreaterThanTen : int -> bool

> // Деление массива
[| 5; 5; 6; 20; 1; 3; 7; 11 |]
|> Array.partition isGreaterThanTen;;
val it : int [] * int []= ([|20; 11|], [|5; 5; 6; 1; 3; 7|])
```

Массивы

```
> // Простая булевая функция
let rec isPowerOfTwo x =
    if x = 2 then
        true
    elif x % 2 = 1 (* is odd *) then
        false
    else isPowerOfTwo (x / 2);;

val isPowerOfTwo : int -> bool

> [| 1; 7; 13; 64; 32 |]
|> Array.tryFind isPowerOfTwo;;
val it : int option = Some 64
> [| 1; 7; 13; 64; 32 |]
|> Array.tryFindIndex isPowerOfTwo;;
val it : int option = Some 3
```

Функции Array.tryFind и Array.tryFindIndex наглядно показывают удобство использования типа Option. В языке C# функции, похожие на tryFindIndex, могли бы возвращать значение -1 как признак отсутствия искомого элемента (в противоположность значению None). Однако для функции tryFind значение -1 может интерпретироваться не только как признак отсутствия искомого элемента, но и как элемент со значением -1.



Массивы

Агрегатные операторы

Модуль Array содержит также агрегатные операторы, аналогичные тем, что имеются в модуле List, а именно: fold, foldBack, map и iter. Кроме того, существуют также версии этих методов, содержащие индексы. В примере 4.9 показано использование функции iteri, которая ведет себя точно так же, как и функция iter, за исключением того, что вместе с самим элементом массива она передает его индекс.

Пример 4.9. Использование агрегатной функции Array.iteri

```
> let vowels = [| 'a'; 'e'; 'i'; 'o'; 'u' |];;

val vowels : char [] = [|'a'; 'e'; 'i'; 'o'; 'u'|]

> Array.iteri (fun idx chr -> printfn "vowel.[%d] = %c" idx chr) vowels
vowel.[0] = a
vowel.[1] = e
vowel.[2] = i
vowel.[3] = o
vowel.[4] = u
val it : unit = ()
```

Массивы

```
> // Создание массива 3x3
let identityMatrix : float[,] = Array2D.zeroCreate 3 3
identityMatrix.[0,0] <- 1.0
identityMatrix.[1,1] <- 1.0
identityMatrix.[2,2] <- 1.0;;

val identityMatrix : float [,] = [[1.0; 0.0; 0.0]
                                  [0.0; 1.0; 0.0]
                                  [0.0; 0.0; 1.0]]
```

Двумерные прямоугольные массивы также позволяют получать срезы с помощью аналогичного синтаксиса путем указания диапазона для каждого измерения отдельно:

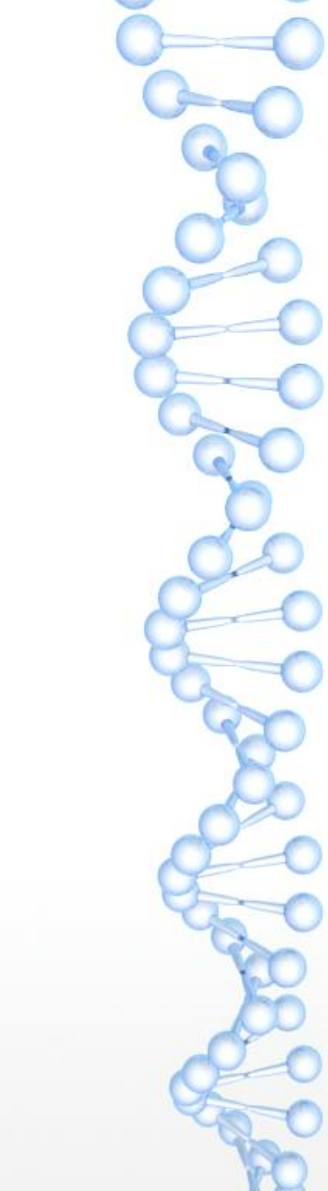
```
> // Все строки для столбцов с индексами 1 и 2
identityMatrix.[*, 1..2];;

val it : float [,] = [[0.0; 0.0]
                      [1.0; 0.0]
                      [0.0; 1.0]]
```

Массивы

Невыровненные массивы

Невыровненные массивы – это просто одномерные массивы одномерных массивов. Каждая строка в основном массиве – это просто другой массив, каждый из которых должен инициализироваться отдельно:

