**Варианты 2018**

**Лямбда-выражения**

**С#**

**Выражения-лямбды**

Лямбда-выражение с выражением с правой стороны оператора => называется выражением-лямбдой. Выражения-лямбды широко используются при конструировании деревьев выражений. Выражения-лямбды возвращают результат выражения и принимают следующую основную форму.

(input-parameters) => expression

Если лямбда-выражение имеет только один входной параметр, скобки можно не ставить; во всех остальных случаях они обязательны. Два и более входных параметра разделяются запятыми и заключаются в скобки:

(x, y) => x == y

Иногда компилятору бывает трудно или даже невозможно определить входные типы. В этом случае типы можно указать в явном виде, как показано в следующем примере.

(int x, string s) => s.Length > x

Для входных параметров нужно задать либо все типы в явном виде, либо все типы в неявном. В противном случае компилятор C# выдает ошибку CS0748.

Нулевое количество входных параметров задается пустыми скобками:

() => SomeMethod()

Обратите внимание, что тело выражения-лямбды может состоять из вызова метода, как было показано в предыдущем примере.

**Лямбды операторов**

Лямбда оператора напоминает выражение-лямбду, за исключением того, что оператор (или операторы) заключается в фигурные скобки:

(входные-параметры) => { оператор; }

Тело лямбды оператора может состоять из любого количества операторов; однако на практике обычно используется не более двух-трех.

delegate void TestDelegate(string s);

TestDelegate del = n => { string s = n + " World";

Console.WriteLine(s); };

Лямбды операторов, как и анонимные методы, не могут использоваться для создания деревьев выражений.

Делегаты в C# подобны функциональным указателям в C или C++. Они объявляются в качестве объектов, содержащих адрес метода. Делегаты — референсный класс, инкапсулирующий метод со специфичной сигнатурой и типом возвращаемого значения.

**Делегаты**

Несмотря на то, что .NET использует концепцию функционального указателя посредством делегатов, есть несколько существенных отличий:

* делегаты нечувствительны к ошибкам ввода;
* объектно-ориентированы;
* безопасны.

Делегаты C# обладают следующими свойствами:

* позволяют обрабатывать методы в качестве аргумента;
* могут быть связаны вместе;
* несколько методов могут быть вызваны по одному событию;
* тип делегата определяется его именем;
* не зависят от класса объекта, на который ссылается;
* сигнатура метода должна совпадать с сигнатурой делегата.

**Синтаксис**

**Шаг 1 — Объявление**

modifier delegate ReturnType DelegateName ([Parameter\_1]);

**Шаг 2 — Инициализация**

Делегат инициализируется путём передачи ему имени метода в качестве аргумента.

DelegateName DelgObjectName = new DelegateName(MethodName);

**Шаг 3 — Вызов**

Вызываем созданный делегат с указанием параметров, если это необходимо.

DelegateObjectName([Parameter\_1]);

**Пример**

Нижеприведённый код складывает между собой два числа:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace DelgateForNotes

{

class Program

{

// Объявление делегата, ссылающегося на функцию

// с двумя параметрами и целочисленным результатом

public delegate int AddDelegate(int num1, int num2);

static void Main(string[] args)

{

// Создание метода делегата и передача функции Add в качестве аргумента

AddDelegate funct1 = new AddDelegate(Add);

// Вызов делегата

int k = funct1(7, 2);

Console.WriteLine("Sumation = {0}", k);

Console.Read();

}

// Статическая функция Add с той же сигнатурой, что и у делегата

public static int Add(int num1, int num2)

{

Console.WriteLine("I am called by Delegate");

int sumation;

sumation = num1 + num2;

return sumation;

}

}

}

Вывод:

I am called by Delegate

Sumation = 9

**C++**

**Синтаксис**

[ capture ] ( params ) mutable exception attribute -> ret { body } (1)

[ capture ] ( params ) -> ret { body } (2)

[ capture ] ( params ) { body } (3)

[ capture ] { body } (4)

1) Полное объявление

2) Объявление константной лямбды: объекты, захваченные копированием, не могут быть изменены.

3) Опущен возвращаемый тип в конце: возвращаемый тип замыкания operator() выводится в соответствии со следующими правилами:

* если body состоит из одного оператора return, то возвращаемый тип является типом возвращаемого выражения (после неявного преобразования rvalue к lvalue, массива к указателю, или функции к указателю)
* В противном случае возвращаемый тип void

4) Пропущен список параметров: функция не принимает аргументов, как если бы список параметров был ()

**Объяснение**

* mutable — позволяет body изменять параметры, захваченные копированием, и вызывать их неконстантные функции-члены
* exception — обеспечивает спецификацию исключения или опцию noexcept для оператора () типа-замыкания
* attribute — обеспечивает спецификацию атрибута для оператора () типа замыкания
* capture — определяет, какие символы, видимые в области объявления функции, ​​будут видны внутри тела функции.
* Список символов может быть передан следующим образом:
* [a,&b] где a захвачена по значению, а b захвачена по ссылке.
* [this] захватывает указатель this по значению.
* [&] захват всех символов по ссылке
* [=] захват всех символов по значению
* [] ничего не захватывает
* params — Список параметров, как в объявлении функции
* ret — Возвращаемый тип. Если нет, то он выводится из возвращаемого значения (или void, если функция не возвращает никакого значения)

**Особенности длины строки от языка(С++, Java, Go, Swift)**

Пусть s описана как переменная строкового типа (string), и в программе есть присваивание s= ”кафе” (все буквы - кириллические). Какова будет длина этой строки в языках C++ (s.size()), Java (s.length), Go (len(s)), Swift (s.count)? Для каждого языка дайте ответ и объясните различия (если они есть).

**Ответ**: С++ - не определено и зависит от кодировки исходного текста, Java,

C#,Swift – 4, Go – 8.

**Вариативный шаблон в C++** был разработан Дугласом Грегором и Яакко Ярви [[1]](https://ru.wikipedia.org/wiki/%D0%92%D0%B0%D1%80%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD#cite_note-1)[[2]](https://ru.wikipedia.org/wiki/%D0%92%D0%B0%D1%80%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD#cite_note-2) и был позже стандартизирован в C++11. До появления C++11, шаблоны (классов и функций) могли принимать только фиксированное число аргументов, которые должны были быть определены, когда шаблон был впервые объявлен.

Синтаксис вариативного шаблона:

**template**<**typename**... Values> **class** **tuple**;

Приведенный выше шаблон класса tuple (от англ. кортеж) может принять любое число входных параметров. Например, экземпляр вышеуказанного шаблонного класса создают с тремя аргументами:

tuple<int, std::vector<int>, std::map<<std::string>, std::vector<int>>> some\_instance\_name;

Число аргументов может быть равно нулю, поэтому tuple<> some\_instance\_name; будет также работать. Если вы не хотите давать возможности создавать вариативные шаблонные объекты с нулём аргументов, можно использовать следующее объявление:

**template**<**typename** First, **typename**... Rest> **class** **tuple**;

Вариативные шаблоны могут также применяться к функциям.

**template**<**typename**... Params> void printf(**const** std::string &str\_format, Params... parameters);

Оператор многоточия (...)  играет две роли. Когда он стоит слева от имени параметра функции, он объявляет набор параметров. Когда оператор многоточия стоит справа от шаблона или аргумента вызова функции, он распаковывает параметры в отдельные аргументы, также как и args... в теле printf ниже.

