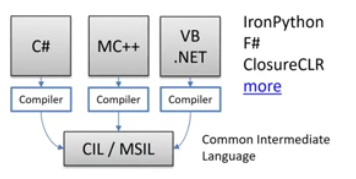
# Умные названия

## CIL(common intermediate language)

это результат компиляции кода(мб с++ или с# и др), чтото на подобии ассемблера



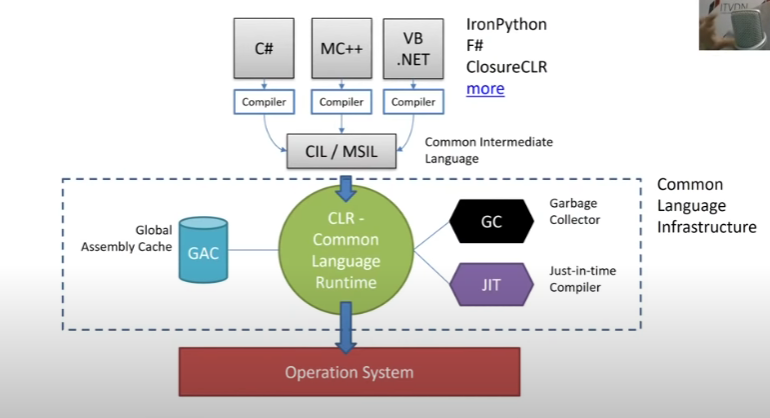
После компиляции будут exe или dll, которые содержат этот промежуточный язык, они для ОС ничего не означают и запустить их не получится

Чтобы запустить их нужен комплекс инструментов…

## CLR – common language runtime

Виртуальная машина управляющая всемвсемвсем(сборщик мусора, управление потокам и тд), и она преобразовывает CIL код в byte code который понятен ОС. Этот механизм который занимается преобразованием, называется ***JID*** компилятор.

***JID*** компилятор компилирует приложение по ходу выполнения. То есть есть функция в начале приложения и еще одна когда нажимаем на кнопку, Так вот код на кнопку будет выполнен только когда нажмем



Gac – это глобальный кэш где все наши инструменты, либы там, все чем приложение пользуется, а в приложении только инструкции.

GC- понятно

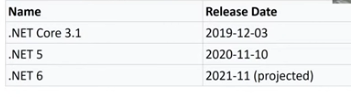
## Реализации CLI

**.NET Framework** -только windows

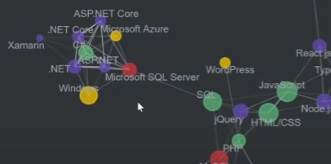
**Micro framework** – для кофейных автоматов

**Xamarin**-android, IOS

**.net Core**- cross-platform – здесь особенности,



Был Кор 3.1 в 2019 году, потом в 2020 вышел .NET 5 и .NET 6, в 2021, его переименовали до 5й чтобы не путали, .NET Framework не будет получать мажорных версий, .NET 4 пропущен чтобы не путать с .NET Framework 4.x, .NET Framework 5.x не будет

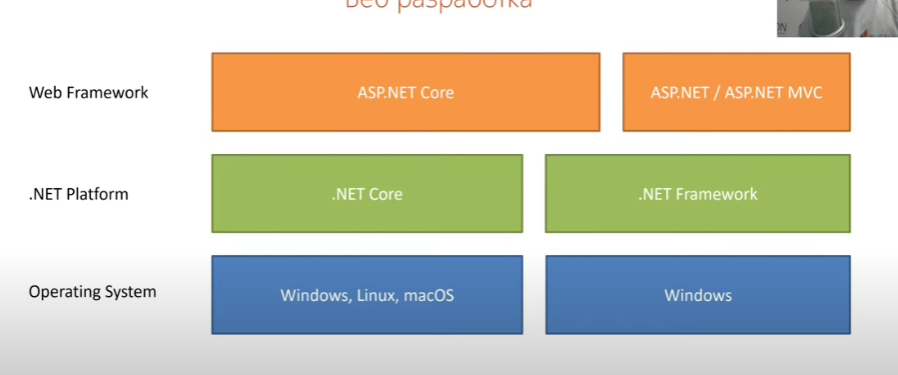


Asp.net – приложения для web’a

ASP.NET Core- работает на .net core

.net- на .net Frameworke

## Web разработка

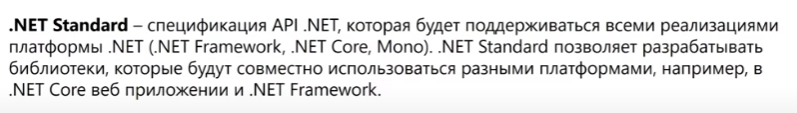


ASP.Net mvc работает только на .net Framework’e, но сейчас в основном на коре стараются делать

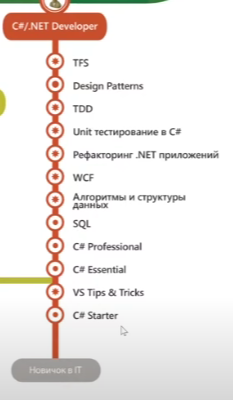
## Core или framework

Тут еще за Core – он полностью переписан и с ним работает куда быстрее чем mvc

## .Net standard



## RoadMap

\* - не обязательно



## notes по обучению

https://app.pluralsight.com/paths/skill/csharp

тебе крч нужен будет фреймворк для веб-сервисов, 100% - https://app.pluralsight.com/paths/skill/aspnet-core

как пробежишься - попробуй codewars, сразу будут вопросы)

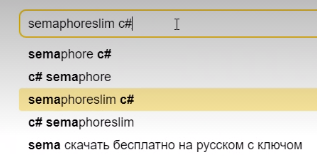
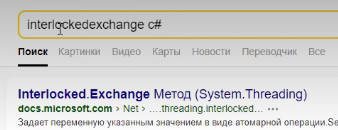
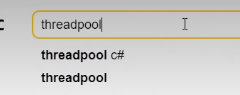
о, в конце прайса есть разработка типа сайта, полезная штука имхо

Важно: List, массивы, Dictionary, HashSet, енумераторы и LINQ

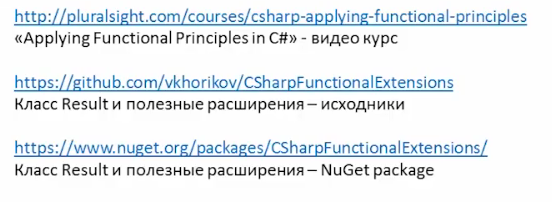
и мегаважно понимать разницу в структурах и классах, типичные вопросы на собеседованиях про них

ConcurentDictionary

readrideLock

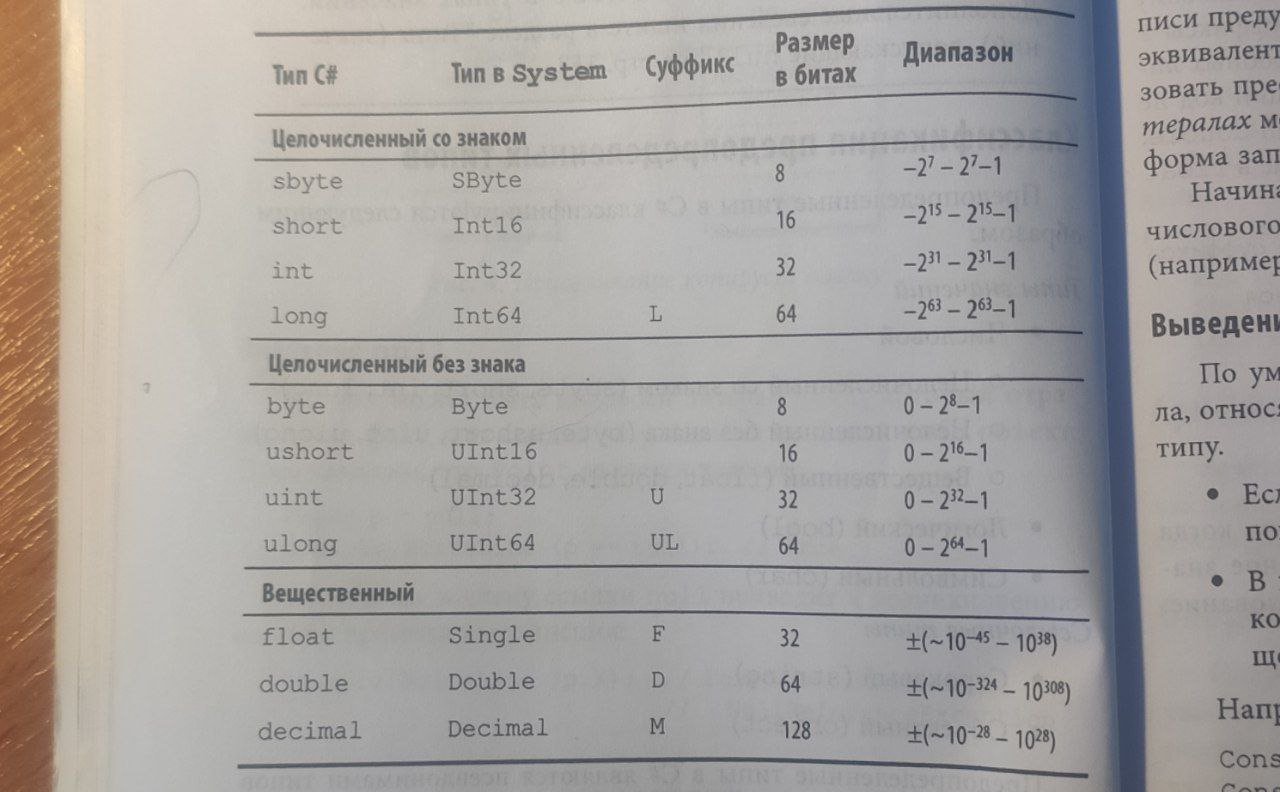
  

Дополнительно про исключения



# Types

## cStandart



**bool**: хранит значение true или false (логические литералы). Представлен системным типом System.Boolean

**byte**: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte

**sbyte**: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte

**short**: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16

**ushort**: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16

**int**: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:

**uint**: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32

**long**: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом System.Int64

**ulong**: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64

* **float**: хранит число с плавающей точкой от -3.4\*1038 до 3.4\*1038 и занимает 4 байта. Представлен системным типом System.Single
* **double**: хранит число с плавающей точкой от ±5.0\*10-324 до ±1.7\*10308 и занимает 8 байта. Представлен системным типом System.Double
* **decimal**: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от ±1.0\*10-28 до ±7.9228\*1028, может хранить 28 знаков после запятой и занимает 16 байт. Представлен системным типом System.Decimal
* **char**: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом System.Char. Этому типу соответствуют символьные литералы:
* **string**: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.
* **object**: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом System.Object, который является базовым для всех других типов и классов .NET.

