МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

Кафедра математической кибернетики и компьютерных наук

**ОСНОВЫ РАБОТЫ С F# - ФУНКЦИИ (ФУНКЦИОНАЛЬНЫЙ СТИЛЬ) - КЛАССЫ (ОБЪЕКТНО - ОРИЕНТИРОВАННЫЙ СТИЛЬ)**

ЛАБОРАТОРНАЯ РАБОТА №2

(Вариант №5)

студента 3 курса 351 группы

направления 09.03.04 — Программная инженерия

факультета КНиИТ

Голикова Артема Олеговича

Проверил

к. пед. н., доцент \_\_\_\_\_\_\_\_\_\_\_\_ В. А. Векслер

Саратов 2020

**Часть 1-я «Функции»**

**Цель работы:** ознакомиться с понятием функции в F#, на примере нескольких простых задач научиться работать с функциями.

**Краткие теоретические сведения**

F# — это язык программирования, обеспечивающий поддержку функционального программирования, а также объектно-ориентированного и императивного (процедурного) программирования.

Язык программирования F# поддерживает следующие конструкции функционального программирования:

* Функции в качестве значений — позволяет гибко управлять функциями.
* Объединение и конвейеризация функций — позволяет объединять функции для создания новых функций и упрощения кодирования последующих операций с данными.
* Определение типа — устраняет необходимость явно вызывать типы без ущерба для безопасности типа.
* Автоматическое обобщение — позволяет повторно использовать код, упрощая написание кода, который работает с множеством разных типов без дополнительных усилий.
* Поддержка сопоставления шаблонов, упрощающая сложный код условия, и размеченные объединения, которые оптимизируются для использования с сопоставлением шаблонов.
* Лямбда-выражения — важны для многих конструкций функционального программирования.
* Частичное применение аргументов функций — обеспечивает возможность неявного создания новых функций из существующих.

Набирать код вы можете в рамках проекта или в рамках работы с интерактивным окном.

Visual F# предоставляет интерактивное окно, интегрированное в среду разработки Visual Studio. Данное окно позволяет вводить код F#, который сразу же компилируется и выполняется. Это позволяет легко создавать прототипы конструкций кода и проверять код при его написании. В интерактивном окне запускается средство интерактивного режима F# (fsi.exe), которое можно также запускать из командной строки. Эта функция предоставляет возможность использовать язык F# в качестве скриптового языка.

В самом интерактивном окне вы можете вводить операторы в необходимом количестве, окончанием набора служат два знака «точка с запятой». После их набора и нажав на клавишу Enter код будет исполнен.

> printf "%A" (3\*3-45);;

-36

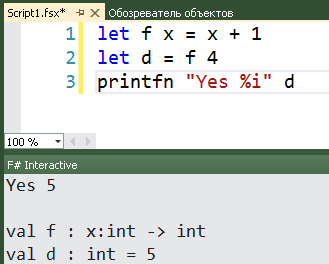
Более удобным вариантом может быть использование файла скрипта.

В Microsoft Visual Studio: Файл – Создать – Файл – Файл скрипта F#.

Нажмите Ctrl-Alt-F для вызова интерактивного окна, позволяющего более подробно просматривать выполненные операции.

Набрав операторы в скрипте, необходимо из выделить и нажать ALT+Enter. Результат появится в интерактивном окне. Так же в интерактивном окне вы увидите и описание всех созданных функций.

Рассмотрите пример:

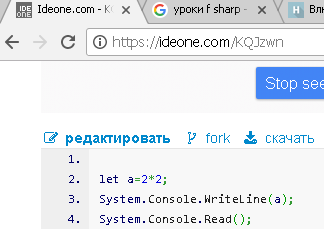


Конструкция printfn позволяет вывести значения по параметру.