Нет простого механизма перебора значений вариативных шаблонов. Есть несколько способов конвертации списка аргументов в один аргумент. Обычно это реализуется с помощью перегрузки функций, или, если функция может принять только один аргумента за раз — с помощью простого маркера расширения:

**template**<**typename**... Args> **inline** void pass(Args&&...) {}

этот шаблон можно использовать следующим образом:

**template**<**typename**... Args> **inline** void expand(Args&&... args) {

pass( some\_function(args)... );

}

expand(42, "answer", true);

и он будет преобразован во что-то вроде:

pass( some\_function(arg1), some\_function(arg2), some\_function(arg3) и т. д. ... );

Использование функции "pass" необходимо, поскольку распаковка аргументов происходит путём разделения аргументов функции через запятую, которые не эквивалентны оператору запятая. Поэтому some\_function(args)...; никогда не будет работать. Кроме того, решение выше будет работать только тогда, когда возвращаемый тип some\_function не void. Кроме того, вызовы some\_function будут выполняться в произвольном порядке, потому что порядок вычисления аргументов функции не определён. Чтобы избежать произвольного порядка, может быть использован список инициализации в скобках, гарантирующий соблюдение последовательности слева-направо.

**struct** pass {

**template**<**typename** ...T> pass(T...) {}

};

pass{(some\_function(args), 1)...};

**Обобщённые классы**

**Java**

class Gen<T> {

T ob; // объявление объекта типа T

Gen(T o) { // Передать конструктору ссылку на объект типа T

ob = o;

}

T getob() { // Вернуть ob

return ob;

}

void showType() { // Показать тип T

System.out.println("Тип T: " + ob.getClass().getName());

}

}

// Работаем с обобщённым классом

Gen<Integer> iOb; // Создаём Gen-ссылку для Integer

iOb = new Gen<Integer>(77); // Создаём объект Gen<Integer>

iOb.showType(); // Показать тип данных, используемый iOb

int value = iOb.getob(); // Получить значение iOb

System.out.println("Значение " + value);

Gen<String> strOb = new Gen<String>("Обобщённый текст"); // Создадим объект Gen для String

strOb.showType(); // Показать тип данных, используемый strOb

String str = strOb.getob(); // Получить значение strOb

System.out.println("Значение: " + str);

В результате мы получим:

Типом T является java.lang.Integer

Значение: 77

Типом T является java.lang.String

Значение: Обобщённый текст

Обобщённый класс с двумя параметрами

class TwoGen<T, V> {

T ob1;

V ob2;

TwoGen(T o1, V o2) { // Передать конструктору ссылки на объекты двух типов

ob1 = o1;

ob2 = o2;

}

void showTypes() {

System.out.println("Тип T: " + ob1.getClass().getName());

System.out.println("Тип V: " + ob2.getClass().getName());

}

T getob1() { return ob1; }

V getob2() { return ob2; }

// Используем созданный класс

TwoGen<Integer, String> twogenObj = new TwoGen<Integer, String>(77, "Обобщённый текст");

twogenObj.showTypes(); // Узнаем типы

int value = twogenObj.getob1(); // Узнаем значения

System.out.println("Значение: " + value);

String str = twogenObj.getob2();

System.out.println("Значение: " + str);

}

**C#**

Обобщенные типы позволяют указать конкретный тип, который будет использоваться. Поэтому определим класс Account как обощенный:

class Account<T>

{

public T Id { get; set; }

public int Sum { get; set; }

}

Угловые скобки в описании class Account<T> указывают, что класс является обобщенным, а тип T, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву T, это может быть и любая другая буква или набор символов. Причем сейчас нам неизвестно, что это будет за тип, это может быть любой тип. Поэтому параметр T в угловых скобках в еще называется универсальным параметром, так как вместо него можно подставить любой тип.

Например, вместо параметра T можно использовать объект int, то есть число, представляющее номер счета. Это также может быть объект string, либо или любой другой класс или структура.

Поскольку класс Account является обобщенным, то при определении переменной после названия типа в угловых скобках необходимо указать тот тип, который будет использоваться вместо универсального параметра T. В данном случае объекты Account типизируется типами int и string:

Account<int> account1 = new Account<int> { Sum = 5000 };

Account<string> account2 = new Account<string> { Sum = 4000 };

Поэтому у первого объекта account1 свойство Id будет иметь тип int, а у объекта account2 - тип string.

При попытке присвоить значение свойства Id переменной другого типа мы получим ошибку компиляции:

Account<string> account2 = new Account<string> { Sum = 4000 };

account2.Id = "4356";

int id1 = account2.Id; // ошибка компиляции

Тем самым мы избежим проблем с типобезопасностью. Таким образом, используя обобщенный вариант класса, мы снижаем время на выполнение и количество потенциальных ошибок.

**Значения по умолчанию**

Иногда возникает необходимость присвоить переменным универсальных параметров некоторое начальное значение, в том числе и null. Но напрямую мы его присвоить не можем:

T id = null;

В этом случае нам надо использовать оператор default(T). Он присваивает ссылочным типам в качестве значения null, а типам значений - значение 0:

class Account<T> {

T id = default(T); }

**Статические поля обобщенных классов**

При типизации обобщенного класса определенным типом будет создаваться свой набор статических членов. Например, в классе Account определено следующее статическое поле:

class Account<T> {

public static T session;

public T Id { get; set; }

public int Sum { get; set; } }

Теперь типизируем класс двумя типами int и string:

Account<int> account1 = new Account<int> { Sum = 5000 };

Account<int>.session = 5436;

**Генераторы Python**

Функция, возвращающая подвид итератора, генерирующий значения.

На заметку

Словом «генератор» обычно обозначается функция-генератор (или метод-генератор), возвращающая итератор генератора. Однако иногда слово может быть использовано и для обозначения самого итератора. В случаях, когда контекст непонятен лучше использовать полные термины: функция-генератор и итератор генератора. Итератор генератора — это объект, порождаемый функцией-генератором.

На заметку

Генераторы являются простым средством для создания итераторов. Всё, что можно сделать при помощи генераторов можно также сделать при помощи итераторов, построенных на классах. Но в случае генераторов методы \_\_iter\_\_() и \_\_next\_\_() создаются автоматически, также автоматически возбуждается StopIteration, да и поддерживать генераторы проще и удобнее, чем реализовывать то же с использованием классов.

Выглядят функции-генераторы также как и обычные, но содержат выражения с ключевым словом yield для последовательного генерирования значений, которые могут быть использованы в циклах for in, либо их получения при помощи функции next().

На каждой yield работа функции временно приостанавливается, при этом сохраняется состояние исполнения, включая локальные переменные, указатель на текущую инструкцию, внутренний стек и состояние обработки исключения. При последующем обращении к итератору генератора (при вызовах его методов) функция продолжает своё исполнение с места, на котором была приостановлена. Этим функции-генераторы отличаются от обычных функций, при вызове которых исполнение всякий раз начинается с начала.