|  |  |
| --- | --- |
| **1**  **2**  **3** | **int a = 10;**  **int b = 0b101;  // бинарная форма b =5**  **int c = 0xFF;   // шестнадцатеричная форма c = 255** |
| 1  2  3 | long a = -10;  long b = 0b101;  long c = 0xFF; |
| **1**  **2**  **3** | **char a = 'A';**  **char b = '\x5A';**  **char c = '\u0420';** |

# Public/private/static/internal

# Разница между class/struct

1. Class is reference type, struct – not

# struct

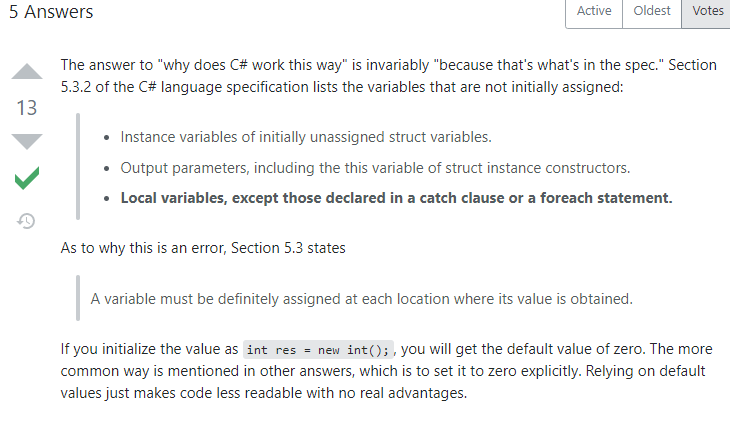
# classes

# questions

1. почему 15 в hex это 15:Х2?
2. Какая раница в задании? Почему у первого ошибка?

SecondNameSpace**.**MyClass demoObj**;**demoObj**.**b**();//Local variable 'demoObj' might not be initialized before accessing**var demoObj2 = new SecondNameSpace**.**MyClass**();**demoObj2**.**b**();**

Типа на стэк овфлоу



Но чето не удовлетворило меня это

Вообще в Свойствах можно держать значение

public int Prop **{** get**;** set**; }** = 1**;**

Как сделать чтобы в set можно было указать значение Свойства а не поля?

public int Prop **{** get**;** set**; }** = 1**;**public int Prop2 **{** get**;** set**{** Prop2 = ..**. }; }** = 1**;**

зачем нужны partial методы классов когда в одном объявление а в другом юзаем? Его и без этого видно там:

partial class MyClass1  
**{** public MyClass1**()  
 {** Cout**();  
 }  
}**partial class MyClass1  
**{** void Cout**()  
 {** Console**.**WriteLine**(**"partial method"**);  
 }  
}**

# Модификаторы доступа

## Internal

Делает поле как паблик во всем проекте

* **в наследнике** (даже если не паблик) - можно без создания экземпляра
* **не в наследнике** – надо создать экземпляр и у него будет доступно методы/переменные

## Protected

Дает доступ наследникам **даже если класс находится в другом проекте**

## protected internal

это как бы сумма обоих, и во всем родном проекте и в наследниках в других проектах

## protected private

доступно из текущего класса и производных классов, которые определены в этом же проекте

то есть как protected но нельзя в других проектах юзать

## Для классов

Дефолт: internal

Private – нельзя, public – можно

Вложенные классы – все как с переменными

# Структуры

## заметки

* не может иметь конструктора без параметров,
* нам не обязательно вызывать конструктор для создания объекта структуры, -new



* конструктор без параметров вызывается втихоря и обнуляет все значения



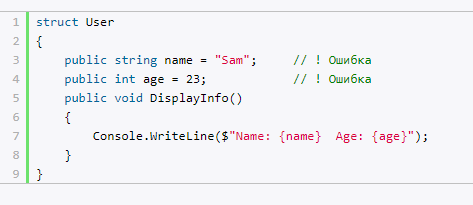
* если через new то обнуляет



* Также, как и для класса, можно использовать инициализатор для создания структуры:



* Но в отличие от класса нельзя инициализировать поля структуры напрямую при их объявлении, например, следующим образом:



* это тип значения а не ссылка
* нет наследования
* нужно обычно где надо копировать значение а не ссылку
* если сами пишем конструктор то надо определять всеее поля

# Классы

## Вызов конструктора из другого

чек порядок выполнения конструкторов, следующий конструктор вызывается перед кодом

class Person  
**{** public string name = "Ben"**;** public int age = 18**;** public string email = "ben@gmail.com"**;** public Person**(**string name**)  
 {** this**.**name = name**;  
 }** public Person**(**string name**,** int age**)** : this**(**name**)  
 {** this**.**age = age**;  
 }** public Person**(**string name**,** int age**,** string email**)** : this**(**"Bob"**,** age**)  
 {** this**.**email = email**;  
 }  
}**

Значения полей:

* name: Bob
* age: 31
* email: tom@gmail.com

Порядок выполнения конструкторов:

1. Вначале вызывается конструктор с тремя параметрами: Person(string name, int age, string email). Он вызывает конструктор с двумя параметрами Person(string name, int age). Этому конструктору передаются значения name: "Bob", age: 31
2. Далее вызывается конструктор с двумя параметрами: Person(string name, int age). Он вызывает конструктор с одним параметром Person(string name). Этому конструктору передаются значения name: "Bob"
3. Далее вызывается конструктор с одним параметром: Person(string name). Он получает строку "Bob" и устанавливает ее в качестве значения для поля name
4. Затем выполняется конструктор с двумя параметрами: Person(string name, int age). Он устанавливает число 31 в качестве значения для поля age
5. Далее выполняется конструктор с тремя параметрами: Person(string name, int age, string email). Он устанавливает строку "tom@gmail.com" в качестве значения для поля email

## Static

статические методы имеют доступ только к статическим полям. Нестатические поля класса не могут использоваться в статических методах.

## Наследование

### Заметки

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

* Не поддерживается множественное наследование, класс может наследоваться только от одного класса.
* При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**.

Однако следует также учитывать, что если базовый и производный класс находятся в разных сборках (проектах), то в этом случае производый класс может наследовать только от класса, который имеет модификатор public.

* Если класс объявлен с модификатором **sealed**, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

|  |  |
| --- | --- |
| 1  2  3 | sealed class Admin  {  } |

* Нельзя унаследовать класс от статического класса.
* если в базовом классе **не определен** конструктор по умолчанию без параметров, а только конструкторы с параметрами (как в случае с базовым классом Person), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово base.

1. Когда наследуем или просто создаем класс всегда сааамым первым создается System.Object.Object()

### Последовательность, конструктуроов

class Person  
**{** string name**;** int age**;**   
 public Person**()  
 {  
 }** public Person**(**string name**)** : this**(**name**,** 18**)  
 {  
 }** public Person**(**string name**,** int age**)  
 {** this**.**name = name**;** this**.**age = age**;  
 }  
}**class Employee : Person  
**{** string company**;**   
 public Employee**()  
 {  
 }** public Employee**(**string name**,** int age**,** string company**)**: base**(**name**,** age**)  
 {** this**.**company = company**;  
 }** public Employee**(**string name**,** string company**)** : base**(**name**)  
 {** this**.**company = company**;  
 }  
}**

Порядок выполнения конструкторов:

1. System.Object.Object()
2. Person(string name, int age)
3. Person(string name)
4. Employee(string name, string company)

### почему родителей поля детей после приведении

var child2 = new Child();

var parent2 = child2

потому что parent2 это теперь не парент, ты в него ссылку на чайлда прописал

в коде поля child не будут доступны

просто дебаггер их видит

## Свойства класса

### Сокращенная запись свойств

    // эквивалентно public string Name { get { return name; } }

    public string Name => name;

### можем писать private set

class Person  
**{** private string name**;**   
 public string Name  
 **{** get  
 **{** return name**;  
 }**   
 private set  
 **{** name = value**;  
 }  
 }** public Person**(**string name**)  
 {** Name = name**;  
 }**

кстати так мы делать, не можем:

public string Name  
**{** internal get => \_name**;** protected set => \_name = value**;  
}**

или так

public string Name  
**{** internal get **{** return name**; }  
}**

короче там как то все по умному

# Namespace

## псевдонимы

using Printer = System**.**Console**;**

будет работать:

Printer**.**WriteLine**(**"printer"**);**System**.**Console**.**WriteLine**(**"sys console"**);**

## Using static

using static System**.**Console**;**namespace HelloApp  
**{** class Program  
 **{** static void Main**(**string**[]** args**)  
 {** WriteLine**(**"Hello from C# 8.0"**);** Read**();  
 }  
 }  
}**

# const / readOnly

## заметки

* Константы должны быть определены во время компиляции, а поля для чтения могут быть определены во время выполнения программы.
* Соответственно инициализировать константу можно устанновить только при ее определении.
* Поле для чтения можно инициализировать либо при его определении, либо в конструкторе класса.
* Константы не могут использовать модификатор static, так как уже неявно являются статическими. Поля для чтения могут быть как статическими, так и не статическими.

## Readonly struct

Кроме полей для чтения в C# можно определять структуры для чтения. Для этого они предваряются модификатором **readonly**:

|  |  |
| --- | --- |
| 1 | readonly struct User { } |

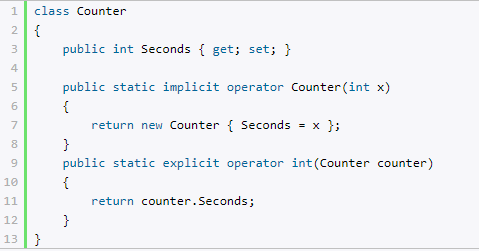
Особенностью таких структур является то, что все их поля должны быть также полями для чтения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | readonly struct User  {      public readonly string name;      public User(string name)      {          this.name = name;      }  } |

То же самое касается и свойств, которые должны быть доступны только для чтения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | readonly struct User  {      public readonly string Name { get; } // указывать readonly необязательно      public int Age { get; } // свойство только для чтения      public User(string name, int age)      {          this.Name = name;          this.Age = age;      }  } |

# Перегрузка операций преобразования типов



# Virtual vs new

The "new" keyword doesn't override, it signifies a new method that has nothing to do with the base class method.

public class Foo

{

public bool DoSomething() { return false; }

}

public class Bar : Foo

{

public new bool DoSomething() { return true; }

}

public class Test

{

public static void Main ()

{

Foo test = new Bar ();

Console.WriteLine (test.DoSomething ());//false

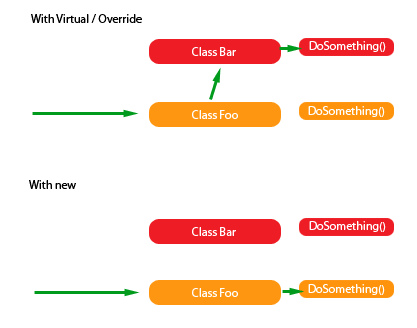
}

}

**This prints false, if you used override it would have printed true**

So, if you are doing real polymorphism you **SHOULD ALWAYS OVERRIDE**. The only place where you need to use "new" is when the method is not related in any way to the base class version.