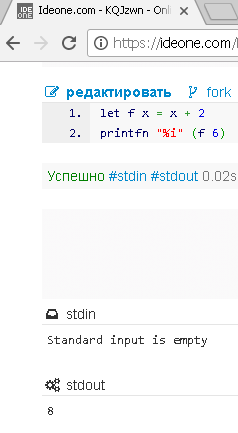
Параметр: %i - на это место выставится значение целочисленной переменной, %f – вещественной переменной.

Для работы с языком программирования вы можете использовать онлайн компилятор. Например ideone.com

Вывод на экран используя стандартные библиотеки:



Пример в онлайн компиляторе:



Функции — это основной элемент выполнения программы в любом языке программирования. Как и в других языках, функция в языке F# имеет имя, может иметь параметры и принимать аргументы, а также функция имеет тело. Язык F# также поддерживает конструкции функционального программирования, например обработку функций как значений, использование в выражениях неименованных функций, объединение функций для образования новых функций, каррированные функции и неявное определение функций посредством частичного применения аргументов функции.

Функции определяются с помощью ключевого слова let или, если функция рекурсивная, комбинации ключевых слов let rec.

Создание константы:

let duplicated = "original value"

Создание переменной:

let mutable modifiable = "original value"

modifiable <- "new value"

Простое определение функции выглядит примерно следующим образом.

let f x = x + 1

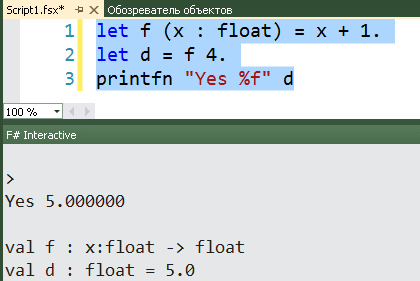
В предыдущем примере имя функции = f, аргумент = x и принадлежит к типу int, тело функции = x + 1, возвращаемое значение имеет тип int. Тело функции определено отступами (не менее длины имени функции).

*Параметры*

Имена параметров перечислены после имени функции. Можно задать тип для параметра, как показано в следующем примере.

let (x : int) = x + 1

Если тип задан, он следует за именем параметра, отделенный двоеточием.



Если тип параметра не указан, он будет выведен компилятором. Например, в следующем определении функции тип аргумента x выведен как тип int, поскольку "1" принадлежит к типу int.

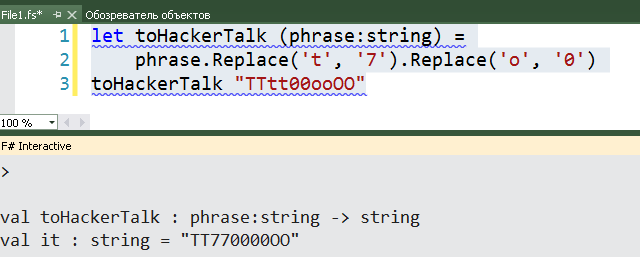
let f x = x + 1

Однако компилятор попытается сделать функцию насколько возможно универсальной. Например, рассмотрим следующий код:

let f x = (x, x)

Функция создает кортеж из одного аргумента типа any. Поскольку тип не задан, функция может использоваться с типом аргумента any.

Пример задания функции обработки строки:



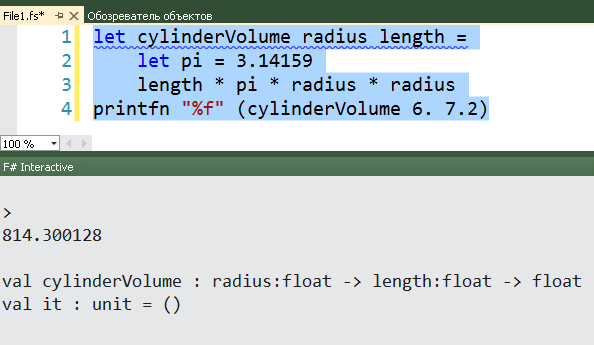
*Тело функции*

Тело функции может содержать определения локальных переменных и функций. Область таких переменных и функций — в теле текущей функции, но не вне его. Если включена возможность облегченного синтаксиса, необходимо использовать отступ для обозначения того, что определение находится внутри тела функции, как показано в следующем примере.

let cylinderVolume radius length =

let pi = 3.14159

length \* pi \* radius \* radius



Поясним: создается функция cylinderVolume которая обрабатывает два параметра и возвращает одно значение. В теле функции два оператора: объявления числа Пи и вычисление результата.