```
> // Создание невыровненного массива
let jaggedArray : int[][] = Array.zeroCreate 3
jaggedArray.[0] <- Array.init 1 (fun x -> x)
jaggedArray.[1] <- Array.init 2 (fun x -> x)
jaggedArray.[2] <- Array.init 3 (fun x -> x);;
val jaggedArray : int [] [] = [|[|0|]; [|0; 1|]; [|0; 1; 2|]|]
```



Типы изменяемых коллекций

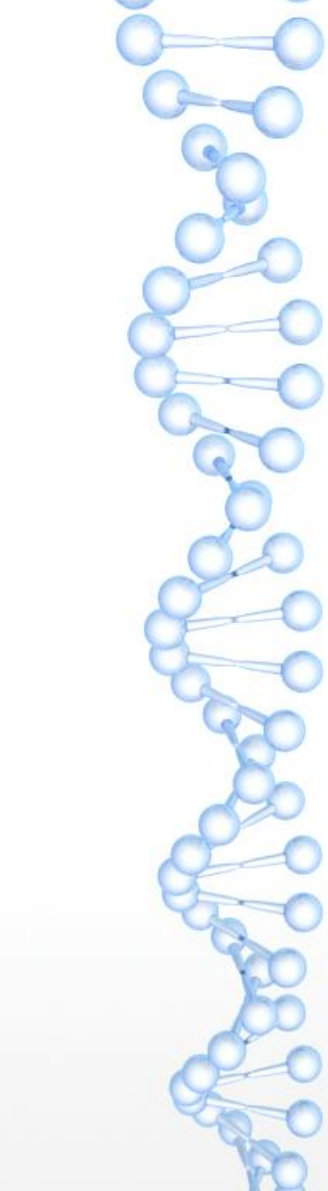
Пример 4.10. Использование типа List

```
> // Создается список планет типа List<_>
open System.Collections.Generic
let planets = new List<string>();;

val planets : Collections.Generic.List<string>

> // Добавить несколько названий планет по отдельности

planets.Add("Mercury")
planets.Add("Venus")
planets.Add("Earth")
planets.Add("Mars");;
> planets.Count;;

val it : int = 4
> // Добавить сразу несколько значений
planets.AddRange( [| "Jupiter"; "Saturn"; "Uranus"; "Neptune"; "Pluto" |] );;
val it : unit = ()
> planets.Count;;
val it : int = 9
> // Прошу прощения, ребята
planets.Remove("Pluto");;
val it : bool = true
> planets.Count;;
val it : int = 8
```

Типы изменяемых коллекций

Таблица 4.2. Методы и свойства типа List<'T>

Функция и ее тип	Описание
`Add` `'a -> unit`	Добавляет элемент в конец списка.
`Clear` `unit -> unit`	Удаляет все элементы из списка.
`Contains` `'a -> bool`	Возвращает признак наличия в списке указанного элемента.
`Count` `int`	Свойство, которое возвращает количество элементов в списке.
`IndexOf` `'a -> int`	Возвращает индекс указанного элемента (отсчет индексов начинается с нуля). Если искомый элемент отсутствует, возвращает -1.
`Insert` `int * 'a -> unit`	Вставляет указанное значение в список, в позицию с указанным индексом.
`Remove` `'a -> bool`	Удаляет указанный элемент, если он присутствует в списке.
`RemoveAt` `int -> unit`	Удаляет элемент с указанным индексом.

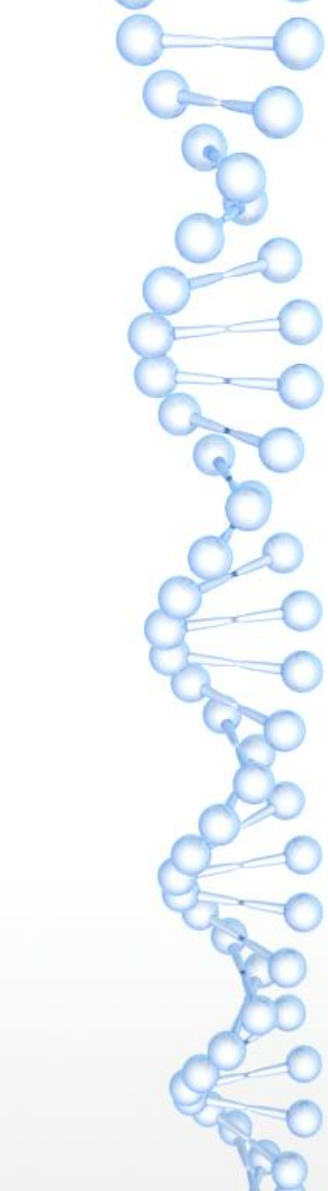


Типы изменяемых коллекций

