Абстракция

**Абстрактный класс имеет пустые абстрактные методы**

**Оливчик – не известно какой(ты сам его реализируешь)**

**Микрософт забыли ввести слово concrete**

// Абстрактный класс может быть унаследован от абстрактного класса.

// Реализация абстрактного метода из базового абстрактного класса, в производном абстрактном классе - не обязательна.

// Абстрактный класс.

abstract class AbstractClass

{

public abstract void Method();

}

// Конкретный класс.

class ConcreteClass : AbstractClass

{

public override void Method()

{

Console.WriteLine("Implementation");

}

}

\*\*\*

abstract class AbstractBaseClass

{

// 1.

// Обычный метод передается производному классу как при наследовании от конкретного класса.

public void SimpleMethod()

{

Console.WriteLine("AbstractBaseClass.SimpleMethod");

}

// 2.

// Виртуальный метод передается производному классу как при наследовании от конкретного класса.

virtual public void VirtualMethod()

{

Console.WriteLine("AbstractBaseClass.VirtualMethod");

}

// 3.

// Абстрактный метод - реализуется в производном классе.

abstract public void AbstractMethod();

}

}

Интерфейсы

**Нет модификатора доступа**

**Интерфейс—семантическая и синтаксическая конструкция в коде программы, используемая для специфицирования услуг,предоставляемых классом или компонентом.**

**Интерфейс-стереотип, являющийся аналогом чистого абстрактного класса,в котором запрещена любая реализация.**

**// Наследование интерфейса от интерфейса.**

**Наследование абстрактных классов от интерфейсов**

interface IInterface

{

void Method();

}

class MyClass : IInterface

{

public void Method()

{

Console.WriteLine("Метод - реализация Интерфейса.");

}

IInterface my = new MyClass();

\*\*\*

class DerivedClass : Interface1, Interface2

{

// На 10-й строке реализуем метод с именем Method из базового интерфейса Interface1

// При реализации метода используем технику явного указания имени интерфейса в имени

// метода, которому принадлежит данный метод.

// По умолчанию одноименные методы являются private,

// но явно указывать модификаторы доступа недопустимо.

void Interface1.Method()

{

Console.WriteLine("Реализация метода Method() из Interface1");

}

void Interface2.Method()

{

Console.WriteLine("Реализация метода Method() из Interface2");

}

}

**Наследование**

View -> ObjectBrowser

// Наследование.

// Наследование — механизм объектно-ориентированного программирования, позволяющий описать новый (производный) класс

// на основе уже существующего (базового),

// при этом свойства и функциональность базового класса заимствуются новым производным классом.

// Базовый класс - Производный класс

// Супер класс - Подкласс или (сабкласс)

// Родительский класс - Дочерний класс

// Родитель - Потомок

//К закрытым полям можна получить доступ через методы

\*\*\*

Здесь происходит инициализация поля базового класса через конструктор потомка

public DerivedClass(int number1, int number2)

{

// Инициализируем поле базового класса.

baseNumber = number1;

// Инициализируем поле производного (данного) класса.

derivedField = number2;

}

\*\*\*

// Пользовательский конструктор.

public BaseClass(int number)

{

this.baseNumber = number;

}

// Пользовательский конструктор.

// Вызывается пользовательский конструктор базового класса, при этом не нужно,

// присваивать значения, унаследованным членам в конструкторе производного класса.

public DerivedClass(int number1, int number2)

: base(number1)

{

derivedField = number2;

}

\*\*\*

UpCast DownCast

DerivedClass instance = new DerivedClass();

instance.Method();

// UpCast

BaseClass instanceUp = instance;

instanceUp.Method();

// DownCast

DerivedClass instanceDown = (DerivedClass)instanceUp;

instanceDown.Method();

Полиморфизм

**Полиморфизм** —возможность объектов с одинаковой спецификацией иметь различную реализацию.

**Формы полиморфизма:**

* 1.Ad-hoc полиморфизм
* 2.Классический (принудительный) полиморфизм:
  + •использование виртуальных членов(переопределение virtual/ override).
  + •приведение типов.

**В случаеодновременного использования двух форм классического полиморфизма, первая форма нейтрализует вторую (доминирует над второй).**

При замещение виртуальный метод при UpCast НЕ ВЫЗВЕТСЯ из базового класа, только если это обычный метод

class BaseClass

{

public virtual void Method()

{

Console.WriteLine("Method from BaseClass");

}

}

class DerivedClass : BaseClass

{

// Переопределение метода базового класса.

public override void Method()

{

// Вызов метода базового класса.

base.Method();

Console.WriteLine("Method from DerivedClass");

}

}

\*\*\*

ClassB b = new ClassB();

ClassA a = null;

//--------------------------------------------- is ---------------------------------------------

// Оператор is - проверяет совместимость объекта с заданным типом.

// Если предоставленный объект может быть приведен к предоставленному типу не вызывая исключение,

// выражение is принимает значение true.

// Например, в следующем коде определяется, является ли объект экземпляром типа A или типа, производного от A:

if (b is ClassA)

{

a = (ClassA)b;

}

else

{

a = null;

}

//--------------------------------------------- as---------------------------------------------

// Оператор as используется для выполнения преобразований между совместимыми ссылочными типами.

// Оператор as подобен оператору приведения. Однако, если преобразование невозможно,

// as возвращает значение null, а не вызывает исключение.

// В общем виде логика работы оператора as представляет собой механизм использования оператора is

// (пример на 25 строке), только в сокращенном виде.

a = b as ClassA;

Индексатор

class MyClass

{

private int[] array = new int[5];

// Индексатор.

public int this[int index]

{

get // Аксессор.

{

return array[index];

}

set // Мутатор.

{

array[index] = value;

}

}

}

\*\*\*

class MyClass

{

private string[] array = new string[5];

// Индексатор.

public string this[int index]

{

get // Аксессор.

{

if (index >= 0 && index < array.Length)

return array[index];

else

return "Попытка обращения за пределы массива.";

}

set // Мутатор.

{

if (index >= 0 && index < array.Length)

array[index] = value;

else

Console.WriteLine("Попытка записи за пределами массива.");

}

}

\*\*\*

class MyClass

{

private int[,] array = new int[3, 3];

public int this[int index1, int index2]

{

get

{

return array[index1, index2];

}

set

{

array[index1, index2] = value;

}

}

**Статичиские и вложиные**

// В статических методах нельзя обращаться к нестатическим полям.

public static void Method()

{

//Console.WriteLine("Instance.Id = {0}", Id);

Console.WriteLine("В статических методах нельзя обращаться к нестатическим полям.");

}

// Константы не могут быть статическими.

// public static const float e = 2.71828182845904523536f;

// Поле, не может быть объявлено как static const, поле const по своему поведению, уже является статическим.

// Поле const относится к типу, а не к экземплярам типа.

// Поэтому к полям const можно обращаться с использованием той же нотации ИмяКласса.ИмяЧлена,

// что и в используемой для статических полей.