*Возвращаемые значения*

Компилятор использует результирующее значение в теле функции, чтобы определить возвращаемое значение и тип. Компилятор может вывести тип результирующего выражения из предшествующих выражений. В функции cylinderVolume, показанной в предыдущем разделе, тип объекта pi определен по типу литерала 3.14159 как float. Компилятор использует тип pi, чтобы определить тип выражения h \* pi \* r \* r как float. Поэтому общий возвращаемый тип функции — float.

Чтобы явно задать возвращаемое значение, напишите код следующим образом.

let cylinderVolume radius length : float =

let pi = 3.14159

length \* pi \* radius \* radius

*Вызов функции*

Вызов функций осуществляется путем указания имени функции, пробела и следующих за ним аргументов через пробел. Например, чтобы вызвать функцию cylinderVolume и назначить результат значению vol, следует написать приведенный ниже код.

let vol = cylinderVolume 2.0 3.0

*Частичное применение аргументов*

Если предоставить меньшее число аргументов, чем задано, будет создана новая функция, ожидающая оставшиеся аргументы. Такой способ работы с аргументами называется каррированием и характерен для языков функционального программирования, таких как F#. Например, предположим, вы работаете с двумя размерами труб: одна труба имеет радиус 2.0, а другая — 3.0. Можно было бы создать функции, определяющие объем трубы следующим образом.

let smallPipeRadius = 2.0

let bigPipeRadius = 3.0

let smallPipeVolume = cylinderVolume smallPipeRadius

let bigPipeVolume = cylinderVolume bigPipeRadius

Затем можно было бы предоставить дополнительный аргумент длины трубы двух различных радиусов.

let length1 = 30.0

let length2 = 40.0

let smallPipeVol1 = smallPipeVolume length1

let smallPipeVol2 = smallPipeVolume length2

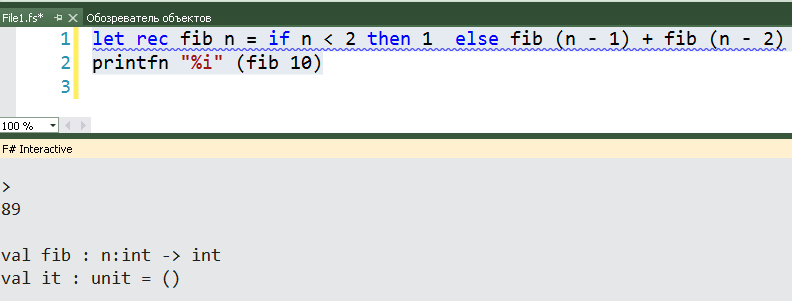
let bigPipeVol1 = bigPipeVolume length1

let bigPipeVol2 = bigPipeVolume length2

*Рекурсивные функции*

Рекурсивные функции — функции, вызывающие самих себя. Они требуют, чтобы вслед за ключевым словом let было указано ключевое слово rec. Рекурсивную функцию можно вызвать из тела функции как любую другую. Следующая рекурсивная функция вычисляет элемент последовательности Фибоначчи с индексом n. Последовательность чисел Фибоначчи известна с античных времен и является последовательностью, в которой каждый последующий элемент является суммой двух предыдущих.

let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2)



*Функциональные значения*

В языке F# все функции считаются значениями, которые известны как функциональные значения. Так как функции являются значениями, их можно использовать как аргументы для других функций или в других контекстах, где используются значения. Далее приведен пример функции, которая принимает как аргумент функциональное значение.