Если функция достигает инструкции return, либо конца (без указания упомянутой инструкции), возбуждается исключение StopIteration и итератор исчерпывает себя.

def my\_animal\_generator():

yield 'корова'

for animal in ['кот', 'собака', 'медведь']:

yield animal

yield 'кит'

for animal in my\_animal\_generator():

print(animal) # корова кот собака медведь кит

На заметку

Функции-генераторы весьма похожи на сопрограммы: могут выдывать значения несколько раз, имеют более одной точки входа, их выполнение может быть приостановлено. Единственным различием является то, что функции-генераторы не могут определять то, как должно продолжаться исполнение (что должно происходить) после выдачи значения — управление всегда передаётся коду, вызвавшему генератор.

Инструкция yield может употребляться и в конструкции try except. Если к генератору не обратились до его финализации (финализация происходит, когда счётчик ссылок доходит до нуля, либо когда происходит сборка мусора), будет вызван метод итератора .close(), что позволяет выполнить оставшиеся в блоке finally инструкции.

**Генераторы**

* Главная их особенность — это сохранение состояния между вызовами.

Пример:

def read\_file\_line\_by\_line(file\_name):

with open(file\_name, 'r') as f:

while True:

line = f.readline()

if not line:

break

yield line

Эта функция принимает на вход имя файла и возвращает его строчка за строчкой, не загружая целиком в память, что может быть необходимо при чтении больших файлов. Такой приём называют ленивым (lazy) чтением, подразумевая, что мы не делаем «работу» без необходимости.

В общем случае, работа с генераторами выглядит следующим образом:

In [78]: lines\_generator = read\_file\_line\_by\_line("data.csv")

In [79]: type(lines\_generator)

Out[79]: generator

In [83]: lines\_generator.next()

Out[83]: 'time,host,event\n'

In [84]: lines\_generator.next()

Out[84]: '1374039728,localhost,reboot\n'

In [85]: lines\_generator.next()

Out[85]: '1374039730,localhost,start\n'

In [86]: lines\_generator.next()

---------------------------------------------------------------------------

StopIteration Traceback (most recent call last)

<ipython-input-86-65df1a2cb71b> in <module>()

----> 1 lines\_generator.next()

StopIteration:

# Соответственно у меня в файле только 3 строчки

# Как только читать больше нечего, возникает исключение StopIteration, как и с любым итерируемым объектом.

Естественно, чаще мы читаем значения из генератора в цикле, а не построчно.

uniq = []

for line in lines\_generator:

if line not in uniq:

uniq.append(line)

Так же возможна короткая запись генератора:

In [92]: gen = (x for x in xrange(0, 100\*10000))

In [93]: gen.next()

Out[93]: 0

In [94]: gen.next()

Out[94]: 1

In [95]: gen.next()

Out[95]: 2

In [96]: gen.next()

Out[96]: 3

In [97]: gen.next()

Out[97]: 4

Не требует создания всего списка range(0, 100\*10000) в памяти, возвращаемое значение вычисляется» каждый раз при обращении.

**Сопрограммы** как частный случай генераторов

Генератор может не только возвращать значения, но и принимать их на вход.

Напишем простую реализацию генератора, который может складывать два аргумента, хранить историю результатов и выводить историю.

def calc():

history = []

while True:

x, y = (yield)

if x == 'h':

print history

continue

result = x + y

print result

history.append(result)

c = calc()

print type(c) # <type 'generator'>

c.next() # Необходимая инициация. Можно написать c.send(None)

c.send((1,2)) # Выведет 3

c.send((100, 30)) # Выведет 130

c.send((666, 0)) # Выведет 666

c.send(('h',0)) # Выведет [3, 130, 666]

c.close() # Закрывем генератор

Т.е. мы создали генератор, проинициализировали его и подаём ему входные данные.

Он, в свою очередь, эти данные обрабатывает и сохраняет своё состояние между вызовами до тех пор пока мы его не закрыли. После каждого вызова генератор возвращает управление туда, откуда его вызвали. Это важнейшее свойство генераторов мы и будем использовать.

**2016**

**Виртуа́льное насле́дование** в языке программирования C++ — один из вариантов наследования, который нужен для решения некоторых проблем, порождаемых наличием возможности множественного наследования (особенно «ромбовидного наследования»), путём разрешения неоднозначности того, методы которого из суперклассов (непосредственных классов-предков) необходимо использовать. Оно применяется в тех случаях, когда множественное наследование вместо предполагаемой полной композиции свойств классов-предков приводит к ограничению доступных наследуемых свойств вследствие неоднозначности. Базовый класс, наследуемый множественно, определяется виртуальным с помощью ключевого слова virtual.

**Представление класса:**

Прежде чем продолжить, полезным будет рассмотреть, как классы представляются в C++. В частности, при наследовании классы предка и наследника просто помещаются в памяти друг за другом. Таким образом объект класса Bat это на самом деле последовательность объектов классов (Animal,Mammal,Animal,WingedAnimal,Bat), размещенных последовательно в памяти, при этом Animal повторяется дважды, что и приводит к неоднозначности.

Виртуальное наследование реализуется через добавление указателей в классы Mammal и WingedAnimal. Таким образом, Bat представляется, как (ptr, Mammal, ptr, WingedAnimal, Bat, Animal). \*ptr содержит информацию о смещении в памяти между началом Mammal и его Animal. Это необходимо, потому что для указателя Animal\* p;, который может ссылаться на Animal, на Mammal, на WingedAnimal, смещение в памяти между началом объекта и его Animal части неизвестно на этапе компиляции, а выясняется только во время выполнения.

#include <iostream>

class A {

public:

int foo() { return 1; }

};

class B : public virtual A {};

class C : public virtual A {};

class D : public B, public C {};

int main () {

D d;

std::cout << d.foo();

return 0;

}

Если убрать ключевое слово virtual, то метод foo() не может быть определён однозначно и в результате не будет доступен как объект класса D и код не скомпилируется.

**Ромбовидное наследование** — ситуация в объектно-ориентированных языках программирования с поддержкой множественного наследования, когда два класса B и C наследуют от A, а класс D наследует от обоих классов B и C. При этой схеме наследования может возникнуть неоднозначность: если метод класса D вызывает метод, определенный в классе A (и этот метод не был переопределен в классе D), а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: B или C?

**Интерфейс Java**

Механизм наследования очень удобен, но он имеет свои ограничения. В частности мы можем наследовать только от одного класса, в отличие, например, от языка С++, где имеется множественное наследование.

В языке Java подобную проблему частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.

Чтобы определить интерфейс, используется ключевое слово interface.

Например:

interface Printable{

void print();

}

Данный интерфейс называется Printable. Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов. Так, в данном случае объявлен один метод, который не имеет реализации.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Чтобы класс применил интерфейс, надо использовать ключевое слово implements.

В данном случае класс Book реализует интерфейс Printable. При этом надо учитывать, что если класс применяет интерфейс, то он должен реализовать все методы интерфейса, как в случае выше реализован метод print. Потом в методе main мы можем объект класса Book и вызвать его метод print. Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут эти методы.

В тоже время мы не можем напрямую создавать объекты интерфейсов, поэтому следующий код не будет работать:

Printable pr = new Printable();

pr.print();

**C# Интерфейсы**

Интерфейс представляет ссылочный тип, который определяет набор методов и свойств, но не реализует их. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

Для определения интерфейса используется ключевое слово interface. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования. Например, интерфейс IMovable:

interface IMovable

{

void Move();

}

У интерфейса методы и свойства не имеют реализации, в этом они сближаются с абстрактными методами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не принимает никаких параметров и ничего не возвращает.