\*\*\*

class NotStaticClass

{

// Статическое поле.

static int field;

// Статическое свойство.

public static int Property

{

get { return field; }

set { field = value; }

}

}

\*\*\*

/ Статический конструктор.

// Статические конструкторы обладают следующими свойствами:

// 1. Статический конструктор не имеет модификаторов доступа и не принимает параметров.

// 2. Статический конструктор вызывается автоматически для инициализации класса перед созданием первого экземпляра

// или ссылкой на какие-либо статические члены.

// 3. Статический конструктор нельзя вызывать напрямую.

// 4. Пользователь не управляет тем, когда статический конструктор выполняется в программе.

// 5. Типичным использованием статических конструкторов является случай, когда класс использует файл журнала и

// конструктор применяется для добавления записей в этот файл.

namespace Static

{

class NotStaticClass

{

// Статические поля readonly должны быть инициализированы в конструкторе.

static readonly long readonlyField = 2;

// Статическое свойство только для чтения.

public static long ReadonlyField

{

get { return NotStaticClass.readonlyField; }

}

// Статический конструктор.

static NotStaticClass()

{

readonlyField = 1;

}

}

}

\*\*\*

private static int field;

public static int Property

{

get { return field; }

set { field = value; }

}

// Статический конструктор.

// Единственное назначение статических конструкторов - присваивать исходные значения статическим переменным.

static NotStaticClass()

{

Console.WriteLine("Статический конструктор - NotStaticClass()");

field = 1;

}

\*\*\*

// Статический класс в C#, выражает идею паттерна проектирования - Singleton.

// Правила:

// 1. Экземпляр статического класса нельзя создать.

// 2. Static class всегда наследуется от Оbject (Попытка наследования от чего либо другого приводит к ошибке компиляции).

// 3. Static class не реализует интерфейсы. Попытка наследования от интерфейса приводит к ошибке уровня компиляции.

// 4. Содержит только статические члены (наличие в нем нестатического члена приведет к ошибке компиляции).

// 5. Статический класс не может содержать конструкторов экземпляров.

// 6. Статический класс закрыт для наследования от него. Попытка наследования от статического класса приводит к ошибке уровня компиляции.

namespace Static

{

static class StaticClass //: object // Наследование только от Object (явно или неявно)

{

// Контруктор экземпляра - недопустим.

// public StaticClass()

// {

// Console.WriteLine("Constructor.");

// }

// Статический конструктор.

static StaticClass()

{

Console.WriteLine("Static Constructor");

}

// Статический метод.

public static void StaticMethod()

{

Console.WriteLine("StaticMethod");

}

// Нестатический метод - недопустим.

// public void NotStaticMethod()

// {

// Console.WriteLine("NotStaticMethod");

// }

}

}

\*\*\*

Расширяющие методы. (Extension methods)

// Аргумент расширения всегда должен быть только один и стоять первым в списке аргументов.

namespace Extension

{

static class ExtensionClass

{

public static void ExtensionMethod(this string value1, string value2)

{

Console.WriteLine(value1 + value2);

}

}

class Program

{

static void Main()

{

string text = "Hello ";

text.ExtensionMethod("world!");

\*\*\*

NESTED КЛАСС

class MyClass

{

private static int field = 0;

public class Nested

{

public void Method(int a)

{

field = a;

Console.WriteLine(field);

}

}

}

class Program

{

static void Main()

{

MyClass.Nested instance = new MyClass.Nested();

instance.Method(1);

// Delay.

Console.ReadKey();

}

Структуры

// Структуры могут содержать вложенные классы.

// От структур нельзя наследоваться.

// Структуры не могут иметь protected членов.

//protected int field;

Структуры появились раньше чем классы, еще в процедурных языках.

Структура объединяет перемениею.

Екземляры структуры хранятся в стеке или на куче.

В структурах нельзя инициализировать поля непосредственно в месте создания.

Если в структуре имеются члены, которые обращаются к полю и нет пользовательского конструктора,то требуется при создании экземпляра вызывать конструктор по умолчанию. (Иначе будет ошибка.)

\*\*\*

struct MyStruct

{

public int field;

// Конструкторы по умолчанию нельзя задавать явно.

//public MyStruct()

//{

//}

// Если в структуре имеется пользовательский конструктор, то требуется в нем инициализировать все поля.

public MyStruct(int value)

{

Console.WriteLine("Constructor");

this.field = value;

}

}

// Структуры.

// Структуры могут содержать статические члены.

// Статические структуры недопустимы.

// Статический конструктор всегда отрабатывает первым.

static MyStruct()

{

Console.WriteLine("Static Constructor");

}

// Создание экземпляра структурного типа, с вызовом пользовательского конструктора.

MyStruct instance = new MyStruct (33) { field = 0 };

\*\*\*

// В структурах можно создавать автоматически реализуемые свойства,

// при этом требуется использовать конструктор при построении экземпляра.

namespace Structure

{

struct MyStruct

{

public int MyProperty { get; set; }

}

class Program

{

static void Main()

{

MyStruct instance = new MyStruct();

instance.MyProperty = 1;

Console.WriteLine(instance.MyProperty);

// Delay.

\*\*\*

interface IInterface

{

void Method();

}

struct MyStruct : IInterface

{

public void Method()

{

Console.WriteLine("Method");

}

}

Boxing and UnBoxing

// Упаковка и распаковка.

// Упаковка (Boxing) - преобразование структурного типа (типа значения) в ссылочный тип

// (Object или любой другой тип интерфейса, реализуемый этим типом значения).

// Когда тип значения упаковывается средой CLR, она создает программу-оболочку значения внутри

// System.Object и сохраняет ее в управляемой куче.

// Распаковка (Unboxing) - преобразование ссылочного типа в структурный тип.

// Операция распаковки-преобразования извлекает тип значения из объекта.

// Перед распаковкой среда выполнения проверяет совместимость между объектом и структурой,

// в которую будет происходить распаковка.

namespace Boxing

{

class Program

{

static void Main()

{

short a = 25;

// Упаковка переменной - а (Boxing).

object o = a;

// Распаковка объекта (UnBoxing).

short b = (short)o;

// Распаковка должна производиться только в тот тип, из которого производилась упаковка.

//byte s = (byte)o;

\*\*\*

MyStruct my;

my.Method();

// Упаковка (Boxing).

IInterface boxed = my;

boxed.Method();

// Распаковка объекта (UnBoxing).

MyStruct unBoxed = (MyStruct)boxed;

DateTime

// DateTime представляет момент(значение) времени, тогда как TimeSpan представляет интервал(промежуток) времени.

// Это означает, что можно вычесть один экземпляр DateTime из другого для получения объекта TimeSpan,

// который представляет собой временной интервал между ними.

// Или можно прибавить положительное значение TimeSpan к текущему значению DateTime, чтобы получить значение

// DateTime, которое представляет собой будущую дату.

namespace DateTime2

{

class Program

{

static void Main()

{

// Создание Новой даты. DateTime(гг, мм, дд)

DateTime newYearDate = new DateTime(2013, 1, 1);

DateTime today = DateTime.Now;

// Представляет интервал времени.