let apply1 (transform : int -> int ) y = transform y

Тип функционального значения можно задать с помощью токена ->. В левой части токена находится тип аргумента, а в правой части — возвращаемое значение. В предыдущем примере apply1 является функцией, принимающей функцию transform как аргумент, где transform является функцией, которая принимает целое число и возвращает другое целое число. Следующий код показывает, как использовать функцию apply1.

let increment x = x + 1

let result1 = apply1 increment 100

Значение result будет равно 101 после запуска показанного выше кода.

Несколько аргументов разделяются токенами ->, как показано в следующем примере.

let apply2 ( f: int -> int -> int) x y = f x y

let mul x y = x \* y

let result2 = apply2 mul 10 20

Результат равен 200.

*Лямбда-выражения*

Лямбда-выражение — это неименованная функция. В предыдущих примерах вместо определения именованных функций increment и mul можно было бы использовать лямбда-выражения, как показано ниже.

let result3 = apply1 (fun x -> x + 1) 100

let result4 = apply2 (fun x y -> x \* y ) 10 20

Лямбда-выражения определяются с помощью ключевого слова fun.

*Композиция функций и конвейеризация*

Функции в F# могут состоять из других функций. Композиция двух функций function1 и function2 является другой функцией, которая представляет собой применение функции function1 и последующее применение function2:

let function1 x = x + 1

let function2 x = x \* 2

let h = function1 >> function2

let result5 = h 100

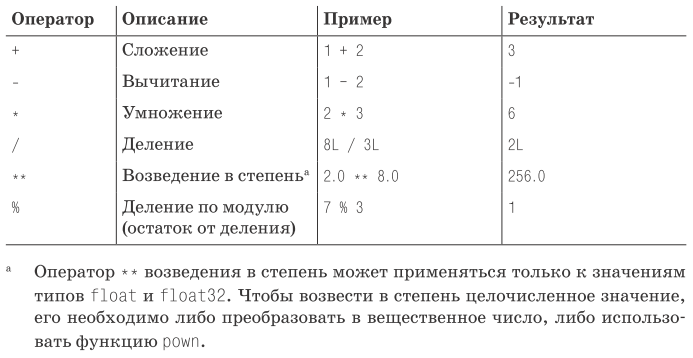
Результат равен 202.

Конвейеризация позволяет объединить вызовы функций в цепочку последовательных операций. Конвейеризация работает следующим образом.

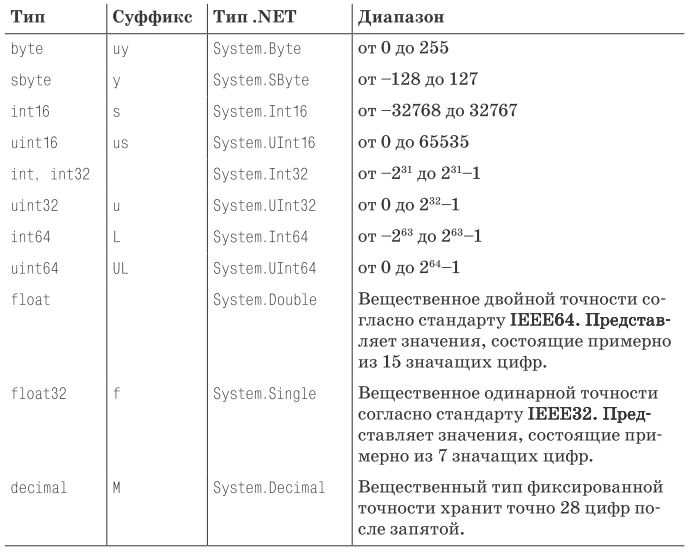
let result = 100 |> function1 |> function2

Снова будет получен результат 202.