```
open System.Collections.Generic
let periodicTable = new Dictionary<string, Atom>()
periodicTable.Add( "H", { Name = "Hydrogen"; Weight = 1.0079<amu> })
periodicTable.Add("He", { Name = "Helium"; Weight = 4.0026<amu> })
periodicTable.Add("Li", { Name = "Lithium"; Weight = 6.9410<amu> })
periodicTable.Add("Be", { Name = "Beryllium"; Weight = 9.0122<amu> })
periodicTable.Add( "B", { Name = "Boron "; Weight = 10.811<amu> })
// ...

// Поиск элемента
let printElement name =

    if periodicTable.ContainsKey(name) then
        let atom = periodicTable.[name]
        printfn
            "Atom with symbol with '%s' has weight %A."
            atom.Name atom.Weight
    else
        printfn "Error. No atom with name '%s' found." name

// Альтернативный способ получения значений. Возвращает кортеж
// 'success * result'
let printElement2 name =

    let (found, atom) = periodicTable.TryGetValue(name)
    if found then
        printfn
            "Atom with symbol with '%s' has weight %A."
            atom.Name atom.Weight
    else
        printfn "Error. No atom with name '%s' found." Name
```

Типы изменяемых коллекций

Функция и ее тип	Описание
`Add` `'k * 'v -> unit`	Добавляет в словарь новую пару ключ/значение.
`Clear` `unit -> unit`	Удаляет все элементы из словаря.
`ContainsKey` `'k -> bool`	Проверяет наличие указанного ключа в словаре.
`ContainsValue` `'v -> bool`	Проверяет наличие указанного значения в словаре.
`Count` `int`	Возвращает количество элементов в словаре.
`Remove` `'k -> bool`	Удаляет из словаря пару ключ/значение с указанным ключом.



Типы изменяемых коллекций

Тип HashSet, который также определен в пространстве имен System.Collections.Generic, обеспечивает эффективный способ хранения неупорядоченного набора элементов. Представьте, что вы создаете приложение, которое будет просматривать содержимое веб-страниц. Вам придется каким-то образом запоминать, какие страницы уже были просмотрены, чтобы не застрять в бесконечном цикле. Однако если сохранять адреса посещенных страниц в списке типа List, проверка наличия очередной страницы в списке посещенных ранее будет выполняться слишком медленно. Тип HashSet хранит коллекцию уникальных значений на основе *хеш-кодов*, благодаря чему проверка наличия элемента в коллекции выполняется намного быстрее.

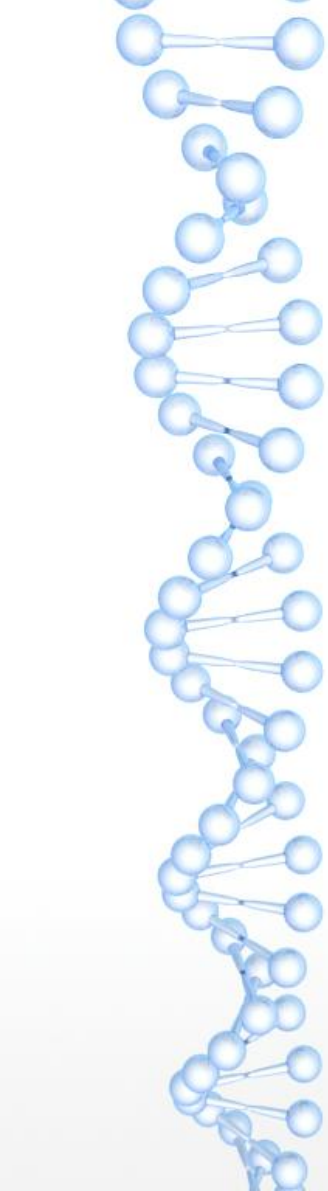
Подробнее о хеш-кодах рассказывается в следующей главе, а пока считайте их одним из способов идентификации объектов. Именно они обеспечивают высокую скорость поиска элементов в коллекциях типа HashSet и Dictionary.

В примере 4.12 показано использование коллекции HashSet для определения того, была ли кинокартина удостоена премии Оскар.

Типы изменяемых коллекций

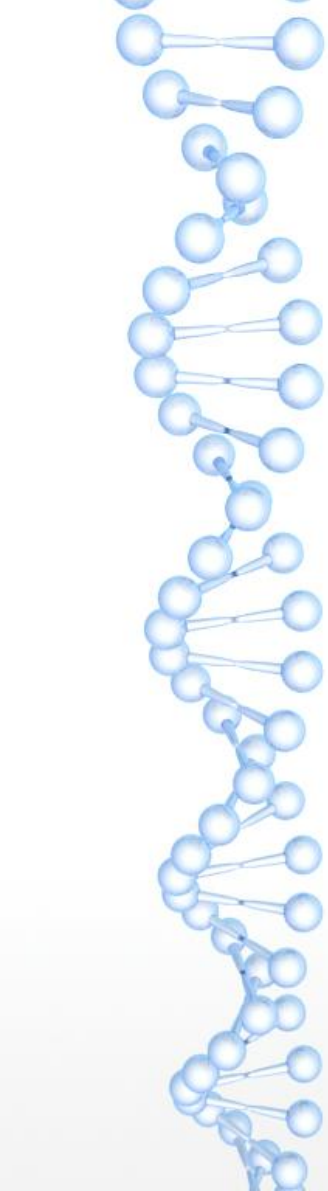
Пример 4.12. Использование типа HashSet