TimeSpan left = newYearDate - today;

Console.WriteLine("До нового года осталось " + left.Days + " дней");

Console.WriteLine("До нового года осталось " + left.TotalHours + " часов");

// Создание Новой даты и времени. DateTime(гг, мм, дд, чч, мин, сек)

DateTime newDate = new DateTime(2012, 12, 05, 23, 11, 11);

Console.WriteLine(newDate); // Вывод значения даты и времени на экран

Console.WriteLine(newDate.TimeOfDay); // Вывод значение времени, установленного

// пользователем на экран

// Преобразует заданное строковое представление даты и времени в его эквивалент

Console.WriteLine(DateTime.Parse("3/12/2012"));

Console.WriteLine(DateTime.Parse("05 march 2012")); // Месяц написать на локальном языке OS.

// Задержка.

Console.ReadKey();

}

\*\*\*

DateTime now = DateTime.Now;

Console.WriteLine("Дата месяц(прописью) год : {0:D}", now);

Console.WriteLine("Дата.месяц.год : {0:d}", now);

Console.WriteLine("Дата месяц(прописью) год время(чч:мм:сс) : {0:F}", now);

Console.WriteLine("Дата месяц(прописью) год время(чч:мм) : {0:f}", now);

Console.WriteLine("Дата.месяц.год время(чч:мм:cc) : {0:G}", now);

Console.WriteLine("Дата.месяц.год время(чч:мм) : {0:g}", now);

Console.WriteLine("Текущий месяц и дата : {0:M}", now);

Console.WriteLine("Текущий месяц и год : {0:Y}", now);

Console.WriteLine("Время(чч:мм:cc) : {0:T}", now);

Console.WriteLine("Время(чч:мм) : {0:t}", now);

Перечесление

// Перечисления.

// Перечисление - это набор именованных констант, которые хранят числовые значения.

// Перечисление определяет именованные константы, каждой из которых соответствует числовое значение.

// Все перечисления в C# происходят от единого Базового класса System.Enum

namespace Enums

{

// При компиляции - компилятор подставляет вместо имен,

// установленные им в соответствие числовые значения. [имя] = [число]

// По умолчанию типом данных констант перечисления будет int.

// Можно использовать любой целый тип данных C# (byte, sbyte, short, ushort, int, uint, long, ulong)

enum EnumType : byte // Явно указываем использовать тип byte.

{

Zero = 0,

One = 1,

Two = 2,

Three = 3

}

class Program

{

static void Main()

{

Console.WriteLine(EnumType.One);

Console.WriteLine((byte)EnumType.One);

EnumType digit = EnumType.Zero;

Console.WriteLine(digit);

Console.WriteLine((byte)digit);

// Delay.

Console.ReadKey();

}

}

\*\*\*

// Можно использовать псевдоним для любого целого типа данных C# (byte, sbyte, short, ushort, int,

// uint, long, ulong)

// Нельзя использовать любой системный целый тип данных C# (Byte, SByte, Int16, UInt16, Int32,

// UInt32, Int64, UInt64)

enum EnumType //: Int32 // Ошибка.

{

Zero = 0,

One = 1,

Two = 2,

Three = 3

}

Делегаты (Delegates)

**Делегат**(**delegate)**—это разновидность объектов которые содержат в себе указатели на методы.

static class MyClass

{

// Создаем статический метод, который планируем сообщить с делегатом.

public static void Method()

{

Console.WriteLine("Строку вывел метод сообщенный с делегатом.");

}

}

// На 21-й строке создаем класс-делегата с именем MyDelegate,

// метод, который будет сообщен с экземпляром данного класса-делегата,

// не будет ничего принимать и не будет ничего возвращать.

public delegate void MyDelegate(); // Создаем класс делегата. (1)

class Program

{

static void Main()

{

MyDelegate myDelegate = new MyDelegate(MyClass.Method); // Создаем экземпляр делегата. (2)

myDelegate.Invoke(); // Вызываем метод сообщенный с делегатом. (3)

myDelegate(); // Другой способ вызова метода сообщенного с делегатом. (3')

// Delay.

Console.ReadKey();

}

\*\*\*

// Класс, метод которого будет сообщен с делегатом.

class MyClass

{

// Создаем метод, который планируем сообщить с делегатом.

public string Method(string name)

{

return "Hello " + name;

}

}

// На 21-й строке создаем класс-делегата с именем MyDelegate,

// метод, который будет сообщен с экземпляром данного класса-делегата,

// будет принимать один строковой аргумент и возвращать строковое значение.

public delegate string MyDelegate(string name); // Создаем класс делегата. (1)

class Program

{

static void Main()

{

MyClass instance = new MyClass();

MyDelegate myDelegate = new MyDelegate(instance.Method); // Создаем экземпляр делегата и сообщаем с ним метод. (2)

string greeting = myDelegate.Invoke("Jeffrey Richter"); // Вызываем метод сообщенный с делегатом. (3)

Console.WriteLine(greeting);

greeting = myDelegate("Grady Booch"); // Другой способ вызова метода сообщенного с делегатом. (3')

Console.WriteLine(greeting);

// Delay.

\*\*\*

MyDelegate myDelegate = null;

MyDelegate myDelegate1 = new MyDelegate(Method1);

MyDelegate myDelegate2 = new MyDelegate(Method2);

MyDelegate myDelegate3 = new MyDelegate(Method3);

MyDelegate myDelegate5 = myDelegate - myDelegate2;

// Комбинируем делегаты.

myDelegate = myDelegate1 + myDelegate2 + myDelegate3

\*\*\*

// Анонимные (лямбда) методы.

namespace Delegates

{

// Создаем класс делегата.

public delegate void MyDelegate();

class Program

{

static void Main()

{

// Создаем экземпляр класса-делегата MyDelegate и сообщаем с ним анонимный метод.

MyDelegate myDelegate = delegate { Console.WriteLine("Hello world!"); };

// Вызов анонимного метода, сообщенного с делегатом.

myDelegate();

// Delay.

Console.ReadKey();

}

\*\*\*

public delegate int MyDelegate(int a, int b);

class Program

{

static void Main()

{

int summand1 = 1, summand2 = 2, sum = 0;

MyDelegate myDelegate = delegate(int a, int b) { return a + b; };

sum = myDelegate(summand1, summand2);

Console.WriteLine("{0} + {1} = {2}", summand1, summand2, sum);

// Delay.

Console.ReadKey();

}

}

\*\*\*

// Создаем класс делегата.

public delegate void MyDelegate(ref int a, ref int b, out int c);

class Program

{

static void Main()

{

int summand1 = 1, summand2 = 2, sum;

MyDelegate myDelegate = delegate(ref int a, ref int b, out int c) { a++; b++; c = a + b; };

myDelegate(ref summand1, ref summand2, out sum);

Console.WriteLine("{0} + {1} = {2}", summand1, summand2, sum);

// Delay.