Короче мы идем из Base to Derived и говорим что у них нет ничего общего если пишем new



# Обобщенные классы

Ну мы можем юзать поля класса, подставляя туда разные типы, можем ограничивать типы через where:

class Account<T**,** U>  
 where U : struct  
 where T : MyClass**,** new**()  
{** public static T Session**;** *// int or string* public int Id **{** get**;** set**; }** public U Sum **{** get**;** set**; }** public void Display**()  
 {** Console**.**WriteLine**(**$"Session: {Session} \nId: {Id} \nSum: {Sum}\n" + new string**(**'-'**,** 10**));  
 }  
}**

Теперь вместо Т можно подставлять только MyClass или наследников

## Статические

### Статические поля обобщенных классов

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Account<int> account1 = new Account<int> { Sum = 5000 };  Account<int>.session = 5436;    Account<string> account2 = new Account<string> { Sum = 4000 };  Account<string>.session = "45245";    Console.WriteLine(Account<int>.session);      // 5436  Console.WriteLine(Account<string>.session);   // 45245 |

В итоге для Account<string> и для Account<int> будет создана своя переменная session.

## Список ограничений

В качестве ограничений мы можем использовать следующие типы:

* Классы
* Интерфейсы
* class - универсальный параметр должен представлять класс
* struct - универсальный параметр должен представлять структуру
* new() - универсальный параметр должен представлять тип, который имеет общедоступный (public) конструктор без параметров

|  |  |
| --- | --- |
| where T : struct | The type argument must be a non-nullable [value type](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-types). For information about nullable value types, see [Nullable value types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types). Because all value types have an accessible parameterless constructor, the struct constraint implies the new() constraint and can't be combined with the new() constraint. You can't combine the struct constraint with the unmanaged constraint. |
| where T : class | The type argument must be a reference type. This constraint applies also to any class, interface, delegate, or array type. In a nullable context in C# 8.0 or later, T must be a non-nullable reference type. |
| where T : class? | The type argument must be a reference type, either nullable or non-nullable. This constraint applies also to any class, interface, delegate, or array type. |
| where T : notnull | The type argument must be a non-nullable type. The argument can be a non-nullable reference type in C# 8.0 or later, or a non-nullable value type. |
| where T : default | This constraint resolves the ambiguity when you need to specify an unconstrained type parameter when you override a method or provide an explicit interface implementation. The default constraint implies the base method without either the class or struct constraint. For more information, see the [default constraint](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/unconstrained-type-parameter-annotations" \l "default-constraint) spec proposal. |
| where T : unmanaged | The type argument must be a non-nullable [unmanaged type](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/unmanaged-types). The unmanaged constraint implies the struct constraint and can't be combined with either the struct or new() constraints. |
| where T : new() | The type argument must have a public parameterless constructor. When used together with other constraints, the new() constraint must be specified last. The new() constraint can't be combined with the struct and unmanaged constraints. |
| where T : *<base class name>* | The type argument must be or derive from the specified base class. In a nullable context in C# 8.0 and later, T must be a non-nullable reference type derived from the specified base class. |
| where T : *<base class name>?* | The type argument must be or derive from the specified base class. In a nullable context in C# 8.0 and later, T may be either a nullable or non-nullable type derived from the specified base class. |
| where T : *<interface name>* | The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic. In a nullable context in C# 8.0 and later, T must be a non-nullable type that implements the specified interface. |
| where T : *<interface name>?* | The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic. In a nullable context in C# 8.0, T may be a nullable reference type, a non-nullable reference type, or a value type. T may not be a nullable value type. |
| where T : U | The type argument supplied for T must be or derive from the argument supplied for U. In a nullable context, if U is a non-nullable reference type, T must be non-nullable reference type. If U is a nullable reference type, T may be either nullable or non-nullable. |

## Наследование

Первый вариант заключается в создание класса-наследника, который типизирован тем же типом, что и базовый:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class UniversalAccount<T> : Account<T>  {      public UniversalAccount(T id) : base(id)      {        }  } |

Применение класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Account<string> acc1 = new Account<string>("34");  Account<int> acc3 = new UniversalAccount<int>(45);  UniversalAccount<int> acc2 = new UniversalAccount<int>(33);  Console.WriteLine(acc1.Id);  Console.WriteLine(acc2.Id);  Console.WriteLine(acc3.Id); |

Второй вариант представляет создание обычного необобщенного класса-наследника. В этом случае при наследовании у базового класса надо явным образом определить используемый тип:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class StringAccount : Account<string>  {      public StringAccount(string id) : base(id)      {      }  } |

Теперь в производном классе в качестве типа будет использоваться тип string. Применение класса:

|  |  |
| --- | --- |
| 1  2  3  4 | StringAccount acc4 = new StringAccount("438767");  Account<string> acc5 = new StringAccount("43875");  // так нельзя написать  //Account<int> acc6 = new StringAccount("45545"); |

Третий вариант представляет типизацию производного класса параметром совсем другого типа, отличного от универсального параметра в базовом классе. В этом случае для базового класса также надо указать используемый тип:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class IntAccount<T> : Account<int>  {      public T Code { get; set; }      public IntAccount(int id) : base(id)      {      }  } |

Здесь тип IntAccount типизирован еще одним типом, который может не совпадать с типом, который используется базовым классом. Применение класса:

|  |  |
| --- | --- |
| 1  2  3  4 | IntAccount<string> acc7 = new IntAccount<string>(5) { Code = "r4556" };  Account<int> acc8 = new IntAccount<long>(7) { Code = 4587 };  Console.WriteLine(acc7.Id);  Console.WriteLine(acc8.Id); |

### При этом стоит учитывать

что если на уровне базового класса для универсального параметра установлено ограничение, то подобное ограничение должно быть определено и в производных классах, которые также используют этот параметр:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Account<T> where T : class  {      public T Id { get; private set; }      public Account(T \_id)      {          Id = \_id;      }  }  class UniversalAccount<T> : Account<T>      where T: class  {      public UniversalAccount(T id) : base(id)      {        }  } |

То есть если в базовом классе в качестве ограничение указано class, то есть любой класс, то в производном классе также надо указать в качестве ограничения class, либо же какой-то конкретный класс.

# Try catch

Можно отлавливать конкретные

int x = 1**;** int y = 0**;**   
 try  
 **{** int result = x / y**;  
 }** catch**(**DivideByZeroException**)** when **(**y==0 && x == 0**)  
 {** Console**.**WriteLine**(**"y не должен быть равен 0"**);  
 }** catch**(**DivideByZeroException ex**)  
 {** Console**.**WriteLine**(**ex**.**Message**);  
 }**

# тип exception

* **InnerException**: хранит информацию об исключении, которое послужило причиной текущего исключения
* **Message**: хранит сообщение об исключении, текст ошибки
* **Source**: хранит имя объекта или сборки, которое вызвало исключение
* **StackTrace**: возвращает строковое представление стека вызывов, которые привели к возникновению исключения
* **TargetSite**: возвращает метод, в котором и было вызвано исключение

Но также есть более специализированные типы исключений, которые предназначены для обработки каких-то определенных видов исключений. Их довольно много, я приведу лишь некоторые:

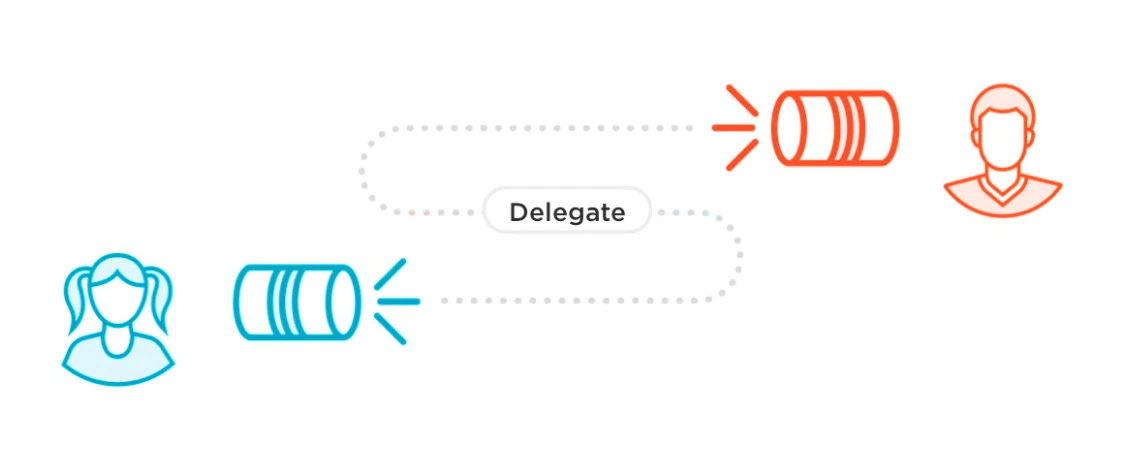
* **DivideByZeroException**: представляет исключение, которое генерируется при делении на ноль
* **ArgumentOutOfRangeException**: генерируется, если значение аргумента находится вне диапазона допустимых значений
* **ArgumentException**: генерируется, если в метод для параметра передается некорректное значение
* **IndexOutOfRangeException**: генерируется, если индекс элемента массива или коллекции находится вне диапазона допустимых значений
* **InvalidCastException**: генерируется при попытке произвести недопустимые преобразования типов
* **NullReferenceException**: генерируется при попытке обращения к объекту, который равен null (то есть по сути неопределен)
* **{** try  
   **{** object obj = "you"**;** int num = **(**int**)**obj**;**     *// InvalidCastException* Console**.**WriteLine**(**$"Результат: {num}"**);  
   }** catch **(**DivideByZeroException**)  
   {** Console**.**WriteLine**(**"Возникло исключение DivideByZeroException"**);  
   }** catch **(**IndexOutOfRangeException**)  
   {** Console**.**WriteLine**(**"Возникло исключение IndexOutOfRangeException"**);  
   }** catch **(**Exception ex**)  
   {** Console**.**WriteLine**(**$"Исключение: {ex**.**Message}"**);  
   }**    
   Console**.**Read**();  
  }**

И в данном случае блок catch (Exception ex){} будет обрабатывать все исключения кроме DivideByZeroException и IndexOutOfRangeException.