```
open System.Collections.Generic

let bestPicture = new HashSet<string>()
bestPicture.Add("No Country for Old Men")
bestPicture.Add("The Departed")
bestPicture.Add("Crash")
bestPicture.Add("Million Dollar Baby")
// ...

// Проверить, была ли удостоена премии Оскар следующая кинокартина
if bestPicture.Contains("Manos: The Hands of Fate") then
    printfn "Sweet..."
    // ...
```



Типы изменяемых коллекций

Функция и ее тип	Описание
`Add` `'a -> unit`	Добавляет новый элемент в коллекцию типа `HashSet`.
`Clear` `unit -> unit`	Удаляет все элементы из коллекции типа `HashSet`.
`Count` `int`	Возвращает количество элементов в коллекции типа `HashSet`.
`IntersectWith` `seq<'a> -> unit`	Изменяет коллекцию типа `HashSet` так, чтобы она содержала только те элементы, которые также присутствуют в указанной последовательности.
`IsSubsetOf` `seq<'a > -> bool`	Определяет, является ли коллекция типа `HashSet` подмножеством указанной последовательности. То есть последовательность содержит все элементы, имеющиеся в коллекции типа `HashSet`.
`IsSupersetOf` `seq<'a > -> bool`	Определяет, является ли коллекция типа `HashSet` надмножеством указанной последовательности. То есть коллекция типа `HashSet` содержит все элементы, имеющиеся в последовательности.
`Remove` `'a -> bool`	Удаляет указанный элемент из коллекции.
`UnionWith` `seq<'a> -> unit`	Изменяет коллекцию типа `HashSet` так, чтобы она содержала все элементы, которые имеются в данной коллекции и в указанной последовательности.

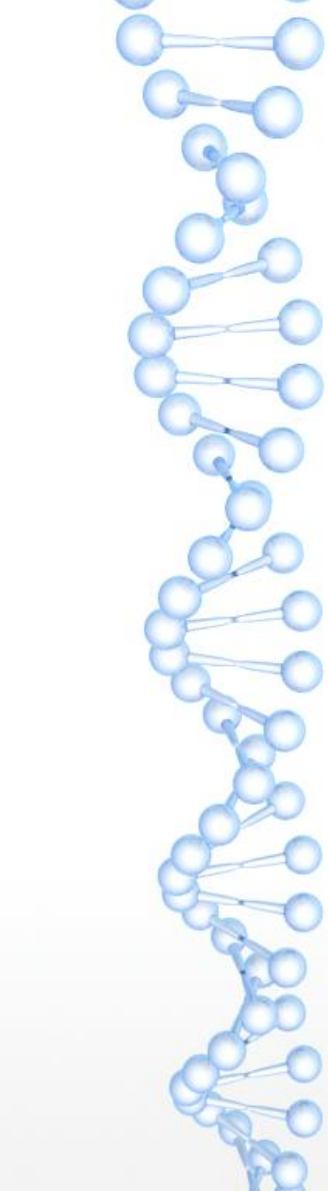


Циклы