Console.ReadKey();

}

\*\*\*

Лямбда выражение

myDelegate = delegate(int x) { return x \* 2; }; // Лямбда-Метод

myDelegate = (x) => { return x \* 2; }; // Лямбда-Оператор.

myDelegate = x => x \* 2; // Лямбда-Выражение.

int result = myDelegate(4);

**Лямбда-выражение**—это анонимные методы,которая содержит выражения и операторы и может использоваться для создания делегатов.

\*\*\*

MyDelegate myDelegate;

myDelegate = delegate { Console.WriteLine("Hello 1"); }; // Лямбда-Метод

myDelegate += () => { Console.WriteLine("Hello 2"); }; // Лямбда-Оператор.

myDelegate += () => Console.WriteLine("Hello 3"); // Лямбда-Выражение.

myDelegate();

\*\*\*

delegate Delegate3 Functional(Delegate1 delegate1, Delegate2 delegate2);

delegate string Delegate1();

delegate string Delegate2();

delegate string Delegate3();

class Program

{

public static Delegate3 MethodF(Delegate1 delegate1, Delegate2 delegate2)

{

return delegate { return delegate1.Invoke() + delegate2.Invoke(); };

}

public static string Method1() { return "Hello "; }

public static string Method2() { return "world!"; }

static void Main()

{

Functional functional = new Functional(MethodF);

Delegate3 delegate3 = functional.Invoke(new Delegate1(Method1), new Delegate2(Method2));

Console.WriteLine(delegate3.Invoke());

\*\*\*

delegate void MyDelegate(int argument);

class Program

{

static void Main()

{

MyDelegate my = null; // Требуется обязательно присвоить null.

// Требуется отдельное присвоение ссылки на делегат с сообщенным лямбда оператором,

// в месте создания переменной, недопустимо сразу создавать лямбда оператор.

my = (int i) =>

{

i--;

Console.WriteLine("Begin {0}", i);

if (i > 0)

{

my(i);

}

Console.WriteLine("End {0}", i);

};

my(3);

// Delay.

Console.ReadKey();

}

Универсальные шаблоны (обобщение)

**Обобщение(Универсальныешаблоны)**–элемент кода,способный адаптироваться для выполнения общин (сходных) действий над различными типами данных.

class MyClass<T>

{

public T field;

public void Method()

{

Console.WriteLine(field.GetType());

}

}

class Program

{

static void Main()

{

// Создаем экземпляр класса MyClass и в качестве параметра типа (тип MyClass) передаем тип int.

MyClass<int> instance1 = new MyClass<int>();

instance1.Method();

// Создаем экземпляр класса MyClass и в качестве параметра типа (тип MyClass) передаем тип long.

MyClass<long> instance2 = new MyClass<long>();

instance2.Method();

// Создаем экземпляр класса MyClass и в качестве параметра типа (тип MyClass) передаем тип string.

MyClass<string> instance3 = new MyClass<string>();

instance3.field = "ABC";

instance3.Method();

// Delay.

Console.ReadKey();

}

\*\*\*

class MyGenerics<TYPE1, TYPE2>

{

// Поля

private TYPE1 variable1;

private TYPE2 variable2;

// Конструктор.

public MyGenerics(TYPE1 argument1, TYPE2 argument2)

{

this.variable1 = argument1;

this.variable2 = argument2;

}

}

MyGenerics<int, int> instance1 = new MyGenerics<int, int>(1, 2);

Console.WriteLine(instance1.Variable1 + instance1.Variable2);

\*\*\*

class MyClass

{

public void Method<T>(T argument)

{

T variable = argument;

Console.WriteLine(variable);

}

}

class Program

{

static void Main()

{

MyClass instance = new MyClass();

instance.Method<string>("Hello world!");

instance.Method("Привет мир!");

// Delay.

Console.ReadKey();

}

}

\*\*\*

// На 11-й строке создаем класс-делегата с именем MyDelegate, параметризированный двумя Указателями Места Заполнения Типом - Т и R,

// метод который будет сообщен с экземпляром данного класса-делегата, будет принимать один аргумент, типа Указателя Места Заполнения Типом - Т,

// и возвращать значение типа Указателя Места Заполнения Типом - R.

delegate R MyDelegate<T, R>(T t);

class Program

{

public static int Add(int i)

{

return ++i;

}

public static string Concatenation(string s)

{

return "Hello " + s + "!";

}

static void Main()

{

MyDelegate<int, int> myDelegate1 = new MyDelegate<int, int>(Add);

int i = myDelegate1.Invoke(1);

Console.WriteLine(i);

MyDelegate<string, string> myDelegate2 = new MyDelegate<string, string>(Concatenation);

string s = myDelegate2("Alex");

Console.WriteLine(s);

Событие(Events)

Событийно-ориентированное программированиетия

**Событийно-ориентированное программирование(event-drivenprogramming)**— парадигма программирования, в которой выполнение программы определяется событиями—действиями пользователя (клавиатура, мышь), сообщениями других программи потоков ,событиями операционной системы (например, поступлением селевого пакета).

**События**-это особый тип многоадресных делегатов,которые можно вызвать только из класса или структуры, вкоторой они объявлены (класс издателя).Если насобытие подписаны другие классы или структуры, их методыобработчиков событий будут вызваны, когда класс издателя инициирует событие.

\*\*\*

namespace Events

{

public delegate void EventDelegate();

public class MyClass

{

public event EventDelegate myEvent = null;

public void InvokeEvent()

{

myEvent.Invoke();

}

}

class Program

{

// Методы обработчики события.

static private void Handler1()

{

Console.WriteLine("Обработчик события 1");

}

static private void Handler2()

{

Console.WriteLine("Обработчик события 2");

}

static void Main()

{

MyClass instance = new MyClass();

// Присоединение обработчиков событий. (Подписка на событие)

instance.myEvent += new EventDelegate(Handler1);

instance.myEvent += Handler2;

// Метод который вызывает событие.

instance.InvokeEvent();

Console.WriteLine(new string('-', 20));

// Открепляем Handler2().

instance.myEvent -= new EventDelegate(Handler2);

instance.InvokeEvent();

// Delay.