Еще один момент в объявлении интерфейса: все его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

В целом интерфейсы могут определять следующие сущности:

* Методы
* Свойства
* Индексаторы
* События

Однако интерфейсы не могут определять статические члены, переменные, константы.

Затем какой-нибудь класс или структура могут применить данный интерфейс:

class Person : IMovable // применение интерфейса в классе

{

public void Move() {

Console.WriteLine("Человек идет"); }

}

struct Car : IMovable // применение интерфейса в структуре

{

public void Move() {

Console.WriteLine("Машина едет"); }

}

При применени интерфейса, как и при наследовании после имени класса или структуры указывается двоеточие и затем идут названия применяемых интерфейсов. При этом класс должен реализовать все методы и свойства применяемых интерфейсов. При этом поскольку все методы и свойства интерфейса являются публичными, при реализации этих методов и свойств в классе к ним можно применять только модификатор public. Поэтому если класс должен иметь метод с каким-то другим модификатором, например, protected, то интерфейс не подходит для определения подобного метода.

Если класс или структура не реализуют какие-либо свойства или методы интерфейса, то мы столкнемся с ошибкой на этапе компиляции.

Применение интерфейса в программе:

using System;

namespace HelloApp

{

interface IMovable {

void Move(); }

class Person : IMovable

{

public void Move() {

Console.WriteLine("Человек идет"); }

}

struct Car : IMovable

{

public void Move() {

Console.WriteLine("Машина едет"); }

}

class Program

{

static void Action(IMovable movable) {

movable.Move(); }

static void Main(string[] args)

{

Person person = new Person();

Car car = new Car();

Action(person);

Action(car);

Console.Read();

}

}

}

В данной программе определен метод Action(), который в качестве параметра принимает объект интерфейса IMovable. На момент написания кода мы можем не знать, что это будет за объект - какой-то класс или структура. Единственное, в чем мы можем быть уверены, что этот объект обязательно реализует метод Move и мы можем вызвать этот метод.

Иными словами, интерфейс - это контракт, что какой-то определенный тип обязательно реализует некоторый функционал.

Вывод данной программы:

Человек идет

Машина едет

Если класс применяет интерфейс, то этот класс должен реализовать все методы и свойства интерфейса. Однако также можно и не реализовать методы, сделав их абстрактными, переложив право их реализации на производные классы:

interface IMovable {

void Move(); }

abstract class Person : IMovable {

public abstract void Move(); }

class Driver : Person {

public override void Move() {

Console.WriteLine("Шофер ведет машину"); }

}

**С# Пространства имен**

Пространства имен (namespace) — это способ, благодаря которому .NET избегает конфликтов имен между классами. Они предназначены для того, чтобы исключить ситуации, когда вы определяете класс, представляющий заказчика, называете его Customer, а после этого кто-то другой делает то же самое (подобный сценарий достаточно распространен).

Пространство имен — это не более чем группа типов данных, но дающая тот эффект, что имена всех типов данных в пределах пространства имен автоматически снабжаются префиксом - названием пространства имен. Пространства имен можно вкладывать друг в друга. Например, большинство базовых классов .NET общего назначения находятся в пространстве имен System. Базовый класс Array относится к этому пространству, поэтому его полное имя — System.Array.

Платформа .NET требует, чтобы все имена были объявлены в пределах пространства имен; например, вы можете поместить свой класс MyClass в пространство имен MyCompany. Тогда полное имя этого класса будет выглядеть как MyCompany.MyClass.

При изучении перечня, приведенного в таблице, нетрудно было заметить, что пространство имен System является корневым для приличного количества вложенных пространств имен (таких как System.IO, System.Data и т.д.). Как оказывается, однако, помимо System в библиотеке базовых классов предлагается еще и ряд других корневых пространств имен наивысшего уровня, наиболее полезным из которых является пространство имен Microsoft.

**Способы описания и использования**

Все определяемые классы не существуют сами по себе, а, как правило, заключаются в специальные контейнеры - пространства имен. Создаваемый по умолчанию класс Program уже находится в пространстве имен, которое обычно совпадает с названием проекта:

namespace HelloApp

{ class Program

{ static void Main(string[] args) {} }

}

Пространство имен определяется с помощью ключевого слова namespace, после которого идет название. Так в данном случае полное название класса Program будет HelloApp.Program.

Класс Program видит все классы, которые объявлены в том же пространстве имен:

namespace HelloApp

{ class Program {

static void Main(string[] args)

{ Account account = new Account(4); }

}

class Account {

public int Id { get; private set;} // номер счета

public Account(int \_id) { Id = \_id; }

}

}

Но чтобы задействовать классы из других пространств имен, эти пространства надо подключить с помощью директивы using:

using System;

namespace HelloApp

{ class Program

{ static void Main(string[] args) { Console.WriteLine("hello"); }

}

}

Здесь подключается пространство имен System, в котором определен класс Console. Иначе нам бы пришлось писать полный путь к классу:

static void Main(string[] args) {

System.Console.WriteLine("hello"); }

Пространства имен могут быть определены внутри других пространств:

using HelloApp.AccountSpace;

namespace HelloApp {

class Program {

static void Main(string[] args) { Account account = new Account(4); }

}

namespace AccountSpace {

class Account

{

public int Id { get; private set;}

public Account(int \_id) { Id = \_id; }

}

}

}

В этом случае для подключения пространства указывается его полный путь с учетом внешних пространств имен: using HelloApp.AccountSpace;

**Псевдонимы**

Для различных классов мы можем использовать псевдонимы. Затем в программе вместо названия класса используется его псевдоним. Например, для вывода строки на экран применяется метод Console.WriteLine(). Но теперь зададим для класса Console псевдоним:

using printer = System.Console;

class Program {

static void Main(string[] args) {

printer.WriteLine("Hello from C#");

printer.Read(); }

}

С помощью выражения using printer = System.Console указываем, что псевдонимом для класса System.Console будет имя printer. Это выражение не имеет ничего общего с подключением пространств имен в начале файла, хотя и использует оператор using. При этом используется полное имя класса с учетом пространства имен, в котором класс определен. И далее, чтобы вывести строку, применяется выражение printer.WriteLine("Hello from C#").

**Java пакеты**

Как правило, в Java классы объединяются в пакеты. Пакеты позволяют организовать классы логически в наборы. По умолчанию java уже имеет ряд встроенных пакетов, например, java.lang, java.util, java.io и т.д. Кроме того, пакеты могут иметь вложенные пакеты.

Организация классов в виде пакетов позволяет избежать конфликта имен между классами. Ведь нередки ситуации, когда разработчики называют свои классы одинаковыми именами. Принадлежность к пакету позволяет гарантировать однозначность имен.

Чтобы указать, что класс принадлежит определенному пакету, надо использовать директиву package, после которой указывается имя пакета:

package название\_пакета;

Как правило, названия пакетов соответствуют физической структуре проекта, то есть организации каталогов, в которых находятся файлы с исходным кодом. А путь к файлам внутри проекта соответствует названию пакета этих файлов. Например, если классы принадлежат пакету mypack, то эти классы помещаются в проекте в папку mypack.