**Арифметические операции**



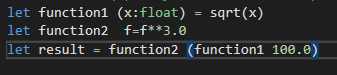
Типы данных



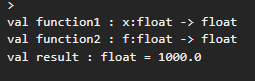
**Ход работы:**

**Задание №1.** Решите задачу принимая как аргумент функциональное значение.

Функция возводит в третью степень значение функции находящей корень от числа.



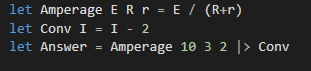
Результат выполнения:



**Задание №2. Базовые элементы.**

Решите задачу. При помощи конвейеризации уменьшите результат на 2.

Найти силу тока в полной цепи, если ЭДС источника равна 10 В, внутреннее сопротивление равно 2 Ом, а внешнее – 3 Ом.



Сила тока будет равна 2, уменьшает результат на 2. Ответ будет равен 0.

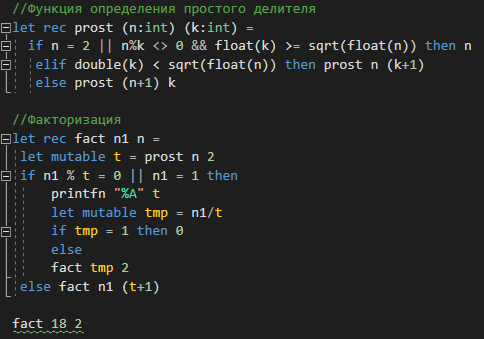
Результат выполнения:



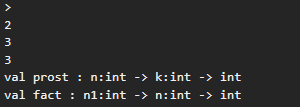
**Задание №3.** **Рекурсия.**

При решении задачи нельзя использовать циклы, строки, списки, массивы.

Дано натуральное число n>1. Выведите все простые множители этого числа.



Результат выполнения для числа 18:



18 = 2 \* 3 \* 3. Ответ верен. Программа работает корректно.

**Вывод:** ознакомился с понятием функции в F#. Научился работать с функциями на примере нескольких простых задач.

**Контрольные вопросы:**

1. Дайте определение функции.

Функции — это основной элемент выполнения программы в любом языке программирования.

1. Дайте определение понятию рекурсии.

Определение, описание, изображение какого-либо объекта или процесса внутри самого этого объекта или процесса Рекурсивные функции — функции, вызывающие самих себя.

1. Дайте определение лямбда – выражению.

Лямбда-выражение — это неименованная функция. Применяется, как правило, для объявления анонимных функций по месту их использования, и обычно допускает замыкание на лексический контекст, в котором это выражение использовано. Используя лямбда-выражения можно объявлять функции в любом месте кода.

1. Из каких элементов состоит функция.

Как и в других языках, функция в языке F# имеет имя, может иметь параметры и принимать аргументы, а также функция имеет тело.

1. Что такое токен?

Токен — минимальная единица языка, имеющая самостоятельный смысл. Существуют следующие виды токенов: имена (идентификаторы), ключевые слова, знаки операций, разделители, литералы (константы).

1. Дайте определение каррированию.

Каррирование или карринг (англ. Currying) — преобразование функции от многих аргументов в функцию, берущую свои аргументы по одному.

1. Что такое конвейеризация?

Конвейеризация позволяет объединить вызовы функций в цепочку последовательных операций. Конвейерный оператор помогает компилятору «увидеть» последний параметр функции ранее, благодаря чему механизм выведения типов может определить правильные типы функций без дополнительных аннотаций.

1. Что такое «it»?

«it» — это идентификатор, который привязывается к последнему выраженному выражению.

1. Объясните запись «int -> int -> int ->float».

Последовательность аргументов функции имеют типы int, int, int, float. Либо функция принимает 3 аргумента типа int, а результат типа float.