# Делегаты

## Theory:

Это связующее звено между listner’OM and senderом, like pipeline



Под капотом когда у нас есть event and delegate, there is class MulticlassDelegate, он отслеживает что куда идет

## Syntax

Это указатель на метод, можно несколько методов выполнять, даже одних и тех же

Общее:

delegate int Operation**(**int x**,** int y**);** Operation del = Add**;** *// делегат указывает на метод Add* int result = del**(**4**,**5**);** *// фактически Add(4, 5)*

--

private static int Add**(**int x**,** int y**)  
{** return x+y**;  
}**

параметры и тип делегата должен четко совпадать

delegate void SomeDel(int a, double b); только такие параметры и только такой тип

добавляем методы так: mes1 += HowAreYou, причем они должны быть СТАТИЧЕСКИМИ если они из этого же класса

Вообще присваивать делегату можно не только статические методы "родного" класса или статические методы "чужого" класса, но и не статические методы. Я имею в виду присваивание делегату метода через обьект класса, что позволяет пользоваться не только статическими методами.  
Что-то на подобие этого...  
SomeClass obj = new SomеClass();  
MyDelegate del = obj.SC\_Method;

С делегатами тесно связаны **анонимные методы**. Анонимные методы используются для создания экземпляров делегатов.

delegate void MessageHandler**(**string message**);**static void Main**(**string**[]** args**)  
{** ShowMessage**(**"hello!"**,** delegate**(**string mes**)  
 {** Console**.**WriteLine**(**mes**);  
 });** Console**.**Read**();  
}**static void ShowMessage**(**string mes**,** MessageHandler handler**)  
{** handler**(**mes**);  
}**

# анонимные методы

То есть Делегаты указывали на методы какие то объявленные, а сейчас мы их сразу прописываем.

delegate void MessageHandler**(**string message**);**static void Main**(**string**[]** args**)  
{** MessageHandler handler = delegate**(**string mes**)  
 {** Console**.**WriteLine**(**mes**);  
 };** handler**(**"hello world!"**);  
}**

Другой пример анонимных методов - передача в качестве аргумента для параметра, который представляет делегат:

delegate void MessageHandler**(**string message**);**static void Main**(**string**[]** args**)  
{** ShowMessage**(**"hello!"**,** delegate**(**string mes**)  
 {** Console**.**WriteLine**(**mes**);  
 });** Console**.**Read**();  
}**static void ShowMessage**(**string mes**,** MessageHandler handler**)  
{** handler**(**mes**);  
}**

# лямбды

упрощенная запись анонимных методов

Operation del = delegate**(**int x**,** int y**)  
 {** Console**.**WriteLine**(**"not lambda"**);** return x + y**;  
 };** Operation del2 = **(**int x**,** int y**)** =>  
 **{** Console**.**WriteLine**(**"lambda"**);** return x + y**;  
 };** int u = del**(**4**,** 5**);  
same -** int u2 = del2**(**4**,** 5**);**

Ну и есесна мы их тоже можем ккак параметр передавать

delegate int Operation**(**int x**);**int DoSomethingWithX**(**int x**,** Operation operation**)  
{** return operation**(**x**);  
}**public int Do**()  
{** int val = DoSomethingWithX**(**5**, (**int x**)** =>  
 **{** Console**.**WriteLine**(**x**);** return x+5**;  
 });** return val**;  
}**

# event’ы

Event – Штука торчащая из класса, поле его, которая говорит какие события произошли в классе

Делегат вызывается внутри класса, а что будет происходить когда он вызывается с этим навешивается снаружи

## prac

Это все как в js, событие и оно летит к n subscribers.

По сути это те же указатели на функцию

delegate void OperationMade**(); делегат**

private event OperationMade Made**; сделали эвент**

Made += **()** => Console**.**WriteLine**(**"operation made"**); добавили указатель**

Made?**.**Invoke**(); вызвали**

**Обработчики событий**

Обработчики событий - это именно то, что выполняется при вызове событий.

Made += **()** => Console**.**WriteLine**(**"operation made"**); любой метод**

В примерах на метаните просто все объявили, описали несколько методов и запиънули вызов метода в функцию и все

class Account  
**{** public delegate void AccountHandler**(**string message**); делегат** public event AccountHandler Notify**;** *// сделали эвент* public Account**(**int sum**)  
 {** Sum = sum**;  
 }** public int Sum **{** get**;** private set**;}** public void Put**(**int sum**)   
 {** Sum += sum**;** Notify?**.**Invoke**(**$"На счет поступило: {sum}"**);***// 2.Вызов события* **}** public void Take**(**int sum**)  
 {** if **(**Sum >= sum**)  
 {** Sum -= sum**;** Notify?**.**Invoke**(**$"Со счета снято: {sum}"**);** *// 2.Вызов события* **}** else  
 **{** Notify?**.**Invoke**(**$"Недостаточно денег на счете. Текущий баланс: {Sum}"**); ;  
 }  
 }  
}**class Program  
**{** static void Main**(**string**[]** args**)  
 {** Account acc = new Account**(**100**);** acc**.**Notify += DisplayMessage**;** *// запихнули в эвент(делегат) метод* acc**.**Notify += DisplayRedMessage**;** *// еще один* acc**.**Put**(**20**);** *вызываем метод в котором вызывается эвент* acc**.**Notify -= DisplayRedMessage**;** *// потом удалили*

acc**.**Put**(**300**);**  **}** private static void DisplayMessage**(**string message**)  
 {** Console**.**WriteLine**(**message**);  
 }** private static void DisplayRedMessage**(**String message**)  
 {** *// Устанавливаем красный цвет символов* Console**.**ForegroundColor = ConsoleColor**.Red;** Console**.**WriteLine**(**message**);** *// Сбрасываем настройки цвета* Console**.**ResetColor**();  
 }  
}**

**Ковариантность и контравариантность делегатов**

Так можно: выше объявлен class Client : Person { }

delegate void ClientInfo**(**Client client**);**static void Main**(**string**[]** args**)  
{** ClientInfo clientInfo = GetPersonInfo**;** *// контравариантность* Client client = new Client**{**Name = "Alice"**};** clientInfo**(**client**);** Console**.**Read**();  
}**private static void GetPersonInfo**(**Person p**)  
{** Console**.**WriteLine**(**p**.**Name**);  
}**

и еще так:

delegate Person PersonFactory**(**string name**);**static void Main**(**string**[]** args**)  
{** PersonFactory personDel**;** personDel = BuildClient**;** *// ковариантность* Person p = personDel**(**"Tom"**);** Console**.**WriteLine**(**p**.**Name**);**    
 Console**.**Read**();  
}**private static Client BuildClient**(**string name**)  
{** return new Client **{**Name = name**};  
}**

То есть здесь делегат возвращает объект Person. Однако благодаря ковариантности данный делегат может указывать на метод, который возвращает объект производного типа, например, Client.

## Theory

Event – Штука торчащая из класса, поле его, которая говорит какие события произошли в классе

https://github.com/praused/DelegatesAndEvents



Часто на event вешают имя …handler

Под капотом когда у нас есть event and delegate, there is class MulticlassDelegate, он отслеживает что куда идет

### Calling the event

Собстна мы все объявляем

public delegate int WorkPerfomedDelegate**(**int hours**,** WorkType workType**);**

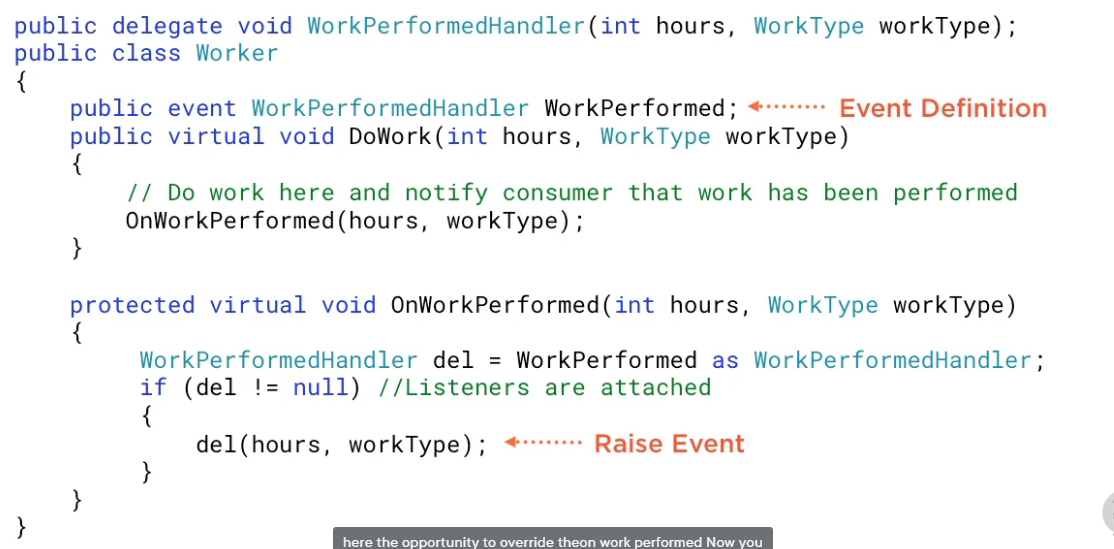
public event WorkPerfomedDelegate WorkPerfomed**;**public event EventHandler WorkCompleted**;**

и потом вызываем как метод, проверяя на null, **EventHandler**-это встроенная штука, то есть можем хоть так хоть так



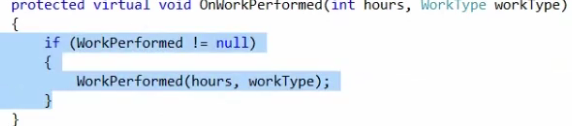


Делаем функцию проверки на null для эвентов

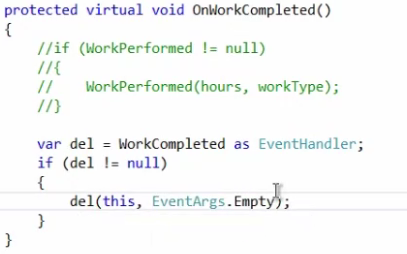


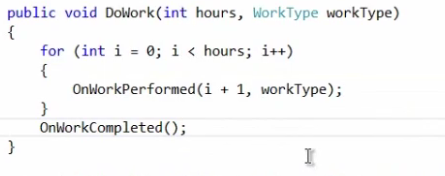
OnWorkPerfomed - Типо когда работа закончена мы отправляем эвент.