Классы необязательно определять в пакеты. Если для класса пакет не определен, то считается, что данный класс находится в пакете по умолчанию, который не имеет имени.

**Java package** — механизм, позволяющий организовать Java классы в пространстве имен аналогично модулям в языке программирования Модула.

Java пакеты могут содержаться в сжатом виде в JAR файлах. Обычно в пакеты объединяют классы одной и той же категории, либо предоставляющие сходную функциональность.

* Каждый пакет предоставляет уникальное пространство имен для своего содержимого.
* Допустимы вложенные пакеты.

Классы, определенные без явно заданных модификаторов доступа (public, protected, private), видимы только внутри пакета.

Класс Пакет Подклассы Все

private Да — — —

(без) Да Да — —

protected Да Да Да

public Да Да Да Да

* private — доступ только внутри класса (наиболее рекомендуемый)
* (без модификатора) — только внутри пакета (по умолчанию)
* protected — межпакетный доступ только для подклассов
* public — межпакетный доступ (наименее рекомендуемый)

**Различия (между пространствами имен в С# и пакетах на Java)**

**Java**

Пакеты используются для организации файлов или общедоступных типов, чтобы избежать конфликтов типов. Конструкции пакетов могут быть сопоставлены с файловой системой.

System.Security.Cryptography.AsymmetricAlgorithm aa;

может быть заменен:

import System.Security.Crypography;

class xxx { ...

AsymmetricAlgorithm aa;

Нет псевдонимов для пакетов. Вы должны использовать оператор импорта или полное имя, чтобы указать конкретный тип.

package N1.N2;

class A {}

class B {}

или

package N1.N2;

class A {}

Другой исходный файл:

package N1.N2;

class B {}

Пакет не может быть вложен. Один исходный файл может иметь только один оператор пакета.

**С#**

Пространства имен используются для организации программ, как для "внутренней" системы организации для программы, так и для "внешней" организации.

System.Security.Cryptography.AsymmetricAlgorithm aa;

может быть заменен:

using System.Security.Crypography;

AsymmetricAlgorithm aa;

В качестве альтернативы можно указать псевдоним для пространства имен, например

using myAlias = System.Security.Crypography;

а затем обратитесь к классу с

myAlias.AsymmetricAlgorithm

namespace N1.N2

{ class A {}

class B {} }

или

namespace N1

{

namespace N2

{ class A {}

class B {} }

}

**2015**

**C# override**

Модификатор override требуется для расширения или изменения абстрактной или виртуальной реализации унаследованного метода, свойства, индексатора или события.

Метод override предоставляет новую реализацию члена, унаследованного от базового класса. Метод, переопределенный объявлением override, называется переопределенным базовым методом. Переопределенный базовый метод должен иметь ту же сигнатуру, что и метод override. Дополнительные сведения о наследовании см. в разделе [Наследование](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/inheritance).

Невиртуальный или статический метод нельзя переопределить.Переопределенный базовый метод должен иметь тип virtual, abstractили override.

Объявление override не может изменить доступность метода virtual.Методы override и virtual должны иметь одинаковый [модификатор уровня доступа](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/access-modifiers).

Модификаторы new, static и virtual нельзя использовать для изменения метода override.

Переопределяющее объявление свойства должно задавать такие же модификатор уровня доступа, тип и имя, которые имеются у унаследованного свойства, а переопределенное свойство должно иметь тип virtual, abstract или override.

**2017**

**Java Внутренние, вложенные, анонимные классы**

Определение класса может размещаться внутри определения другого класса. Такие классы называются вложенными или внутренними. Область видимости вложенного класса ограничена областью видимости внешнего класса. Поэтому, если вы создадите класс B внутри класса A, то класс B не сможет существовать независимо от класса A.

Вложенные классы позволяют группировать классы, логически принадлежащие друг другу, и управлять доступом к ним.

Существуют два типа вложенных класса - статические и нестатические.

**Вложенные классы**

Если связь между объектом внутреннего класса и объектом внешнего класса не нужна, можно сделать внутренний класс статическим (static). Такой класс называют вложенным (nested).

Применение статического внутреннего класса означает следующее:

* для создания объекта статического внутреннего класса не нужен объект внешнего класса
* из объекта вложенного класса нельзя обращаться к нестатическим членам внешнего класса

Вложенный класс имеет доступ к членам своего внешнего класса, в том числе и к закрытым членам. Однако, внешний класс не имеет доступа к членам вложенного класса. Вложенный класс при этом является членом внешнего класса.

Статический класс объявляется ключевым словом static. При этом класс должен обращаться к нестатическим членам своего внешнего класса при помощи объекта, т.е. он не может обращаться напрямую на нестатические члены своего внешнего класса. На практике подобные классы используются редко.

// внешний класс

class outerClassName {

private static class innerClassName { // тело вложенного класса }

}

**Внутренние классы**

Нестатические вложенные классы называют также внутренними классами (inner). Внутренний класс имеет доступ ко всем переменным и методам своего внешнего класса и может непосредственно ссылаться на них.

Внутренние классы создаются внутри окружающего класса:

// внешний класс

class Outer {

int outer\_x = 9;

void test() {

Inner inner = new Inner();

inner.display(); }

// внутренний класс

class Inner {

void display() {

Log.i(TAG, outer\_x); }

}

}

class MainActivity...{

// В методе onCreate() активности

Outer outer = new Outer();

outer.test();

}

Внутренний класс Inner определён в области видимости класса Outer. Поэтому любой код в классе Inner может непосредственно обращаться к переменной outer\_x. Когда мы создаём экземпляр класса Outer и вызываем метод test(), то создаём также экземпляр класса Inner с вызовом метода display().

Кстати говоря, внутренний класс можно определить не только на уровне класса, но и внутри метода или внутри тела цикла.

Если понадобится создать объект внутреннего класса не в статическом методе внешнего класса, тип этого объекта должен задаваться в формате ИмяВнешнегоКласса.ИмяВнутреннегоКласса.

Объект внутреннего класса связан с внешним объектом-создателем и может обращаться к его членам без каких-либо дополнительных описаний. Для внутренних классов доступны все элементы внешнего класса.

Если вам понадобится получить ссылку на объект внешнего класса, запишите имя внешнего класса, за которым следует точка, а затем ключевое слово this.

**Анонимные классы**

Существует разновидность внутреннего класса, которая называется анонимным классом, так как у него нет имени. Подобные классы очень часто встречаются в примерах на Android. Например, когда вы пишете код для щелчка или других событий.

seekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {

@Override

public void onProgressChanged(SeekBar seekBar, int i, boolean b) {}

@Override

public void onStartTrackingTouch(SeekBar seekBar) {}

@Override

public void onStopTrackingTouch(SeekBar seekBar) {}

});

В этом коде вы используете конструкцию new SeekBar.OnSeekBarChangeListener(), но обходитесь без создания переменной для этого класса.

**Локальные классы**

Локальный класс – это класс, объявленный в блоке Java кода. Обычно локальный класс определяется в методе, но он также может быть объявлен в инициализаторе экземпляра класса. Поскольку все блоки Java кода находятся внутри определения класса, то все локальные классы вложены в окружающие классы.

Определяющей характеристикой локального класса является вложенность для блока кода. Как и локальная переменная, локальный класс действителен только в области видимости окружающего блока.