Console.ReadKey();

}

}

}

\*\*\*

public delegate void EventDelegate();

public class MyClass

{

EventDelegate myEvent = null;

// Реализация методов доступа add и remove для события.

public event EventDelegate MyEvent

{

add { myEvent += value; }

remove { myEvent -= value; }

}

public void InvokeEvent()

{

myEvent.Invoke();

}

}

\*\*\*

namespace Events

{

public delegate void PressKeyEventHandler();

public class Keyboard

{

public event PressKeyEventHandler PressKeyA = null;

public event PressKeyEventHandler PressKeyB = null;

// C ... Z

public void PressKeyAEvent()

{

if (PressKeyA != null)

{

PressKeyA.Invoke(); // вызываем метод Invoke на событии

}

}

public void PressKeyBEvent()

{

if (PressKeyB != null)

{

PressKeyB.Invoke();

}

}

public void Start()

{

while (true)

{

string s = Console.ReadLine();

switch (s)

{

case "a":

case "A":

PressKeyAEvent();

break;

case "b":

case "B":

PressKeyBEvent();

break;

case "exit":

goto Exit;

default:

Console.WriteLine("Нет обработчика нажатия на клавишу {0}", s);

break;

}

}

Exit:

Console.WriteLine("Exit!");

}

}

class Program

{

// Методы обработчики события.

static private void PressKeyA\_Handler()

{

Console.ForegroundColor = ConsoleColor.Blue;

Console.WriteLine();

Console.WriteLine(" X ");

Console.WriteLine(" X X ");

Console.WriteLine(" X X ");

Console.WriteLine(" XXXXXXX ");

Console.WriteLine("X X");

Console.WriteLine();

Console.ForegroundColor = ConsoleColor.Gray;

}

static private void PressKeyB\_Handler()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine();

Console.WriteLine("XXXXX ");

Console.WriteLine("X X ");

Console.WriteLine("XXXXXX ");

Console.WriteLine("X X");

Console.WriteLine("XXXXXX ");

Console.WriteLine();

Console.ForegroundColor = ConsoleColor.Gray;

}

static void Main()

{

Keyboard keyboard = new Keyboard();

// Подписка на событие.

keyboard.PressKeyA += new PressKeyEventHandler(PressKeyA\_Handler);

keyboard.PressKeyB += PressKeyB\_Handler;

// Запуск метода, который будет следить за нажатием клавиш

keyboard.Start();

}

}

}

\*\*\*

public delegate void EventDelegate();

interface IInterface

{

event EventDelegate MyEvent; // Абстрактное событие.

}

public class BaseClass : IInterface

{

EventDelegate myEvent = null;

public virtual event EventDelegate MyEvent // Виртуальное событие.

{

add { myEvent += value; }

remove { myEvent -= value; }

}

\*\*\*

public delegate void EventDelegate();

interface IInterface

{

event EventDelegate MyEvent;

void InvokeEvent();

}

public class BaseClass : IInterface

{

public virtual event EventDelegate MyEvent = null;

public virtual void InvokeEvent()

{

MyEvent.Invoke();

}

}

public class DerivedClass : BaseClass

{

public override event EventDelegate MyEvent = null;

public override void InvokeEvent()

{

MyEvent.Invoke();

}

}

\*\*\*

Анонимный метод

MyClass instance = new MyClass();

// Присоединение обработчиков событий.

instance.MyEvent += new EventDelegate(Handler1);

instance.MyEvent += new EventDelegate(Handler2);

instance.MyEvent += delegate { Console.WriteLine("Анонимный метод 1."); };

instance.InvokeEvent();

Потоки (Threads)

**Многозадачность**—свойство операционной системы или среды программирования обеспечивать возможность параллельной (или псевдопараллельной ) обработки нескольких процессов

Процессор выполняет каждую програму понимножку.

\*\*\*

class Program

{

// Статический метод, который планируется выполнять одновременно в главном (первичном) и во вторичном потоках.

static void WriteSecond()

{

// CLR назначает каждому потоку свой стек и методы имеют свои собственные локальные переменные.

// Отдельный экземпляр переменной counter создается в стеке каждого потока,

// поэтому для каждого потока выводятся, свои значения counter - 0,1,2.

int counter = 0;

while (counter < 10)

{

Console.WriteLine("Thread Id {0}, counter = {1}", Thread.CurrentThread.GetHashCode(), counter);

counter++;

}

}

static void Main()

{

// Работа вторичного потока.

Thread thread = new Thread(WriteSecond);

thread.Start();

// Работа первичного потока.

WriteSecond();

// Delay.

Console.ReadKey();

}

}

\*\*\*

Передача аргументов в метод потока либо можна создать класс и потом в методе вызывать его поля

class Program

{

// Метод, который планируется выполнять в отдельном потоке.

static void WriteSecond(object argument)

{

for (int i = 0; i < 10; i++)

{

Console.WriteLine(argument);

Thread.Sleep(1000);

}

}

static void Main()

{

ParameterizedThreadStart writeSecond = new ParameterizedThreadStart(WriteSecond);

Thread thread = new Thread(WriteSecond);

thread.Start("Hello");

Thread.Sleep(500);

// Delay.

Console.ReadKey();

}

}

\*\*\*

thread.IsBackground = true; - завершает роботу вторичного потока

class Program

{

// Метод, который планируется выполнять в отдельном потоке.

static void WriteSecond()

{

while (true)

{

Console.WriteLine(new string(' ', 15) + "Secondary");

Thread.Sleep(500);

}

}

static void Main()

{

// Работа вторичного потока.

ThreadStart writeSecond = new ThreadStart(WriteSecond);

Thread thread = new Thread(writeSecond);

thread.Start();

// Работа первичного потока.

for (int i = 0; i < 10; i++)

{

Console.WriteLine("Primary");

Thread.Sleep(500);

}

// Завершить работу вторичного потока

thread.IsBackground = true;

}

}

\*\*\*

// Критическая секция (critical section).

// lock - блокирует блок кода так, что в каждый отдельный момент времени, этот блок кода

// сможет использовать только один поток. Все остальные потоки ждут пока текущий поток, закончит работу.

В данном примере консоль есть разделяемым ресурсом.

Критическая секция – это lock

Объект блокировки – это block

Конструкция lock – это безопасная конструкция (пример с туалетом)

class MyClass

{

object block = new object();

public void Method()

{

int hash = Thread.CurrentThread.GetHashCode();

lock (block) // Закомментировать lock.

{

for (int counter = 0; counter < 10; counter++)

{

Console.WriteLine("Поток # {0}: шаг {1}", hash, counter);

Thread.Sleep(100);

}

Console.WriteLine(new string('-', 20));

}

}

class Program

{

static void Main()

{

Console.SetWindowSize(80, 40);

MyClass instance = new MyClass();

for (int i = 0; i < 3; i++)

{

new Thread(instance.Method).Start();

}

Thread.Sleep(500);

// Delay.

Console.ReadKey();

}

}

}

\*\*\*

// lock - это сокращенное использование System.Threading.Monitor.

// Monitor.Enter(this) - блокирует блок кода так, что его может использовать только текущий поток.

// Все остальные потоки ждут пока текущий поток, закончит работу и вызовет Monitor.Exit(this).

Monitor.Enter(block); // Закомментировать.

for (int counter = 0; counter < 10; counter++)

{

Console.WriteLine("Поток # {0}: шаг {1}", hash, counter);

Thread.Sleep(100);

}

Console.WriteLine(new string('-', 20));

Monitor.Exit(block); // Закомментировать.