**Часть 2-я**

**Классы**

**Цель работы:** ознакомиться с понятием класса в языке F#. Научиться работать с классами на примере.

**Краткие теоритические сведения**

Классы представляют фундаментальное описание типов объектов .NET. Класс — основная реализация понятия типа с поддержкой объектно-ориентированного программирования в F#.

Синтаксис:

**// Class definition:**

**type [access-modifier] type-name [type-params] [access-modifier] ( parameter-list ) [ as identifier ] =**

**[ class ]**

**[ inherit base-type-name(base-constructor-args) ]**

**[ let-bindings ]**

**[ do-bindings ]**

**member-list**

**...**

**[ end ]**

**// Mutually recursive class definitions:**

**type [access-modifier] type-name1 ...**

**and [access-modifier] type-name2 ...**

**...**

Where,

* The **type-name** is any valid identifier. Default access modifier for this is **public**.
* The **type-params** describes optional generic type parameters.
* The **parameter-list** describes constructor parameters. Default access modifier for primary constructor is **public**.
* The **identifier** used with the optional **as** keyword gives a name to the instance variable, or **self-identifier,** which can be used in the type definition to refer to the instance of the type.
* The **inherit** keyword allows you to specify the base class for a class.
* The **let** bindings allow you to declare fields or function values local to the class.
* The **do-bindings** section includes code to be executed upon object construction.
* The **member-list** consists of additional constructors, instance and static method declarations, interface declarations, abstract bindings, and property and event declarations.
* The keywords **class** and **end** that mark the start and end of the definition are optional.

F# поддерживает два способа объявления классов: явный и неявный. Первый подходит в тех случаях, когда программисту требуется контроль над тем, как создается объект класса и какие поля он содержит. Второй позволяет переложить большую часть работы на компилятор.

В случае явного объявления класса программист должен объявить поля класса и хотя бы один конструктор. Использование явных конструкторов – наименее приятный способ создания экземпляров класса, но он обеспечивает наиболее полный контроль над происходящим. Когда класс определяется с явным конструктором, необходимо явно описать каждое поле класса и инициализировать его в конструкторе. Полям класса предшествует ключевое слово val. В языке C#, если вы не объявите конструктор самостоятельно, компилятор создаст конструктор по умолчанию. В языке F# дело обстоит совершенно иначе. Язык F# позволяет определять классы без конструкторов, но тогда у вас не будет возможности создавать экземпляры таких классов!

type Book =

val title : string

val author : string

val publish : int

new (t, a, pd) = {

title = t

author = a

publish = pd

}

let p = Book("Три мушкетера", "Дюма", 2000)

printfn "%A" p.title

С помощью ключевого слова val объявляются члены-данные класса, а с помощью ключевого слова new создаётся конструктор класса. Члены-данные класса обязательно должны инициализироваться в конструкторе. Если какое-либо поле не будет инициализировано, компилятор выдаст ошибку. Внутри выражения-конструктора можно только инициализировать поля класса, причем каждое поле — только один раз: попытка сделать что-нибудь ещё опять же приведет к ошибке компиляции.

Ключевые слова class и end, отмечающие начало и конец определения, можно опустить. Все члены-данные по умолчанию неизменяемы. Так же, как и в случае обычных значений, это поведение можно изменить с помощью ключевого слова mutable.

type FClass = class

val mutable private text : string

val mutable public text2 : string

public new() =

{

text = "Hello from F#"

text2 = "bla-bla-bla"

}

end

let xx = new FClass()

printfn "%A" xx.text2

xx.text2 <- "sdsd";

printfn "%A" xx.text2

В конструкторе можно не только инициализировать члены-данные, но и добавить произвольный код, который будет выполняться до инициализации.

**Ход работы:**

Приведите примеры классов заданные явным и неявным образом (не менее четырех полей, двух конструкторов и трех методов). Продемонстрируйте работу с ними.

Класс кассовый чек кафе. Содержит информацию о номере чека, клиенте, официанте, стоимость заказов, скидку и итоговую сумму по чеку.