Пример иллюстрирует определение и использование локального класса Dummy внутри метода i() какого-то класса.

MyInterface myFunction() {

// определение класса Dummy как локального класса

class Dummy implements MyInterface {

// поля и методы класса

}

// возвращаем экземпляр класса Dummy, который только что создали

return Dummy();

}

Свойства локального класса:

* Подобно вложенным нестатическим классам, локальные классы связаны с окружающим экземпляром и имеют доступ ко всем членам, включая private члены окружающего класса.
* Локальный класс нельзя объявить с каким-либо модификатором доступа. Эти модификаторы используются только для членов класса; они не доступны для объявления локальных переменных или классов.
* Как и нестатические вложенные классы, и по тем же причинам, локальные классы не могут иметь static поля, исключение составляют константы, объявленные как static final.

Применение

Основное применение локальные классы находят в тех случаях, когда необходимо написать класс, который будет использоваться внутри одного метода. Таким образом, создание локального класса – способ не загромождать пространство имен.

**Анонимные классы**

Анонимный класс – локальный класс без имени. Анонимный класс определяется и инициализируется в едином выражении с помощью оператора new. Несмотря на то, что определение локального класса в Java – это оператор в блоке, определение анонимного класса представляет собой выражение. Это означает, что его можно записать как часть большого выражения, например метода.

Пример демонстрирует определение и использование анонимного класса.

public class Foo {

// определено множество различных методов

public void foo1() { }

public void foo2() { }

…

public void foo100() { }

}

Хотим создать всего один экземпляр точно такого же класса, как Foo, но с новым методом foo21():

Foo f = new Foo {

public void foo21() {

// код метода

}

}

То есть мы получили экземпляр класса наследника класса Foo с новым методом foo21().

**Свойства анонимного класса:**

* Так как анонимные классы представляют собой один из типов локальных классов, то они также как и локальные классы не могут иметь static поля и методы, кроме констант - static final. Также как и локальные классы, анонимные классы быть записаны с модификатором доступа.
* Заметим, что, поскольку у анонимного класса нет имени, то невозможно определить конструктор анонимного класса.

Применение

Полезной отличительной особенностью локальных и анонимных классов от других «внутренних» классов является то, что первые могут ссылаться на локальные переменные и параметры той области видимости, в которой они определены, объявленные как final. Это доступно локальному классу потому, что компилятор автоматически отдает private поле класса для копии каждой используемой локальной переменной. Кроме того, компилятор добавляет скрытые параметры в каждый конструктор локального класса для инициализации автоматически созданных private полей. Поэтому локальный класс не обращается к локальным переменным, а работает с собственными их копиями. Эти локальные переменные могут быть только final, что дает гарантию того, что значения внутренних копий локальных переменных не расходятся с настоящими значениями локальных переменных.

Можно заметить, что в нашем примере для достижения результата с таким же успехом можно было просто создать новый класс, отнаследованный от класса Foo и создать объект этого нового класса. Но что будет, если нам захочется создать множество единичных экземпляров классов, каждый из которых отличается от Foo всего лишь одной функцией. Согласитесь, в таком случае использование анонимных классов куда удобнее.

Таким образом, особенность анонимных классов заключается в том, что они позволяют определить короткий класс как раз в том месте, где это необходимо. Кроме того, у анонимных классов лаконичный синтаксис, что уменьшает путаницу в коде.

**С# События**

События позволяют сигнализируют системе о том, что произошло определенное действие.

События объявляются в классе с помощью ключевого слова event, после которого идет название делегата:

// Объявляем делегат

public delegate void AccountStateHandler(string message);

// Событие, возникающее при выводе денег

public event AccountStateHandler Withdrawn;

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры, что и делегат, и возвращать тот же тип, что и делегат.

Итак, посмотрим на примере. Для этого возьмем класс Account из прошлой темы и изменим его следующим образом:

class Account

{

// Объявляем делегат

public delegate void AccountStateHandler(string message);

// Событие, возникающее при выводе денег

public event AccountStateHandler Withdrawn;

// Событие, возникающее при добавление на счет

public event AccountStateHandler Added;

int \_sum; // Переменная для хранения суммы

public Account(int sum) { \_sum = sum; }

public int CurrentSum { get { return \_sum; } }

public void Put(int sum)

{

\_sum += sum;

if (Added != null)

Added($"На счет поступило {sum}");

}

public void Withdraw(int sum)

{

if (sum <= \_sum) {

\_sum -= sum;

if (Withdrawn != null)

Withdrawn($"Сумма {sum} снята со счета");

} else {

if (Withdrawn != null)

Withdrawn("Недостаточно денег на счете");

}

}

}

Здесь мы определили два события: Withdrawn и Added. Оба события объявлены как экземпляры делегата AccountStateHandler, поэтому для обработки этих событий потребуется метод, принимающий строку в качестве параметра.

Затем в методах Put и Withdraw мы вызываем эти события. Перед вызовом мы проверяем, закреплены ли за этими событиями обработчики (if (Withdrawn != null)). Так как эти события представляют делегат AccountStateHandler, принимающий в качестве параметра строку, то и при вызове событий мы передаем в них строку.

Теперь используем события в основной программе:

class Program

{

static void Main(string[] args)

{

Account account = new Account(200);

// Добавляем обработчики события

account.Added += Show\_Message;

account.Withdrawn += Show\_Message;

account.Withdraw(100);

// Удаляем обработчик события

account.Withdrawn -= Show\_Message;

account.Withdraw(50);

account.Put(150);

Console.ReadLine();

}

private static void Show\_Message(string message)

{ Console.WriteLine(message); }

}

Для прикрепления обработчика события к определенному событию используется операция += и соответственно для открепления - операция -=: событие += метод\_обработчика\_события. Опять же обращаю внимание, что метод обработчика должен иметь такие же параметры, как и делегат события, и возвращать тот же тип. В итоге мы получим следующий консольный вывод:

Сумма 100 снята со счета

На счет поступило 150

Кроме использованного выше способа прикрепления обработчиков есть и другой с использованием делегата. Но оба способа будут равноценны:

account.Added += Show\_Message;

account.Added += new Account.AccountStateHandler(Show\_Message);

**Замыкания**

Замыкание (англ. closure) в программировании — функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами. Говоря другим языком, замыкание — функция, которая ссылается на свободные переменные в своём контексте.

Scheme

Пример работы замыканий на Scheme:

(define (make-adder n) ; возвращает замкнутое лямбда-выражение

(lambda (x) ; в котором x - связанная переменная,

(+ x n))) ; а n - свободная (захваченная из внешнего контекста)

(define add1 (make-adder 1)) ; делаем процедуру для прибавления 1

(add1 10) ; возвращает 11

(define sub1 (make-adder -1)); делаем процедуру для вычитания 1

(sub1 10) ; возвращает 9

C#

Анонимные методы в C# 2.0 могут замыкаться на локальный контекст:

int[] ary = { 1, 2, 3 };

int x = 2;

var ary1 = Array.ConvertAll<int, int>(ary, delegate(int elem) { return elem \* x; }); // { 2, 4, 6 }

// or..

var ary2 = Array.ConvertAll<int, int>(ary, elem => elem \* x); // { 2, 4, 6 }

Функция Array.ConvertAll преобразует один список/массив в другой, применяя для каждого элемента передаваемую ей в качестве параметра функцию.