Его можно было сделать так еще:



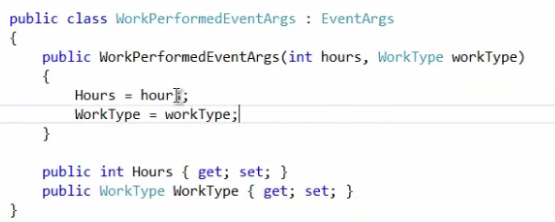
Для EventHandler(встроенная) это



Ну и вызываем мы эвенты так: 

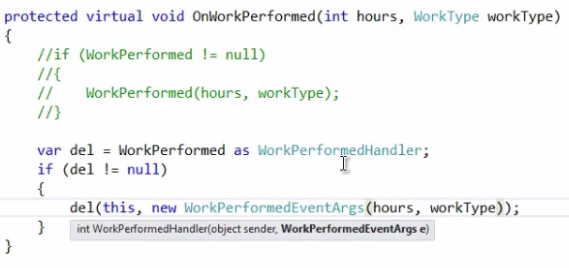
### ActionHandler

Вообще делать эвент с параметрами int hours, Worktype не есть хорошо, поэтому переделаем под стандарт





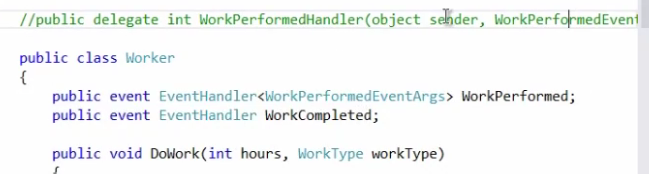
Пофиксим юзез



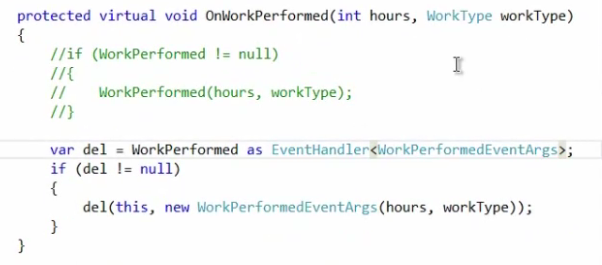
*//Чтобы не писать длинную строчку до этого:*



Можем сделать так:

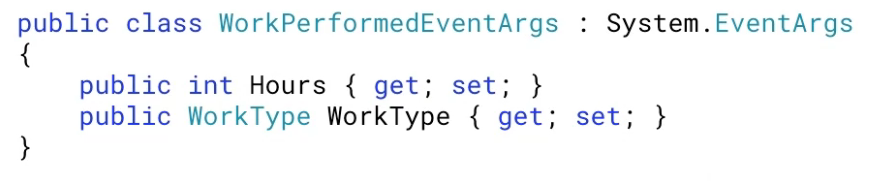


Фиксим…



## EventArgs

Это дата которую отправляет event, наследуют от системного



Чтобы юзать EventHandler(System) нам нужно описать что мы отправляем



Теперь можно сокращенно объявить эвент, без делегата

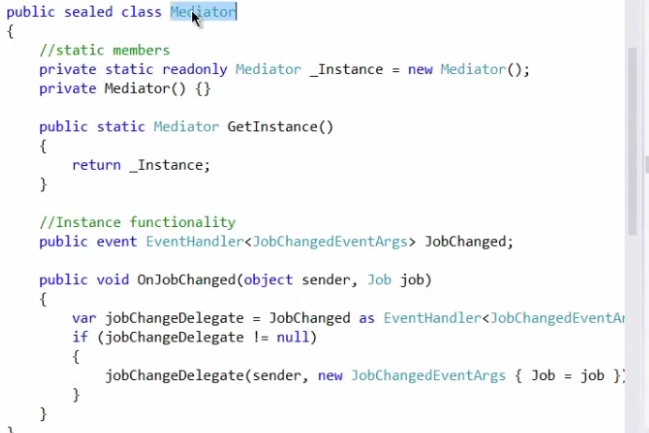


тогда нам не нужна строка

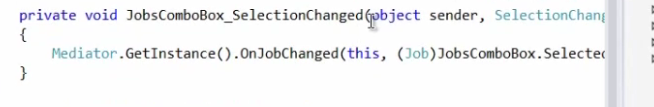


## Events and delegates in action

Mediator.cs -посредник между компонентами приложения, мы просто говорим что есть эвент и функция вызывающая его

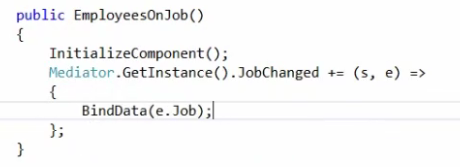


Jobs.xaml – форма, там выпадающий список и на изменение мы вызываем OnJobChanged



EmployesList.xaml – здесь нам надо отобразить выбранного сотрудника

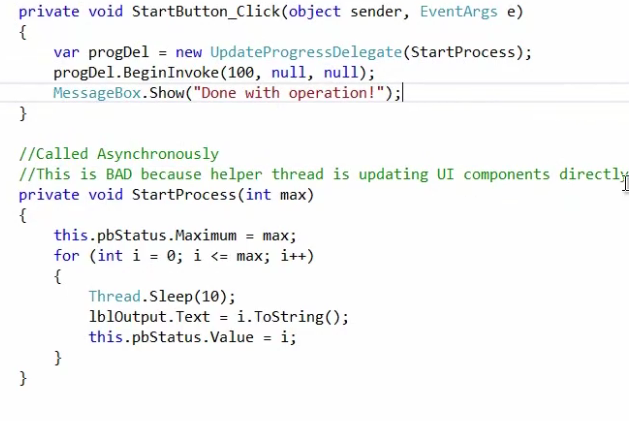
Поэтому мы забираем EventArgs



Здесь мы берем event(delegate) и прикрепляем ему метод, причем s,e это sender, EventArgs которые передаются из эвента

### Async Delegates – bad





progDel - Делегат указывающий на startProcess

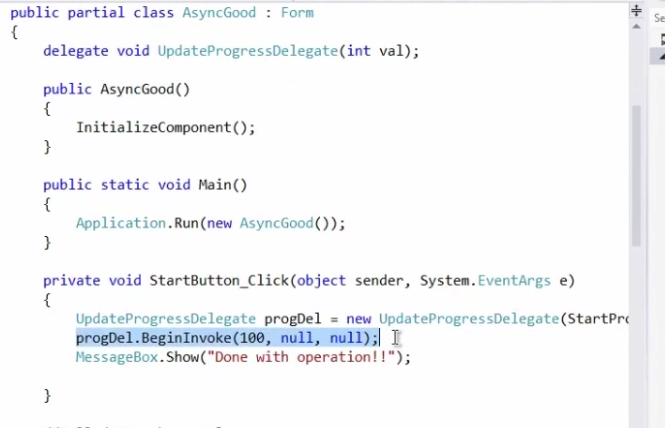
progDel.BeginInvoke(val, callback, object) -ззапускает параллельный поток

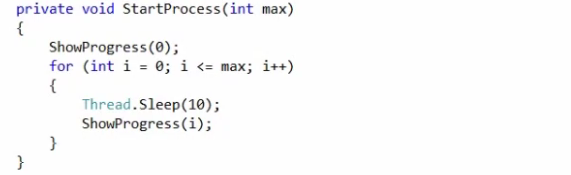
Это не сработает потому что GUI and BeginInvoke – разные потоки, и мы передаем дату в GUI из другого потока

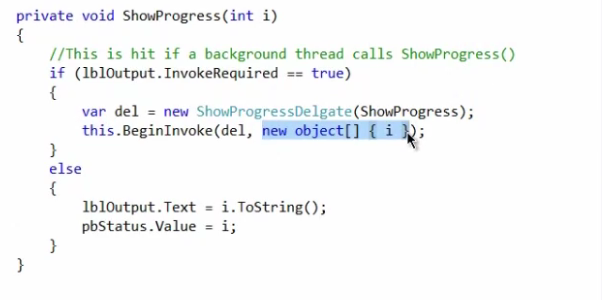
 что не разрешено

### Async Delegates – good

edited







BeginInvoke(Delegate del, params object[] args) – вызывает делегат в контексте this(FORMS)

-Итак, начинаем, запускаем progdel.BeginInvoke(StartProcess), он стартует параллельно GUI Thread, и мы как бы хотим в будущем передать в ГУИ поток дату(100)

-вызывается StartProcess(100), где вызывается ShowProgress(1-100), это все в потоке progdel.BeginInvoke(StartProcess), приходим в ShowProgress

-InvokeRequired делает проверку, в каком контексте работает текущий код. InvokeRequired = true, если контекст чужого потока (не потока GUI). Получается в else – только поток GUI

Теперь у нас идет поток progdel.BeginInvoke(StartProcess), и мы вызываем ShowProgress в контексте this(Forms):

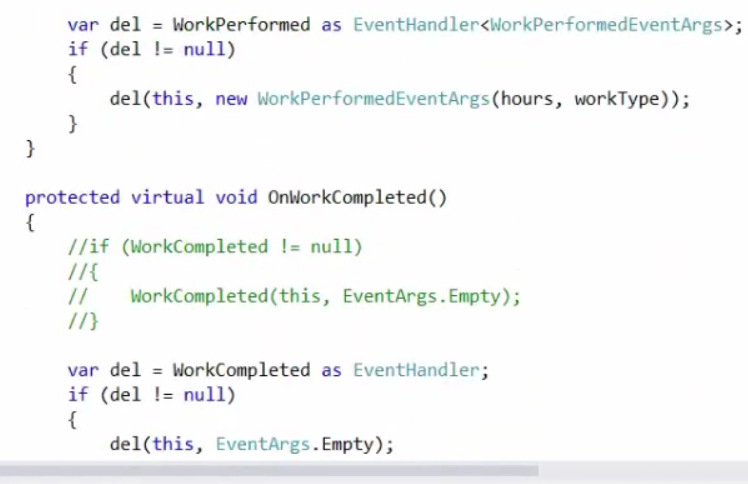
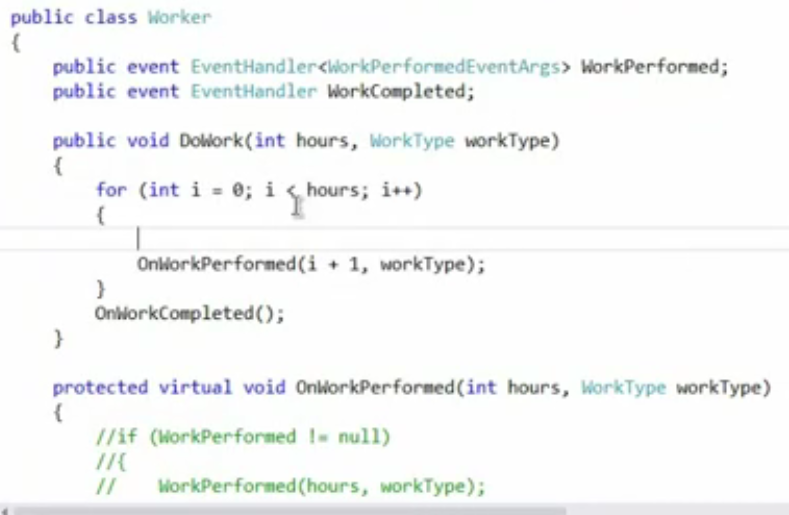


Только в другом потоке вызываем ShowProgress, и там уже нормально меняем значения

Но если честно я все равно не очень понял, мы типо сначала передали i а потом перевызвали в другом потоке, то есть поток progdel.BeginInvoke(StartProcess) донес в своем потоке i вызвал метод и потом вызвал метод в другом потоке, верно?