\*\*\*

Если закоментировать lock вторичного потока – будет полный ппц

}

Коллекции

yield

class UserCollection

{

public static IEnumerable Power()

{

yield return "Hello world!";

}

}

foreach (var element in UserCollection.Power())

Console.WriteLine(element);

\*\*\*

**Коллекция**–это класс, предназначенный для группировки связанных объектов,управленияими и обработки их в циклах.

**Коллекциистоитприменять,если:**

* •Отдельные элементы используются для одинаковых целей иодинаково важны.
* •Намомент компиляции числоэлементов неизвестно или незафиксировано.
* •Необходима піддержка операции перебора всех элементов.
* •Необходима піддержка упорядочивания элементов.
* •Необходимо использовать элементы из библиотеки, откоторой потребитель ожидает наличия типа коллекции

- колекции можна разширить.

IEnumerator – bool Current; boll MoveNext(); void Reset();

Обработка исключений

**Исключения** – неожиданные события, прерывающие нормальную работу кода ивозникающие вовремя исполнения программы.

В процессора есть Алу Ариф Лог устрой.

\*\*\*

try

{

// Попытка деления на ноль.

a = a / (2 - n);

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

}

catch (Exception e)

{

Console.WriteLine("Обработка исключения.");

Console.WriteLine(e.Message);

}

\*\*\*

Exception ex = new Exception("Мое Исключение");

try

{

throw ex;

}

catch (Exception e)

{

Console.WriteLine("Обработка исключения.");

Console.WriteLine(e.Message);

}

По слабой ссылке

\*\*\*

try

{

throw new Exception("Мое Исключение");

}

catch (Exception e)

{

Console.WriteLine("Обработка исключения.");

Console.WriteLine(e.Message);

}

\*\*\*

class MyClass

{

public void MyMethod()

{

Exception exception = new Exception("Мое исключение");

exception.HelpLink = "http://MyCompany.com/ErrorService";

exception.Data.Add("Причина исключения: ", "Тестовое исключение");

exception.Data.Add("Время возникновения исключения: ", DateTime.Now);

throw exception;

}

}

class Program

{

static void Main()

{

try

{

MyClass instance = new MyClass();

instance.MyMethod();

}

catch (Exception e)

{

Console.WriteLine("Имя члена: {0}", e.TargetSite);

Console.WriteLine("Класс определяющий член: {0}", e.TargetSite.DeclaringType);

Console.WriteLine("Тип члена: {0}", e.TargetSite.MemberType);

Console.WriteLine("Message: {0}", e.Message);

Console.WriteLine("Source: {0}", e.Source);

Console.WriteLine("Help Link: {0}", e.HelpLink);

Console.WriteLine("Stack: {0}", e.StackTrace);

foreach (DictionaryEntry de in e.Data)

Console.WriteLine("{0} : {1}", de.Key, de.Value);

}

\*\*\*

Блок finally выполнится всегда

// finally - срабатывает в случае возникновения исключения OutOfMemoryException.

int a = 1, n = 2;

try

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Попытка деления на ноль.");

Console.WriteLine("a / (2 - n) = {0}", a / (2 - n));

}

catch (Exception e)

{

Console.ForegroundColor = ConsoleColor.White;

Console.BackgroundColor = ConsoleColor.Red;

Console.WriteLine("Обработка исключения.");

Console.WriteLine(e.Message);

}

finally

{

Console.ForegroundColor = ConsoleColor.Gray;

Console.BackgroundColor = ConsoleColor.Black;

}

\*\*\*

Пользоватильское исключение

try

{

throw new UserException();

}

catch (UserException userException)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("Обработка исключения 1:");

userException.Method();

try

{

throw userException;

}

catch (UserException exception)

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Обработка исключения 2:");

exception.Method();

}

}

finally

{

Console.ForegroundColor = ConsoleColor.Gray;

}

Операторы

1. Два екземпляра одного и того же класса имеют разные хэши, поэтому метод Equals возвращает false, чтобы они были одинаковые стоит ссылкакм присвоить один и тотже обьект.
2. Статические члены всегда наследуются

Присвоить свой хэшкод обьекту

class MyClass

{

public override int GetHashCode()

{

return 1234567890;

}

}

class Program

{

static void Main()

{

MyClass instance = new MyClass();

Console.WriteLine(instance.GetHashCode());

// Delay.

Console.ReadKey();

}

}

\*\*\*

Клонирование

// Клонирование.

Статическое поле всегда одинаковое как для оригинала так и клона

Клонирование происходит таке и по ирархии наследования

Асоциации не клонируются, а клонируются поверхосно

MyDerivedClass clone = original.MemberwiseClone() as MyDerivedClass;

Console.WriteLine(clone.age + " " + clone.name + " " + MyDerivedClass.CompanyName + "\n");

\*\*\*

public class Point : ICloneable

{

public int x, y;

public Point()

{

}

public Point(int x, int y)

{

this.x = x;

this.y = y;

}

// Реализация метода интерфейса ICloneable.

public object Clone()

{

return new Point(this.x, this.y) as object;

}

public override string ToString()

{

return "X: " + x + " Y: " + y;

}

}

Point original = new Point(100, 100);

Point clone = original.Clone() as Point;

Console.WriteLine("Первая проверка");

Console.WriteLine(original);// сработает переопределленый метод ToString

Console.WriteLine(clone);

// Изменяем clone.x (при этом original.x не изменится)

clone.x = 0;

// Проверка.

Console.WriteLine("Вторая проверка после изменения");

Console.WriteLine(original);

Console.WriteLine(clone);

Перегрузка операторов

// Использовать ключевое слово operator, можно только вместе с ключевым словом static !

public static Point operator +(Point p1, Point p2)

{

return new Point(p1.x + p2.x, p1.y + p2.y);

}

public override string ToString()

{

return string.Format("[{0}, {1}]", this.x, this.y);

}

Then

Point c = a + b;

\*\*\*

public static Point operator ++(Point p1)

{

return new Point(p1.x + 1, p1.y + 1);

}

Then

a++ /екземпляр

\*\*\*

public static bool operator ==(Point p1, Point p2)

{

return p1.Equals(p2);

}

// Перегруженный оператор !=.

public static bool operator !=(Point p1, Point p2)

{

return !p1.Equals(p2);

}

public override bool Equals(object o)

{

if (o is Point)

{

if (((Point)o).x == this.x && ((Point)o).y == this.y)

return true;

}

return false;

}

// Переопределение GetHashCode() - обязательна при переопределении Equals().

public override int GetHashCode()

{

return this.ToString().GetHashCode();

}

Then

Console.WriteLine("a == b = {0}", a == b);

Console.WriteLine("a != b = {0}", a != b);

\*\*\*

public static bool operator <(Point p1, Point p2)

{

return (p1.CompareTo(p2) < 0);

}

public static bool operator >(Point p1, Point p2)

{

return (p1.CompareTo(p2) > 0);

}

public int CompareTo(object obj)

{

if (obj is Point)

{

Point p = (Point)obj;

if (this.x > p.x && this.y > p.y)

{

return 1;

}

else if (this.x < p.x && this.y < p.y)

{

return -1;

}

else

{

return 0;

}

}

else

{

throw new ArgumentException();

}

}

Then

Console.WriteLine("a < b = {0}", a < b);

Console.WriteLine("a > b = {0}", a > b);

\*\*\*

Испольовать перегрузку можна зразу в том же класе где есть перегрузка

public static Point operator +(Point p1, Point p2)

{

return new Point(p1.x + p2.x, p1.y + p2.y);

}

public static Point Add(Point p1, Point p2)

{

return p1 + p2;

}

\*\*\*

Операторы преобразования

// Ключевое слово explicit служит для создания оператора явного преобразования типа.

namespace MyNamespace

{

struct Digit

{

public byte value;

// Конструктор.