```
> // Цикл for
for i = 1 to 5 do
    printfn "%d" i;;
1
2
3
4
5
val it : unit = ()
```

Чтобы организовать счет в обратном порядке, следует использовать ключевое слово downto. В этом случае цикл не будет выполняться, если значение счетчика окажется меньше минимального значения:

```
> // Счет выполняется в обратном порядке
for i = 5 downto 1 do
    printfn "%d" i;;
5
4
3
2
1
val it : unit = ()
```



Циклы

```
> // Цикл-перечисление
for i in [1 .. 5] do
    printfn "%d" i;;
1
2
3
4
5
val it : unit = ()
```

Циклы

Пример 4.13. Циклы for с сопоставлением с образцом

```
> // Тип животного
type Pet =
    | Cat of string * int // Кличка, Возраст
    | Dog of string       // Кличка
;;

type Pet =
    | Cat of string * int
    | Dog of string

> let famousPets = [ Dog("Lassie"); Cat("Felix", 9); Dog("Rin Tin Tin") ];;

val famousPets : Pet list = [Dog "Lassie"; Cat ("Felix",9); Dog "Rin Tin Tin"]

> // Вывести клички собак
for Dog(name) in famousPets do
    printfn "%s was a famous dog." name;;
Lassie was a famous dog.
```

Исключения

Самый простой способ сообщить об ошибке в программе на языке F# заключается в использовании функции failwith. Эта функция принимает строку в виде параметра и генерирует исключение System.Exception. Альтернативная версия, функция failwithf, принимает строку формата, подобно функциям printf и sprintf:

```
> // Использование функции failwithf
let divide x y =
    if y = 0 then failwithf "Cannot divide %d by zero!" x
    x / y;;

val divide : int -> int -> int

> divide 10 0;;
System.Exception: Cannot divide 10 by zero!
  at FSI_0003.divide(Int32 x, Int32 y)
  at <StartupCode$FSI_0004>.$FSI_0004._main()
stopped due to error
```

Исключения

```
> // Генерация исключения DivideByZeroException
let divide2 x y =
    if y = 0 then raise <| new System.DivideByZeroException()
    x / y;;

val divide2 : int -> int -> int

> divide2 10 0;;

System.DivideByZeroException: Attempted to divide by zero.
  at FSI_0005.divide2(Int32 x, Int32 y)
  at <StartupCode$FSI_0007>.$FSI_0007._main()
stopped due to error
```

(Попытка деления на нуль. Остановлено из-за ошибки)

Весьма заманчивой выглядит возможность генерировать исключения всякий раз, когда программа переходит в непредусмотренное состояние. Однако генерация исключений может существенно снизить производительность программы. Любые ситуации, которые могут привести к генерации исключения, должны устраняться по мере возможности.

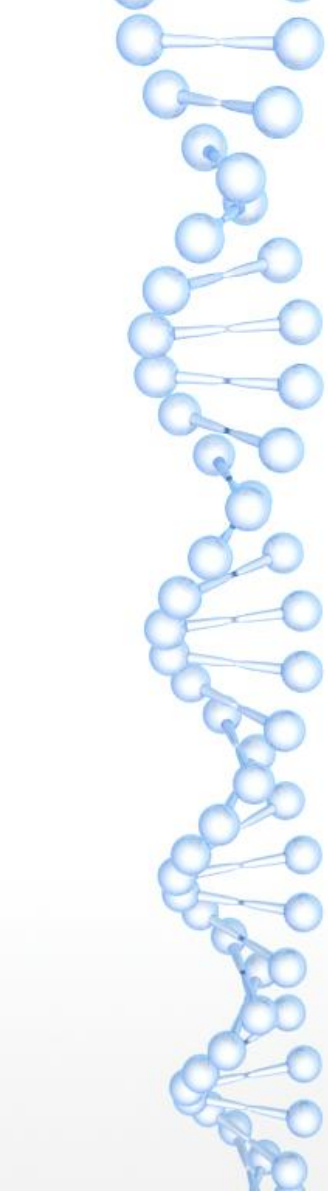
Исключения

```
open System.IO

[<EntryPoint>]
let main (args : string[]) =

    let exitCode =
        try
            let filePath = args.[0]

            printfn "Trying to gather information about file:"
            printfn "%s" filePath

            // Существует ли устройство?
            let matchingDrive =
                Directory.GetLogicalDrives()
                |> Array.tryFind (fun drivePath -> drivePath.[0] = filePath.[0])

            if matchingDrive = None then
                raise <| new DriveNotFoundException(filePath)