# Handling Events

Еще раз что у нас есть:

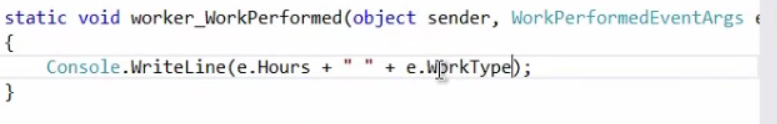
 

Теперь мы хотим прикрепить эти эвенты к handler-ам, пишем:

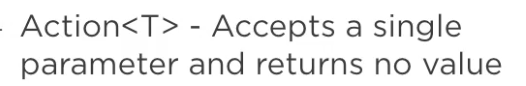
Main:

workerPerfomed)

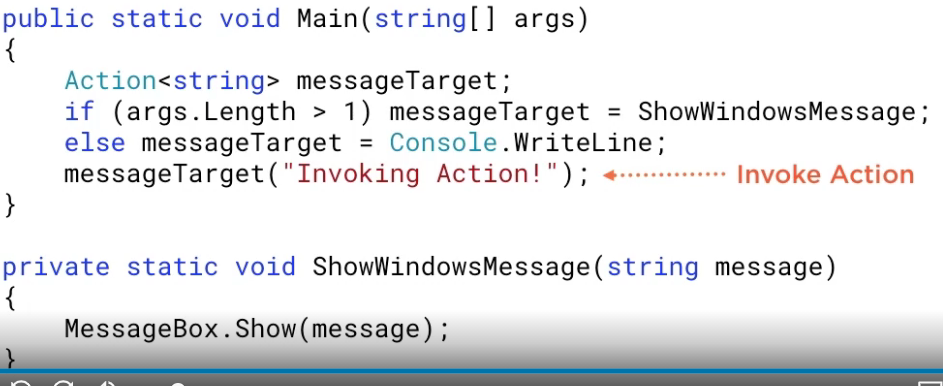




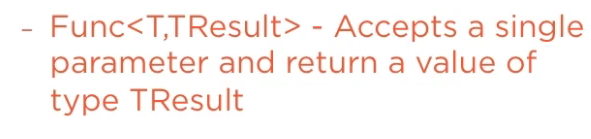
# Action<T>



Все просто, такой же делегат



# Func<T, TResult>



Тоже самое что и экшн, только не void



# Predicate

Тоже самое только возращает булеран

Predicate<int> isPositive = delegate (int x) { return x > 0; };

# Интерфейсы

Интерфейсы – это соглашение что у класса будут эти поля

* Стоит отметить, что если интерфейс имеет приватные методы и свойства (то есть с модификатором private), то они должны иметь реализацию по умолчанию.

Причем можно только const/static

private const int minArea = 0**;**public static int maxArea = 60**;**

* поддерживают множественное наследование

myClass: myInterface1, myInterface2, myInterface3, ...

* Если у метода класса и у метода интерфейса одинаковые имена, то можно юзать **explicit implementation**:
* class ExplicitInterfaceExample : ISaveble  
  **{** void ISaveble**.**ExplicitlySave**()  
   {** Console**.**WriteLine**(**"this is explicitly Save, it is belong to interface"**);  
   }** public void ExplicitlySave**()  
   {** Console**.**WriteLine**(**"this is explicitly Save, it is belong to class"**);  
   }** public void UsualSave**()  
   {** Console**.**WriteLine**(**"We didnt it, cuz we dont want method with same name" +  
   "(that's all what we want from explitily implementation)"**);  
   }** public static void LocalMain**()  
   {** var game = new ExplicitInterfaceExample**();** game**.**ExplicitlySave**();***//usual* **((**ISaveble**)**game**).**ExplicitlySave**();***//explicitly* Console**.**Read**();  
   }  
  }**
* Когда мы говорим что:

IEnumerable<Square> squares = new**[] {** new Square**(**1**,** 1**,** 1**),** new Square**(**2**,** 2**,** 2**),** new Square**(**3**,** 3**,** 3**) };**foreach **(**var item in squares**)  
{** Console**.**WriteLine**(**item**.**LenOfSide**);  
}**

Когда мы объявляем интерфейс, мы просто говорим что это штука у которой есть такие то методы, и других не видим, здесь мы имеем доступ к полям Square, потому что мы создаем объекты с типом Т

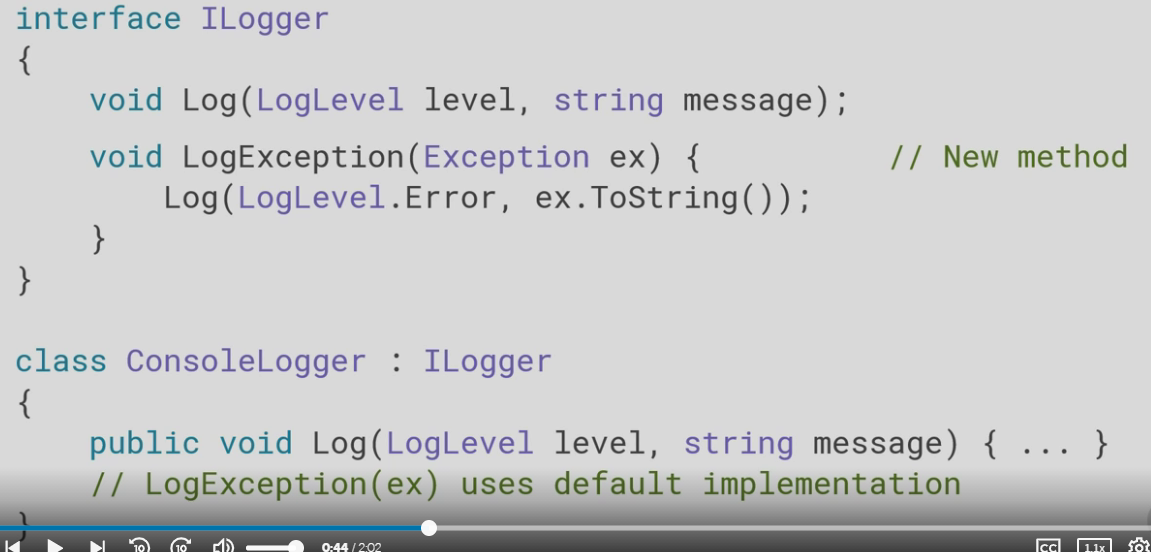
interface IEnumerator<out T> : IDisposable**,** IEnumerator  
**{** T Current **{** get**; }  
}**

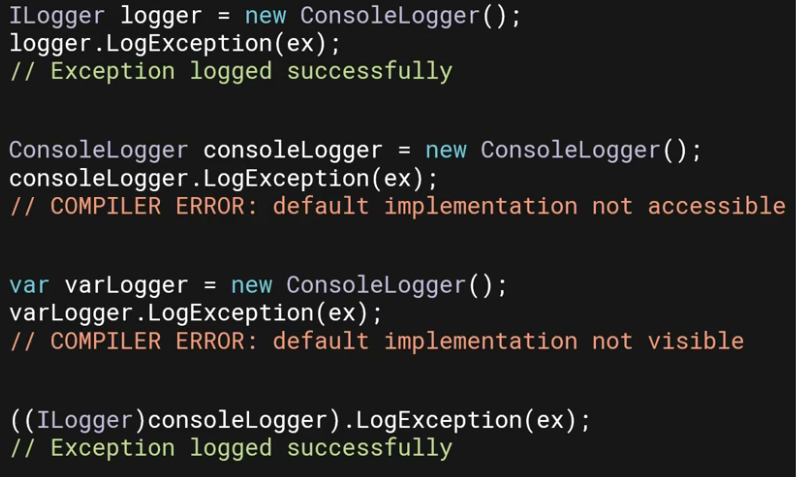
Если напишем

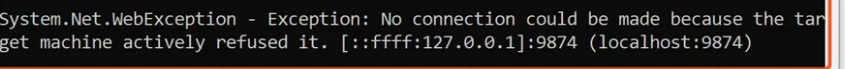
IEnumerable squares = new[] { new Square(1, 1, 1), new Square(2, 2,2)};

То он ограничит до интерфейса IEnumerable и мы не будем иметь доступа к полям Square.