Digit(byte value)

{

this.value = value;

}

// Оператор явного преобразования типа byte-to-Digit.

public static explicit operator Digit(byte argument)

{

Digit digit = new Digit(argument);

return digit;

}

public override string ToString()

{

return this.value.ToString();

}

}

class MainClass

{

static void Main()

{

byte variable = 1;

// Явное преобразование byte-to-Digit.

Digit digit = (Digit)variable;

Console.WriteLine(digit);

// Delay.

Console.ReadKey();

}

}

\*\*\*

Implicit

Неявне

Digit digit = variable;

Анонимные и динамичиские типы. LINQ.

Анонимные типы предлагают удобный способ инкапсуляции набора свойств в один объект без необходимости предварительного явного определениятипа.

**Language Integrated Query (LINQ)**—проект Microsoft по добавлению синтаксиса языказа просов, напоминающего SQL, в язики программирования платформы.NETFramework.

**Все операции запроса LINQ состоят из трех различных действий.**

* •Получение источника данных.
* •Создание запроса.
* •Выполнение запроса

// Анонимные типы предлагают удобный способ инкапсулирования набора свойств в один объект

// без необходимости предварительного явного определения типа.

// Имя типа создается компилятором и недоступно на уровне исходного кода.

// Анонимные типы являются ссылочными типами, которые происходят непосредственно от класса object.

// Компилятор присваивает им имена, несмотря на то что эти имена недоступны для приложения.

namespace Anonymous

{

class Program

{

static void Main()

{

var instance = new { Name = "Alex", Age = 27 };

Console.WriteLine("Name = {0}, Age = {1}", instance.Name, instance.Age);

Type type = instance.GetType();

Console.WriteLine(type.ToString());

// Delay.

Console.ReadKey();

}

}

\*\*\*

var instance = new { Name = "Alex", Age = 27, Id = new { Number = 123 } };

Console.WriteLine("Name = {0}, Age = {1}, Id = {2}", instance.Name, instance.Age, instance.Id.Number);

\*\*\*

class MyClass

{

public int field;

public void Method()

{

Console.WriteLine(field);

}

}

class Program

{

static void Main()

{

var instance = new { My = new MyClass() };//ЧЕРЕЗ СВОЙСТВО можна ссылатся на обьект

instance.My.field = 1;

instance.My.Method();

// Delay.

Console.ReadKey();

}

\*\*\*

class MyClass

{

public int field;

public void Method()

{

Console.WriteLine(field);

}

}

class Program

{

static void Main()

{

new

{

My = new MyClass { field = 1 }

}.My.Method();

// Delay.

Console.ReadKey();

}

}

\*\*\*

delegate void MyDelegate(string @string);

class Program

{

static void Main()

{

var instance = new

{

MyDel = new MyDelegate((string @string) => Console.WriteLine(@string))

};

instance.MyDel("Hello world!");

// Delay.

Console.ReadKey();

}

}

LINQ

Var query – это ссылка на коллекцию

public class Employee

{

public string FirstName { get; set; }

public string LastName { get; set; }

public decimal Salary { get; set; }

public DateTime StartDate { get; set; }

}

class Program

{

static void Main()

{

// База данных сотрудников.

var employees = new List<Employee>

{

new Employee

{

FirstName = "Ivan",

LastName = "Ivanov",

Salary = 94000,

StartDate = DateTime.Parse("1/4/1992")

},

new Employee

{

FirstName = "Petr",

LastName = "Petrov",

Salary = 123000,

StartDate = DateTime.Parse("12/3/1985")

},

new Employee

{

FirstName = "Andrew",

LastName = "Andreev",

Salary = 1000000,

StartDate = DateTime.Parse("1/12/2005")

}

};

// Выражение запроса.

var query = // query - переменная запрса.

from employee in employees // from - объявляет переменную диапазона employee.

where employee.Salary > 100000 // where - фильтр

orderby employee.LastName, employee.FirstName

select new // select - Опреация проекции.

{

LastName = employee.LastName,

FirstName = employee.FirstName

};

Console.WriteLine("Высокооплачиваемые сотрудники:");

foreach (var item in query)

Console.WriteLine("{0} {1}", item.LastName, item.FirstName);

\*\*\*

Сам query – это последователый вызов расшеряющихся методов

var query = // query - переменная запрса.

employees

.Where(emp => emp.Salary > 100000)

.OrderBy(emp => emp.LastName)

.OrderBy(emp => emp.FirstName)

.Select(emp => new

{

LastName = emp.LastName,

FirstName = emp.FirstName

});

\*\*\*

Вызываються наши методы

public static class MySet

{

public static IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, bool> predicate)

{

Console.WriteLine("Вызвана собственная реализация Where.");

return System.Linq.Enumerable.Where(source, predicate);

}

public static IEnumerable<R> Select<T, R>(this IEnumerable<T> source, Func<T, R> selector)

{

Console.WriteLine("Вызвана собственная реализация Select.");

return System.Linq.Enumerable.Select(source, selector);

}

}

class Program

{

static void Main()

{

int[] numbers = { 1, 2, 3, 4 };

var query = from x in numbers

where x % 2 == 0

select x \* 2;

foreach (var item in query)

Console.WriteLine(item);

// Delay.

Console.ReadKey();

}

}

\*\*\*

Таблица умножения

//работает как вложеный цикл сначала в х помещается 1 и в у 1 потом х не трогаем, а в у помещаем 2 и тд

var query = from x in Enumerable.Range(1, 9) // Таблица умножения от 1 до 9.

from y in Enumerable.Range(1, 10)

select new

{

X = x,

Y = y,

Product = x \* y

};

foreach (var item in query)

Console.WriteLine("{0} \* {1} = {2}", item.X, item.Y, item.Product);

\*\*\*

ArrayList numbers = new ArrayList();

numbers.Add(1);

numbers.Add(2);

// Если используем колекциию не generic, то нужно указывать тип.

// ЯВНОЕ указание типа Int32 переменной диапазона - n. (var - НЕВОЗМОЖНО использовать т.к. IEnumerable не параметризированный!)

var query = from int n in numbers

select n \* 2;

foreach (var item in query)

Console.WriteLine(item);

\*\*\*

Join

// Построить коллекцию сотрудников.

var employees = new List<EmployeeID>

{

new EmployeeID {Id = "111", Name = "Ivan Ivanov"},

new EmployeeID {Id = "222", Name = "Andrey Andreev"},

new EmployeeID {Id = "333", Name = "Petr Petrov"},

new EmployeeID {Id = "444", Name = "Alex Alexeev"}

};

// Построить коллекцию национальностей.

var empNationalities = new List<EmployeeNationality>

{

new EmployeeNationality {Id = "111", Nationality = "Russian"},

new EmployeeNationality {Id = "222", Nationality = "Ukrainian"},

new EmployeeNationality {Id = "333", Nationality = "American"},

};

// Построить запрос.