В C# 3.0 введены лямбда-выражения, которые делают синтаксис анонимных методов более кратким и выразительным. Соответственно, они также поддерживают замыкания. То есть, замыкания в C# 3.0 практически аналогичны анонимным функциям из C# 2.0, но синтаксически более кратки. Вот тот же пример с применением лямбда-выражений в C# 3.0:

int[] ary = { 1, 2, 3 };

var x = 2;

var ary1 = ary.Select(elem => elem \* x); // { 2, 4, 6 }

Метод Select аналогичен методу Array.ConvertAll за тем исключением, что он принимает и возвращает IEnumerable<T>.

C++

В языке C++ замыкание долгое время не поддерживалось. Однако стандарт языка C++11 вводит лямбда-функции и выражения, ограниченно поддерживающие замыкание:

function<function<int()>()> f = [] {

int x = 0;

return [=] () mutable {return ++x; };

}

auto fun = f();

for (int i = 0; i < 5; ++i) {

cout << fun() << endl;

}

**Обобщённое программирование** — это такой подход к описанию данных и алгоритмов, который позволяет их использовать с различными типами данных без изменения их описания. В Java, начиная с версии J2SE 5.0, добавлены средства обобщённого программирования, синтаксически основанные на C++. Generics (дженерики) или <<контейнеры типа T>> — подмножество обобщённого программирования. (Generics == шаблоны)

**Сопрограмма(coroutin - корутин) –** частный случай генератора(с возможностью передачи значений). (или это обертка вокруг функции, выполнение которой может приостанавливаться ею с возвратом промежуточных результатов и затем продолжаться вызывающей программой с передачей новых значений параметров.)

def calc():

history = []

while True:

x, y = (yield)

if x == 'h':

print history

continue

result = x + y

print result

history.append(result)

c = calc()

------------------------------------------------

print type(c) # <type 'generator'>

c.next() # Необходимая инициация. Можно написать c.send(None)

c.send((1,2)) # Выведет 3

c.send((100, 30)) # Выведет 130

c.send((666, 0)) # Выведет 666

c.send(('h',0)) # Выведет [3, 130, 666]

c.close() # Закрывем генератор

**Статическая и динамическая параметризации**

При динамической параметризации, параметром служит объект данных, при этом связывание формальных и фактических параметра происходит во время выполнения. Различные типы связывания сводятся либо к передаче адреса, либо к копированию соответствующего параметра в локальную память. Следовательно, возможности контроля крайне ограничены. Максимум, что можно сделать – задать тип параметра. Это достаточно надежно, но недостаточно удобно.

При статической параметризации параметрами могут являться:

* объекты данных
* подпрограммы. В большинстве языков подпрограммы являются объектами данных, поэтому этот случай можно рассматривать как частный случай передачи в качестве параметра объектов данных. Но это не всегда хорошо с точки зрения эффективности потому, что, как бы не был реализован подпрограммный тип данных, в этом случае речь идет о передаче указателя на функцию. А все остальные механизмы (например, делегаты в языке С#) - это надежная обертка понятия указателя на функцию (чтобы мы не могли сослаться на неинициализированный указатель и т.д.). Естественно, при передаче и вызове функции через указатель возникает косвенность, что не всегда хорошо с точки зрения эффективности.
* и самое главное – типы данных. Есть языки, в которых типы данных могут являться динамическими параметрами. Но при этом, поскольку тип данных - очень объемлющее понятие, объем информации, который используется при контроле, слишком велик. Поэтому языки, в которых тип данных динамически параметризуем страдают с точки зрения эффективности.

Статическая параметризация должна обеспечить надежность. Механизм статической параметризации есть только в двух языках, причем не самых современных. Языки, которые появились в последнее время (Delphi, C#, Java) не используют механизм статической параметризации. Основная проблема, по-видимому, связана со сложностью реализации соответствующих понятий. Скажем, в С++ до сих пор большинство компиляторов не соответствуют принятому стандарту языка С++. Причина этого несоответствия – это прежде всего механизм шаблонов. Это самый сложный с точки зрения реализации механизм языка С++. Даже такая известная фирма как Microsoft, которая уже больше 10 лет выпускает соответствующий компилятор С++, постоянно его модифицируя, признает, что их нынешний компилятор (так называемый Visual C++ 7), не смотря на то, что его согласованность со стандартом лучше, полной согласованности со стандартом не имеет.

Механизмы статической параметризации в С++ и в Аде различны. С концептуальной точки в языке Ада статическая параметризация проще.

Язык С++ представляет несколько другой образец статической параметризации. В языке С++ соответствующий механизм называется шаблоны (templates). В языке С++ объектами статической параметризации являются классы и функции. Синтаксис очень простой как и в Аде

template<список аргументов>

спецификация класса или функции;

Различия в видах параметров. В языке С++ применяется только 3 вида параметров (раньше их было 2, третий тип параметров добавился для удобства):

1) Тип.

2) Функция.

3) Объект данных.

Параметризация бывает статическая. Все процедурные абстракции параметризуются динамически, т.е. связывание формальных и фактических параметров происходит динамически. Статическая параметризация означает, что связывание формальных и фактических параметров происходит в момент трансляции. С одной стороны, статическая параметризация выглядит как менее мощный механизм, с другой стороны мы можем передавать в момент трансляции значительно больше информации о параметре, чем в динамике. Например, в языках программирования, которые мы рассматриваем, в процедурах и функциях

void f(X x)

очень мало динамической информации о типе передается вместе с фактическим параметром. Поэтому, единственное, что мы можем параметризовать – это объекты данных, поскольку механизм типизации достаточно хорошо характеризует, какие именно объекты данных мы передаем, по крайней мере их размер. Но если мы используем статическую параметризацию, то можно расширить список объектов, которые можно параметризовать. Основное достоинство статической параметризации в противовес динамической – то, что если динамически параметризуются только объекты данных, т.е. переменные или константы. При статической параметризации - объекты данных (функции тоже являются объектами данных, т.к. в языках есть процедурные типы данных) + типы данных. Пример: стек. Допустим мы выбрали какую-то стратегию реализации стека, например, через массив, т.е. статический стек – стек с фиксированной длиной, и тогда у нас возникают 2 вопроса:

* какой длины брать стек
* какой тип элементов в этом стеке

Если бы не было статической параметризации, для каждого типа элементов стека нужно было бы писать свой стек. В языках, где нет статической параметризации, используют, чтобы не писать для каждого типа данных свой стек, указатели. Если язык не объектно-ориентированный, то это нечто типа стек из void\*. void\* – это тот самый горшочек, в который вы можете положить все, что угодно, но под вашу ответственность. В языках с объектным стилем программирования есть так называемая динамическая информация о типе – RTTI –Run Time Type Identification. Там есть специальные методы. В языке Delphy, в языке Java, в языке C# все коллекции – структуры данных типа стек и т.д. также представляют из себя структуру данных, правда не из указателей, а из ссылок. И далее работа идет с помощью механизма идентификации типа. Но статическая параметризация позволяет нам избежать соответствующих накладных расходов. Она позволяет нам параметризовывать не только длину элементов стека, но и их тип. Из рассматриваемых нами 2 языка, содержат статическая параметризация. В языке Ада это механизм родовых сегментов (родовых модулей). В языке С++ - это механизм шаблонов. Существует 2 подхода, когда нам надо параметризовать некоторые более общие свойства, т.е свойства, которые в общем случае связаны с типом объекта. Один – это механизм статической параметризации, другой – механизм, основанный на динамической идентификации типа. Еще в конце 80-ых г.г. известный специалист в области ООП опубликовал статью, в которой обосновал, что механизм динамической идентификации типа, который основан на понятии наследования является более общим, чем механизм статической параметризации в любом языке.