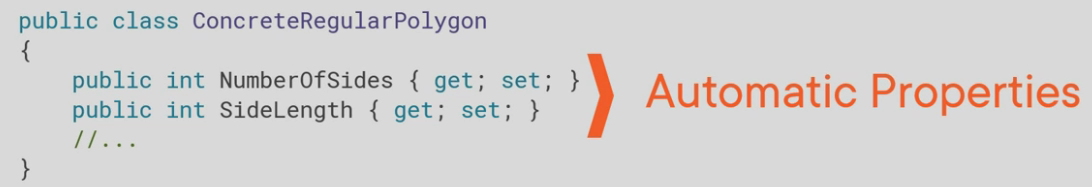
* Default implementation



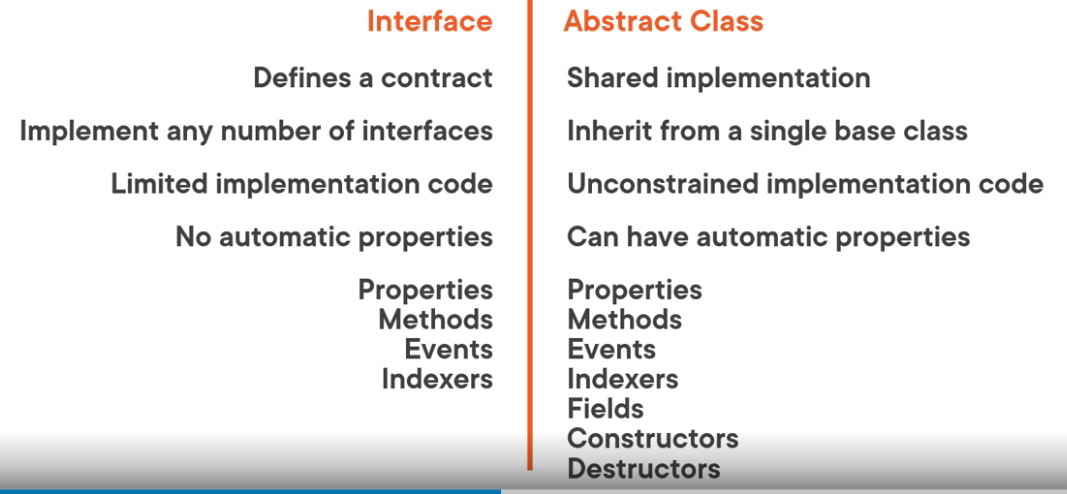


С ними надо аккуратно, потому что если мы напишем в логгере Console.log… то потом если мы будем юзать его в Web application все скрашится потому что 

* Когда мы прописываем свойства в интерфейсе, мы просто говорим что они должны быть и все, это не авто свойство, вообще это не особо смысл имеет

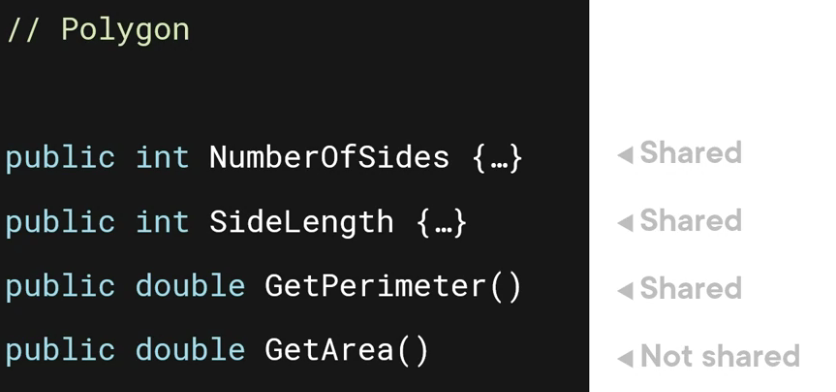


* Interface vs Abstract class



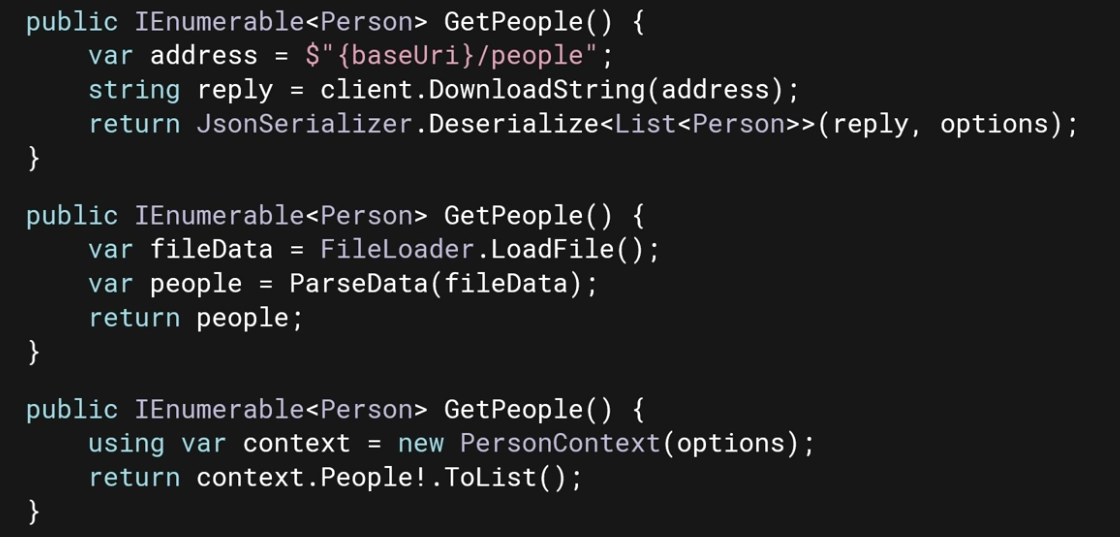
Когда не знаешь что сделать задай вопрос, как много общего кода для наследников? Если много то класс если нет то интерфейс, например

Shared = общий у всех код



Поэтому абстрактный класс

Для ДАТА ридеров



Много разного хоть и названия функций одинаковые, поэтому интерфейсы

# Exceptions – какие обрабатывать



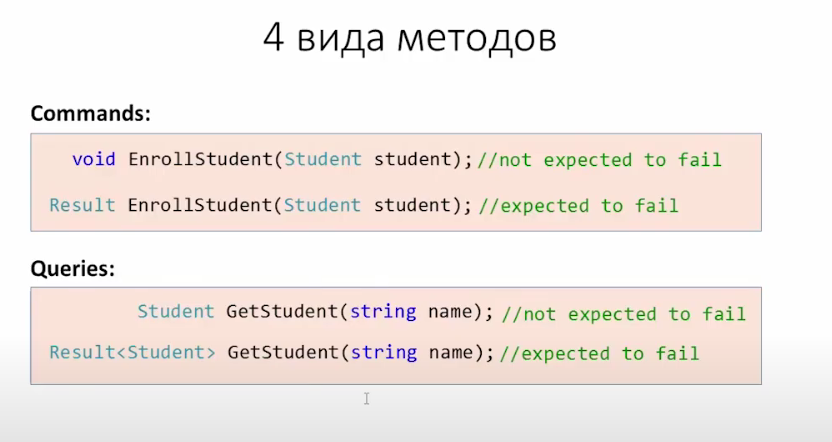
Системные невостанавливаемые – пример стэковерфлоу, ну поймаешь ты. Но это баг по сути и надо исправлять а не эксепшоны кидать

Системные восстанавливаемые – Socket exception, IOException, SecurityException, их надо обрабатывать, сокеты те же могут каждые 30 секунд появляться, если каждый раз крашиться то это плохо))))

Пользовательские восстанавливаемые – валидация какие то

--

Часто делают так, и пишут OnSuccess(Func<Result, T>, …)



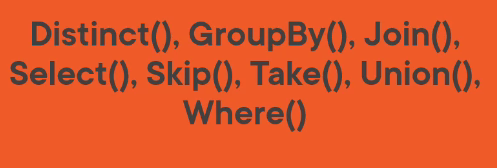
# LINQ

## Deferred and immediate types

Когда пишем линки некоторые операции вычисляются отложено, а некоторые сразу, а некоторые в потоке



### Streaming



Пример:



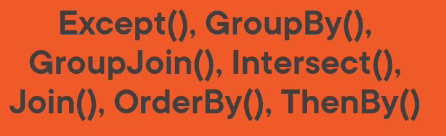
После query=… там еще все еще ничего не лежит, только когда потребуется возьмется

Запрос, представленный этим методом, не выполняется до тех пор, пока объект не будет перечислен либо путем прямого вызова его метода GetEnumerator, либо с помощью foreach

Он работает через yield

### Non-streaming

Вся коллекция читается до того как резалт получен



+

Aggregate Functions – Count, Average, Min, Max, Sum  
Element Operators – First, Last, SingleToList, ToArray, ToDictionary

Считается все вместе, не через yield и птом возвращается, как в примере:

public static IEnumerable<T> FilterSimple<T>**(**this IEnumerable<T> source**,** Func<T**,** bool> predicate**)  
{** var result = new List<T>**();** foreach **(**var item in source**)  
 {** if **(**predicate**(**item**))  
 {** result**.**Add**(**item**);  
 }  
 }** return result**;  
}**

То есть, если в примере выше, streaming заюзать после ордербай то бон все вместе вернет, а не через yield, а потом к нему юзанет where

//Filter=where



# Thread

Thread

Threadpool

Task



monitor

Mutex

Semaphore

ConcurentDictionary

MultiReadSingleWrite

## Props

* Статическое свойство **CurrentContext** позволяет получить контекст, в котором выполняется поток
* Статическое свойство **CurrentThread** возвращает ссылку на выполняемый поток
* Свойство **IsAlive** указывает, работает ли поток в текущий момент
* Свойство **IsBackground** указывает, является ли поток фоновым
* Свойство **Name** содержит имя потока
* Свойство **Priority** хранит приоритет потока - значение перечисления ThreadPriority
* Свойство **ThreadState** возвращает состояние потока - одно из значений перечисления ThreadState

## Methods

* Статический метод **GetDomain** возвращает ссылку на домен приложения
* Статический метод **GetDomainID** возвращает id домена приложения, в котором выполняется текущий поток
* Статический метод **Sleep** останавливает поток на определенное количество миллисекунд
* Метод **Abort** уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно. Для проверки завершенности потока следует опрашивать его свойство ThreadState
* Метод **Interrupt** прерывает поток, который находится в состоянии WaitSleepJoin
* Метод **Join** блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
* Метод **Start** запускает поток

## Status

## Статусы потока содержатся в перечислении **ThreadState**:

* **Aborted**: поток остановлен, но пока еще окончательно не завершен
* **AbortRequested**: для потока вызван метод Abort, но остановка потока еще не произошла
* **Background**: поток выполняется в фоновом режиме
* **Running**: поток запущен и работает (не приостановлен)
* **Stopped**: поток завершен
* **StopRequested**: поток получил запрос на остановку
* **Suspended**: поток приостановлен
* **SuspendRequested**: поток получил запрос на приостановку
* **Unstarted**: поток еще не был запущен
* **WaitSleepJoin**: поток заблокирован в результате действия методов Sleep или Join

## Контент

Через         Thread myThread = new Thread(new ThreadStart(Count));

Мы можем вызвать метод в новом потоке, new ThreadStart(Count) это делегат, но можно вызывать просто new Thread(Count);

если нам надо передать какие-нибудь параметры в поток?