            // Указанная папка существует?
            let directory = Path.GetPathRoot(filePath)
            if not <| Directory.Exists(directory) then
                raise <| new DirectoryNotFoundException(filePath)

            // Файл существует?
            if not <| File.Exists(filePath) then
                raise <| new FileNotFoundException(filePath)

            let fileInfo = new FileInfo(filePath)
            printfn "Created = %s" <| fileInfo.CreationTime.ToString()
            printfn "Access  = %s" <| fileInfo.LastAccessTime.ToString()
            printfn "Size    = %d" fileInfo.Length

            0
        with
        // Объединение образцов по "ИЛИ"
        | :? DriveNotFoundException
        | :? DirectoryNotFoundException
            -> printfn "Unhandled Drive or Directory not found exception"
               1
        | :? FileNotFoundException as ex
            -> printfn "Unhandled FileNotFoundException: %s" ex.Message
               3
        | :? IOException as ex
            -> printfn "Unhandled IOException: %s" ex.Message
               4
        // Групповой символ (результат будет иметь тип System.Exception)
        | _ as ex
            -> printfn "Unhandled Exception: %s" ex.Message
               5

    // Вернуть код завершения
    printfn "Exiting with code %d" exitCode
    exitCode
```

Исключения

Пример 4.15. Выражения try-finally

```
> // Выражения try-finally
let tryFinallyTest() =
    try
        printfn "Before exception..."
        failwith "ERROR!"
        printfn "After exception raised..."
    finally
        printfn "Finally block executing..."

let test() =
    try
        tryFinallyTest()
    with
    | ex -> printfn "Exception caught with message: %s" ex.Message;;

val tryFinallyTest : unit -> unit
val test : unit -> unit


> test();;
Before exception...
Finally block executing...
Exception caught with message: ERROR!
val it : unit = ()
```



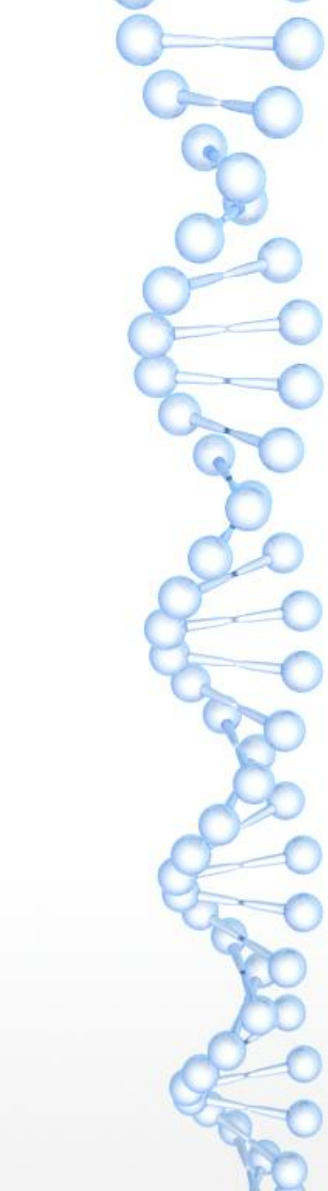
Исключения

Пример 4.16. Повторная генерация исключений

```
open Checked

let reraiseExceptionTest() =
    try
        let x = 0x0fffffff
        let y = 0x0fffffff

        x * y
    with
    | :? System.OverflowException as ex
        -> printfn "An overflow exception occured..."
           reraise()
```

Исключения

```
open System
open System.Collections.Generic

exception NoMagicWand
exception NoFullMoon of int * int
exception BadMojo of string

let castHex (ingredients : HashSet<string>) =
    try

        let currentWand = Environment.MagicWand

        if currentWand = null then
            raise NoMagicWand

        if not <| ingredients.Contains("Toad Wart") then
            raise <| BadMojo("Need Toad Wart to cast the hex!")

        if not <| isFullMoon(DateTime.Today) then
            raise <| NoFullMoon(DateTime.Today.Month, DateTime.Today.Day)

        // Начинаем колдовство...
        let mana =
            ingredients
            |> Seq.map (fun i -> i.GetHashCode())
            |> Seq.fold (+) 0

        sprintf "%x" mana

    with
    | NoMagicWand
        -> "Error: A magic wand is required to hex!"
    | NoFullMoon(month, day)
        -> "Error: Hexes can only be cast during a full moon."
    | BadMojo(msg)
        -> sprintf "Error: Hex failed due to bad mojo [%s]" msg
```



Исключения