// Получение списка имен всех сотрудников вместе с их национальностями, при этом отсортировав список по убыванию.

var query = from emp in employees

join n in empNationalities

on emp.Id equals n.Id

orderby n.Nationality descending // ascending - по возрастанию | descending - по убыванию.

select new

{

Id = emp.Id,

Name = emp.Name,

Nationality = n.Nationality

};

foreach (var person in query)

Console.WriteLine("{0}, {1}, \t{2}", person.Id, person.Name, person.Nationality);

\*\*\*

var query = from emp in employees

orderby emp.Nationality ascending,

emp.LastName descending, // ascending - по возрастанию | descending - по убыванию.

emp.FirstName descending // по умолчанию - ascending.

select emp;

foreach (var person in query)

Console.WriteLine("{0}, \t{1}, \t{2}", person.LastName, person.FirstName, person.Nationality);

\*\*\*

public class Result

{

public Result(int input, int output)

{

Input = input;

Output = output;

}

public int Input { get; set; }

public int Output { get; set; }

}

class Program

{

static void Main()

{

int[] numbers = { 1, 2, 3, 4 };

// Построить запрос.

var query = from x in numbers

select new Result(x, x \* 2);// в х попадет 777, а не 1;

numbers[0] = 777; // Выражение запроса выполняется в момент обращения к переменной запрса в foreach.

foreach (var item in query)

Console.WriteLine("Input = {0}, \t Output = {1}", item.Input, item.Output);

\*\*\*

int[] numbers = { 1, 2, 3, 4 };

// Построить запрос.

var query = from x in numbers

select new { Input = x, Output = x \* 2 };

foreach (var item in query)

Console.WriteLine("Input = {0}, \t Output = {1}", item.Input, item.Output);

\*\*\*

static void Main()

{

// Построить запрос.

// let - представляет новый локальный идентификатор, на который можно ссылаться в остальной части запроса.

// Его можно представить, как локальную переменную видимую только внутри выражения запроса

var query = from x in Enumerable.Range(0, 10)

let innerRange = Enumerable.Range(0, 10)

from y in innerRange

select new { X = x, Y = y, Product = x \* y };

foreach (var item in query)

Console.WriteLine("{0} \* {1} = {2}", item.X, item.Y, item.Product);

\*\*\*

GroupeBy

// Построить коллекцию сотрудников с национальностями.

var employees = new List<Employee>

{

new Employee {LastName = "Andreev", FirstName = "Andrew", Nationality = "Ukrainian"},

new Employee {LastName = "Ivanov", FirstName = "Ivan", Nationality = "Russian"},

new Employee {LastName = "Andreev", FirstName = "Ivan", Nationality = "Ukrainian"},

new Employee {LastName = "Petrov", FirstName = "Petr", Nationality = "American"},

new Employee {LastName = "Andreev", FirstName = "Sergey", Nationality = "Ukrainian"},

new Employee {LastName = "Petrov", FirstName = "Slava", Nationality = "American"}

};

// Построить запрос.

var query = from emp in employees

group emp by new { Nationality = emp.Nationality, LastName = emp.LastName };

foreach (var group in query)

{

Console.WriteLine(group.Key);

foreach (var employee in group)

Console.WriteLine(employee.FirstName);

}

\*\*\*

int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Построить запрос.

// Разделение чисел на четные и нечетные.

// into - подобно let, позволяет определить локальный по отношению к запросу идентификатор.

var query = from x in numbers

group x by x % 2 into partition

where partition.Key == 0

select new { Key = partition.Key, Count = partition.Count(), Group = partition };

foreach (var item in query)

{

Console.WriteLine("mod2 == {0}", item.Key);

Console.WriteLine("Count == {0}", item.Count);

foreach (var number in item.Group)

Console.Write("{0}, ", number);

Динамичиские типы

С динамическими типами не работает интелисенс

События не могут быть динамическими

dynamic variable = 1;

Console.WriteLine(variable);

variable = "Hello world!";

Console.WriteLine(variable);

variable = DateTime.Now;

Console.WriteLine(variable);

Динам стат поля

class Program

{

static dynamic field = 1;

static void Main()

{

Console.WriteLine(field);

field = "Hello world!";

dynamic field = 1, field2 = "Hello", field3 = true;

static void Main()

{

dynamic instance = new Program();

Console.WriteLine(instance.field);

instance.field = "Hello world!";

Console.WriteLine(instance.field);

Множество параметров делегата

delegate R MyDelegate<T, T2, R>(T argument, T2 arg);

class Program

{

static dynamic Method(dynamic argument, dynamic arg)

{

return argument + arg;

}

static void Main()

{

dynamic myDelegate = new MyDelegate<dynamic, dynamic, dynamic>(Method);

dynamic @string = myDelegate("Hello world!","xc");

Console.WriteLine(@string);

// Один из параметров бинарного оператора, должен иметь существующий тип.

//public static dynamic operator +(dynamic pointA, dynamic pointB) - так недопустимо.

public static dynamic operator +(Point pointA, dynamic pointB)

{

return new Point(pointA.x + pointB.x, pointA.y + pointB.y);

}

Пространство имен

**Пространства имен (namespace)**— это способ, благодаря котрому **.NET** избегает конфликтов имен между классами.

Директива using-импортирует пространство имен, избавляя от необходимости погной квалификации имен стереотипов

\*\*\*

// Полная квалификация имени стереотипа, включает имя пространства имен, в котором находится стереотип.

System.Console.WriteLine("Hello world");

\*\*\*

Если закоментировать, то будет ошибка

namespace NamespaceA

{

class MyClass

{

public MyClass()

{

Console.WriteLine("Constructor - MyClass");

}

}

}

using NamespaceA; // Закомментировать.

namespace Namespaces

{

class Program

{

static void Main()

{

MyClass my = new MyClass();

}

}

}

ИЛИ

NamespaceA.MyClass my = new NamespaceA.MyClass();

\*\*\*

ПОЛНЫЙ ПУТЬ ВЛОЖЕННОСТИ

namespace NamespaceA

{

namespace NamespaceB

{

namespace NamespaceC

{

class MyClass

{

public MyClass()

{

System.Console.WriteLine(this.GetType().Name);

}

}

}

}

}

namespace Namespaces

{

using NamespaceA.NamespaceB.NamespaceC;

class Program

{

static void Main()

{

MyClass my = new MyClass();

}

}

}

\*\*\*

// Создание псевдонима MyClass, для класса MyClassC из пространства имен NamespaceA.NamespaceB.NamespaceC.

using MyClass = NamespaceA.NamespaceB.NamespaceC.MyClassC;