### ParameterizedThreadStart

        Thread myThread = new Thread(new ParameterizedThreadStart(Count));

        myThread.Start(number);

мы можем запускать во втором потоке только такой метод, который в качестве единственного параметра принимает объект типа object

есть одно ограничение: метод Thread.Start не является типобезопасным, то есть мы можем передать в него любой тип, и потом нам придется приводить переданный объект к нужному нам типу. Для решения данной проблемы рекомендуется объявлять все используемые методы и переменные в специальном классе, а в основной программе запускать поток через ThreadStart.

#### пример:

class Program  
 **{** static void Main**(**string**[]** args**)  
 {** Counter counter = new Counter**(**5**,** 4**);** Thread myThread = new Thread**(**new ThreadStart**(**counter**.**Count**));** myThread**.**Start**();** *//........................* **}  
 }** public class Counter  
 **{** private int x**;** private int y**;** public Counter**(**int \_x**,** int \_y**)  
 {** this**.**x = \_x**;** this**.**y = \_y**;  
 }** public void Count**()  
 {** for **(**int i = 1**;** i < 9**;** i++**)  
 {** Console**.**WriteLine**(**"Второй поток:"**);** Console**.**WriteLine**(**i \* x \* y**);** Thread**.**Sleep**(**400**);  
 }  
 }  
 }**

### lock(obj)

шобы один поток работал а другие ждали есть лок

#### пример

class Program  
**{** static int x = 0**;** static object locker = new object**();** static void Main**(**string**[]** args**)  
 {** for **(**int i = 0**;** i < 5**;** i++**)  
 {** Thread myThread = new Thread**(**Count**);** myThread**.**Name = "Поток " + i**.**ToString**();** myThread**.**Start**();  
 }** Console**.**ReadLine**();  
 }** public static void Count**()  
 {** lock **(**locker**)  
 {** x = 1**;** for **(**int i = 1**;** i < 9**;** i++**)  
 {** Console**.**WriteLine**(**"{0}: {1}"**,** Thread**.**CurrentThread**.**Name**,** x**);** x++**;** Thread**.**Sleep**(**100**);  
 }  
 }  
 }  
}**

### Monitor

Предоставляет механизм для синхронизации доступа **к объектам. Блокировщик объектов кратко если.**

#### Props

##### LockContentionCount

Возвращает значение, указывающее, сколько раз возникало состязание при попытке установить блокировку монитора.

#### Methods:

##### Enter

Получает эксклюзивную блокировку(запрещает другим транзакциям блокировать строку) указанного объекта.

##### Enter(Object, Boolean)

Получает монопольную(=эксклюзивную) блокировку указанного объекта и единым блоком задает значение, указывающее, была ли выполнена блокировка.

##### Exit

Освобождает эксклюзивную блокировку указанного объекта.

##### IsEntered

Определяет, содержит ли текущий поток блокировку указанного объекта.

##### Pulse

Уведомляет поток в очереди готовности об изменении состояния объекта с блокировкой.

Поток, который в настоящее время владеет блокировкой указанного объекта, вызывает этот метод, чтобы передать следующий поток в строке для блокировки. При получении пульса ожидающий поток перемещается в очередь готовности. Когда вызывающий поток Pulse освобождает блокировку, следующий поток в очереди готовности (который не обязательно является потоком, который был импульсным) получает блокировку.

##### PulseAll

Уведомляет все ожидающие потоки об изменении состояния объекта.

##### TryEnter

Пытается получить монопольную блокировку указанного объекта и единым блоком задает значение, указывающее, была ли выполнена блокировка. Мб некотерое время(параметры)

##### Wait

Освобождает блокировку объекта и блокирует текущий поток до тех пор, пока тот не получит блокировку снова. Мб некотерое время(параметры)

#### Пример 1

В примере создается десять задач, каждый из которых асинхронно выполняется в потоке пула потоков. Каждая задача создает 10 000 случайных чисел, вычисляет их среднее значение и обновляет две переменные уровня процедуры, которые сохраняют общую сумму количества созданных случайных чисел и их сумму. После выполнения всех задач эти два значения затем используются для вычисления общего значения.

#### Пример 2

Фактически конструкция оператора lock из прошлой темы инкапсулирует в себе:

Try-finally, зачем? Потому что если во время кода после Enter возникнет исключение то все равно надо снять блокировку

**{** class Program  
 **{** static int x = 0**;** static object locker = new object**();** static void Main**(**string**[]** args**)  
 {** for **(**int i = 0**;** i < 5**;** i++**)  
 {** Thread myThread = new Thread**(**Count**);** myThread**.**Name = $"Поток {i**.**ToString**()**}"**;** myThread**.**Start**();  
 }** Console**.**ReadLine**();  
 }** public static void Count**()  
 {** bool acquiredLock = false**;** try  
 **{** Monitor**.**Enter**(**locker**,** ref acquiredLock**);** x = 1**;** for **(**int i = 1**;** i < 9**;** i++**)  
 {** Console**.**WriteLine**(**$"{Thread**.**CurrentThread**.**Name}: {x}"**);** x++**;** Thread**.**Sleep**(**100**);  
 }  
 }** finally  
 **{** if **(**acquiredLock**)** Monitor**.**Exit**(**locker**);  
 }  
 }  
 }**

Monitor.Enter принимает два параметра - объект блокировки и значение типа bool, которое указывает на результат блокировки (если он равен true, то блокировка успешно выполнена). Фактически этот метод блокирует объект locker так же, как это делает оператор lock. С помощью А в блоке try...finally с помощью метода Monitor.Exit происходит освобождение объекта locker, если блокировка осуществлена успешно, и он становится доступным для других потоков.

### AutoResetEvent

Класс AutoResetEvent также служит целям синхронизации потоков

позволяет переключить данный объект-событие из сигнального в несигнальное состояние.

#### Пример

Так, пример из предыдущей темы мы можем переписать с использованием AutoResetEvent следующим образом:

class Program  
 **{** static AutoResetEvent waitHandler = new AutoResetEvent**(**true**);** static int x = 0**;** static void Main**(**string**[]** args**)  
 {** for **(**int i = 0**;** i < 5**;** i++**)  
 {** Thread myThread = new Thread**(**Count**);** myThread**.**Name = $"Поток {i**.**ToString**()**}"**;** myThread**.**Start**();  
 }** Console**.**ReadLine**();  
 }** public static void Count**()  
 {** waitHandler**.**WaitOne**();** x = 1**;** for **(**int i = 1**;** i < 9**;** i++**)  
 {** Console**.**WriteLine**(**$"{Thread**.**CurrentThread**.**Name}: {x}"**);** x++**;** Thread**.**Sleep**(**100**);  
 }** waitHandler**.**Set**();  
 }  
 }**

Если у нас в программе используются несколько объектов AutoResetEvent, то мы можем использовать для отслеживания состояния этих объектов методы **WaitAll** и **WaitAny**, которые в качестве параметра принимают массив объектов класса WaitHandle - базового класса для AutoResetEvent.

Так, мы тоже можем использовать WaitAll в вышеприведенном примере. Для этого надо строку waitHandler.WaitOne(); заменить на следующую:

|  |  |
| --- | --- |
| 1 | AutoResetEvent.WaitAll(new WaitHandle[] {waitHandler}); |

### Mutex

Еще один инструмент управления синхронизацией потоков представляет класс **Mutex**, также находящийся в пространстве имен System.Threading. Данный класс является классом-оболочкой над соответствующим объектом ОС Windows "мьютекс".

#### пример

Перепишем пример из прошлой темы, используя мьютексы:

class Program  
**{** static Mutex mutexObj = new Mutex**();** static int x = 0**;** static void Main**(**string**[]** args**)  
 {** for **(**int i = 0**;** i < 5**;** i++**)  
 {** Thread myThread = new Thread**(**Count**);** myThread**.**Name = $"Поток {i}"**;** myThread**.**Start**();  
 }** Console**.**ReadLine**();  
 }** public static void Count**()  
 {** mutexObj**.**WaitOne**( );** x = 1**;** for **(**int i = 1**;** i < 9**;** i++**)  
 {** Console**.**WriteLine**(**$"{Thread**.**CurrentThread**.**Name}: {x}"**);** x++**;** Thread**.**Sleep**(**100**);  
 }** mutexObj**.**ReleaseMutex**();  
 }  
}**

### semaphore

Семафоры позволяют ограничить доступ определенным количеством объектов.

#### пример

Например, у нас такая задача: есть некоторое число читателей, которые приходят в библиотеку три раза в день и что-то там читают. И пусть у нас будет ограничение, что единовременно в библиотеке не может находиться больше трех читателей. Данную задачу очень легко решить с помощью семафоров:

using System**;**using System**.**Threading**;**namespace SemaphoreApp  
**{** class Program  
 **{** static void Main**(**string**[]** args**)  
 {** for **(**int i = 1**;** i < 6**;** i++**)  
 {** Reader reader = new Reader**(**i**);  
 }** Console**.**ReadLine**();  
 }  
 }** class Reader  
 **{** *// создаем семафор* static Semaphore sem = new Semaphore**(**3**,** 3**);** Thread myThread**;** int count = 3**;** *// счетчик чтения* public Reader**(**int i**)  
 {** myThread = new Thread**(**Read**);** myThread**.**Name = $"Читатель {i**.**ToString**()**}"**;** myThread**.**Start**();  
 }** public void Read**()  
 {** while **(**count > 0**)  
 {** sem**.**WaitOne**();** Console**.**WriteLine**(**$"{Thread**.**CurrentThread**.**Name} входит в библиотеку"**);** Console**.**WriteLine**(**$"{Thread**.**CurrentThread**.**Name} читает"**);** Thread**.**Sleep**(**1000**);** Console**.**WriteLine**(**$"{Thread**.**CurrentThread**.**Name} покидает библиотеку"**);** sem**.**Release**();** count--**;** Thread**.**Sleep**(**1000**);  
 }  
 }  
 }  
}**

Для создания семафора используется класс **Semaphore**: static Semaphore sem = new Semaphore(3, 3);. Его конструктор принимает два параметра: первый указывает, какому числу объектов изначально будет доступен семафор, а второй параметр указывает, какой максимальное число объектов будет использовать данный семафор. В данном случае у нас только три читателя могут одновременно находиться в библиотеке, поэтому максимальное число равно 3.

Основной функционал сосредоточен в методе Read, который и выполняется в потоке. В начале для ожидания получения семафора используется метод sem.WaitOne(). После того, как в семафоре освободится место, данный поток заполняет свободное место и начинает выполнять все дальнейшие действия. После окончания чтения мы высвобождаем семафор с помощью метода sem.Release(). После этого в семафоре освобождается одно место, которое заполняет другой поток.

А в методе Main нам остается только создать читателей, которые запускают соответствующие потоки.