Литература:

1. <https://studizba.com/lectures/10-informatika-i-programmirovanie/331-lekcii-po-programmirovaniyu-na-yavu/4420-lekciya-17.html>
2. <https://studfiles.net/preview/972550/page:2/>

**Ковариантность, контравиантность, инвариантность**

Ковариантностью называется сохранение иерархии наследования исходных типов в производных типах в том же порядке. Так, если класс Cat наследуется от класса Animal, то естественно полагать, что перечисление IEnumerable<Cat> будет потомком перечисления IEnumerable<Animal>. Действительно, «список из пяти кошек» — это частный случай «списка из пяти животных». В таком случае говорят, что тип (в данном случае обобщённый интерфейс) IEnumerable<T> ковариантен своему параметру-типу T.

Контравариантностью называется обращение иерархии исходных типов на противоположную в производных типах. Так, если класс String наследуется от класса Object, а делегат Action<T> определён как метод, принимающий объект типа T, то Action<Object> наследуется от делегата Action<String>, а не наоборот. Действительно, если «все строки — объекты», то «всякий метод, оперирующий произвольными объектами, может выполнить операцию над строкой», но не наоборот. В таком случае говорят, что тип (в данном случае обобщённый делегат) Action<T> контравариантен своему параметру-типу T.

Отсутствие наследования между производными типами называется инвариантностью.

Контравариантность позволяет корректно устанавливать тип при создании подтипов (subtyping), то есть, установить множество функций, позволяющее заменить другое множество функций в любом контексте. В свою очередь, ковариантность характеризует специализацию кода, то есть замену старого кода новым в определённых случаях. Таким образом, ковариантность и контравариантность являются независимыми механизмами типобезопасности, не исключающими друг друга, и могут и должны применяться в объектно-ориентированных языках программирования.

**Ковариантность и контравариантность обобщенных интерфейсов**

Понятия ковариантности и контравариантности связаны с возможностью использовать в приложении вместо некоторого типа другой тип, который находится ниже или выше в иерархии наследования.

Имеется три возможных варианта поведения:

* Ковариантность: позволяет использовать более конкретный тип, чем заданный изначально
* Контравариантность: позволяет использовать более универсальный тип, чем заданный изначально
* Инвариантность: позволяет использовать только заданный тип

Начиная с .NET 4.0 в C# была добавлена возможность создания ковариантных и контравариантных обобщенных интерфейсов. Это функциональность повышает гибкость при использовании обобщенных интерфейсов в программе. По умолчанию все обобщенные интерфейсы, например, IAccout<T> являются инвариантными.

Рассморенние на примерах в С#

Для рассмотрения ковариантных и контравариантных интерфейсов возьмем следующие классы:

class Account

{ public virtual void DoTransfer(int sum)

{ Console.WriteLine($"Клиент положил на счет {sum} долларов"); }

}

class DepositAccount : Account

{ public override void DoTransfer(int sum)

{ Console.WriteLine($"Клиент положил на депозитный счет {sum} долларов"); }

}

Здесь определен класс обычного счета - Account и унаследованный от него класс DepositAccount. В классе Account определен метод, который выполняет условную операцию с счетом. Класс DepositAccount немного переопределяет этот интерфейс.

Ковариантные интерфейсы

Обобщенные интерфейсы могут быть ковариантными, если к универсальному параметру применяется ключевое слово out. Например:

interface IBank<out T>

{ T CreateAccount(int sum); }

class Bank<T> : IBank<T> where T : Account, new()

{ public T CreateAccount(int sum)

{ T acc = new T(); // создаем счет

acc.DoTransfer(sum);

return acc; }

}

Обобщенный интерфейс IBank определяет метод CreateAccount для создания счета. При этом на момент определения интерфейса мы не знаем, какой тип будет представлять счет. Ключевое слово out в определении интерфейса указывает, что данный интерфейс будет ковариантным.

Класс Bank, который представляет условный банк, реализует этот интерфейс и возвращает из метода CreateAccount объект, который представляет либо класс Account, либо один из его наследников.

Применим данные типы в программе:

static void Main(string[] args)

{ IBank<DepositAccount> depositBank = new Bank<DepositAccount>();

Account acc1 = depositBank.CreateAccount(34);

IBank<Account> ordinaryBank = new Bank<DepositAccount>();

// или так -> IBank<Account> ordinaryBank = depositBank;

Account acc2 = ordinaryBank.CreateAccount(45);

Console.Read();

}

То есть мы можем присвоить более общему типу IBank<Account> объект более конкретного типа IBank<DepositAccount> или Bank<DepositAccount>.

В то же время если бы мы не использовали ключевое слово out:

interface IBank<out T>

то мы столкнулись бы с ошибкой в строке

IBank<Account> ordinaryBank = depositBank;

Поскольку в этом случае невозможно было бы привести объект Bank<DepositAccount> к типу IBank<Account>

При создании ковариантного интерфейса надо учитывать, что универсальный параметр может использоваться только в качестве типа значения, возвращаемого методами интерфейса. Но не может использоваться в качестве типа аргументов метода или ограничения методов интерфейса.

Контравариантные интерфейсы

Для создания контравариантного интерфейса надо использовать ключевое слово in. Например, возьмем те же классы Account и DepositAccount и определим следующие типы:

interface ITransaction<in T>

{ void DoOperation(T account, int sum); }

class Transaction<T> : ITransaction<T> where T : Account

{ public void DoOperation(T account, int sum)

{ account.DoTransfer(sum); }

}

Здесь определен интерфейс ITransaction, который представляет условную банковскую операцию. Ключевое слово in в определении интерфейса указывает, что этот интерфейс - контравариантный. Интерфейс определяет метод DoOperation, который принимает некоторый счет и выполняет с ним операцию.

Класс Transaction реализует этот интерфейс и реализует его метод DoOperation.

Применим эти типы в программе:

static void Main(string[] args)

{ ITransaction<Account> accTransaction = new Transaction<Account>();

accTransaction.DoOperation(new Account(), 400);

ITransaction<DepositAccount> depAccTransaction = new Transaction<Account>();

depAccTransaction.DoOperation(new DepositAccount(), 450);

Console.Read();

}

Так как интерфейс ITransaction использует универсальный параметр с ключевым словом in, то он является контравариантным, поэтому в коде мы можем объект Transaction<Account> привести к типу ITransaction<DepositAccount>:

ITransaction<DepositAccount> depAccTransaction = new Transaction<Account>();

Если бы ключевое слово in не использовалось бы, то мы не смогли бы выполнить подобное приведение. То есть объект интерфейса с более универсальным типом приводится к объекту интерфейса с более конкретным типом.

При создании контрвариантного интерфейса надо учитывать, что универсальный параметр контрвариантного типа может применяться только к аргументам метода, но не может применяться к аргументам, используемым в качестве возвращаемых типов.