Явный метод:

type CashiersCheck =

val id : int

val client : string

val staff : string

val discount : float

val cost : float

val summ : float

new() = {

id = 0

client = "Client"

staff = "Oficiant"

discount = 1.0

cost = 0.0

summ = 0.0

}

new (Id, Client, Staff, Discount, Cost) = {

id = Id

client = Client

staff = Staff

discount = Discount

cost = Cost

summ = Cost \* Discount

}

member this.IdToStr = sprintf "%i" (this.id)

member this.DiscountToStr = sprintf "%.2f" ((1.0 - this.discount) \* 100.0)

member this.CostToStr = sprintf "%.2f" (this.cost)

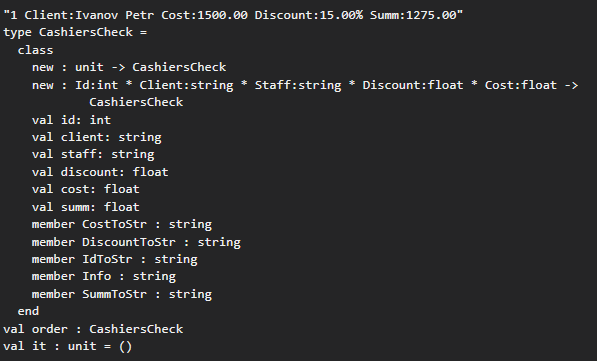
member this.SummToStr = sprintf "%.2f" (this.summ)

member this.Info = this.IdToStr + " Client:" + this.client + " Cost:"+ this.CostToStr + " Discount:" + this.DiscountToStr + "% Summ:" + this.SummToStr

Пример работы (выводит информацию о заказе):

let order = CashiersCheck(1, "Ivanov Petr", "Golovina Ekaterina", 0.85, 1500.0)

printfn "%A" order.Info



Неявный метод:

Данные заполняются из строки разделенной «;»

type CashiersCheck1(Id:int, Client:string, Staff:string, Cost:float, Discount:float) =

member this.id = Id

member this.client = Client

member this.staff = Staff

member this.cost = Cost

member this.discount = Discount

member this.summ = Cost \* Discount

new () = new CashiersCheck1(0, "", "", 0.0, 0.0)

new (str:string) =

let r = str.Split(';')

let Id = (r.[0]) |> int

let Client = r.[1]

let Staff = r.[2]

let Cost = (r.[3])|> float

let Discount = (r.[4])|> float

new CashiersCheck1(Id, Client, Staff, Cost, Discount)

let order2 = new CashiersCheck1(2,"Ivan Zaykov","Ilya Degterev",1300.0, 0.9)

let order3 = new CashiersCheck1("3;Elena Kruglova;Ekaterina Golovina;850.0;0.95");

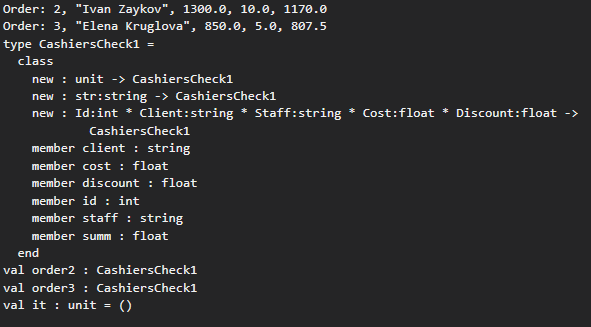
printfn "Order: %A, %A, %A, %A, %A"

order2.id order2.client order2.cost ((1.0 - order2.discount) \* 100.0) order2.summ

printfn "Order: %A, %A, %A, %A, %A"

order3.id order3.client order3.cost ((1.0 - order3.discount) \* 100.0) order3.summ

Результат выполнения:



**Вывод:** ознакомился с понятием класса в F#. Научился работать с ними на примере.