Плывите сосиски (сосите плывиски))))

[**===1. Системы контроля версий. Классификация. Git, основные возможности**](#_7amqgnhnumnw) **5**

[**===2. Состав. NET Framework. Структура среды выполнения CLR.**](#_274wg0q08mu4) **7**

[**===3. Структура управляемого модуля - portable executable (PE). Понятие и исполнение сборки. CIL.**](#_7tvslk3ztz37) **8**

[**===4. CTS (Common Type System). Типы данных C#. Ссылочные и типы значений.**](#_xgzgdd6jsabh) **9**

[**===5. Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, null-объединение**](#_3882iqo8upx) **11**

[**===6. Тип данных String: операции, литералы, пустые и нулевые строки, форматированный вывод.**](#_h6bohmrzr536) **13**

[**===7. Неявная типизация – назначение и использование.**](#_8sgya1fc66n) **16**

[**===8. Массивы C# одномерные, прямоугольные и ступенчатые.**](#_bnh17vquy2nd) **17**

[**===9. Понятие кортежей. Свойства, создание**](#_rweq1rux0lyi) **18**

[**===10. Принципы объектно-ориентированного программирования.**](#_7rzj8rakmwfi) **19**

[**===11. Класс. Элементы класса. Свойства и индексаторы.**](#_otg05e818ij8) **20**

[**===12. Класс. Константы. Поля только для чтения. Инициализаторы класса.**](#_uni0yd1f0j2) **22**

[**===13. Спецификаторы доступа C#. Видимость типов. Доступ к членам типов.**](#_5kxncni0hefs) **22**

[**===14. Класс. Конструкторы и их свойства. Деструкторы**](#_ywnpeel3ri9s) **24**

[**===15. Класс и методы System.Object.**](#_57bzabi2jrq) **25**

[**===16. Статические методы и статические конструкторы класса.**](#_71yb8lmg42hk) **26**

[**===17. Статические классы. Методы расширения и правила их определения.**](#_qv9nofwk0t69) **27**

[**===18. Анонимные типы.**](#_60qkucfr2yl5) **28**

[**===19. Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы.**](#_1u69n72sibuu) **29**

[**===20. Перегрузка методов и операторов. Правила перегрузки операторов.**](#_8miqlfn98c30) **31**

[**===21. Операции преобразования типа. Явная и неявная форма. Ограничения.**](#_9aobmmwrtqvk) **32**

[**===22. Вложенные типы. Вложенные объекты**](#_m0pq2d1s4obw) **34**

[**===23. Правила наследования C#.**](#_frerxjds3t6b) **35**

[**===24. Сокрытие имен при наследовании. Обращение к скрытым членам**](#_laeumnhr3wop) **36**

[**===25. Использование операций is и as**](#_h88s66qicypi) **37**

[**===26. Полиморфизм. Виртуальные методы, свойства и индексаторы. Правила переопределения.**](#_2sa4szet7kui) **38**

[**===27. Понятие раннего и позднего связывания.**](#_kdxxum623v4e) **39**

[**===28. Абстрактные классы и методы. Бесплодные классы.**](#_atk643nh56jv) **40**

[**===29. Структуры в C#.**](#_4b1h4s712ypy) **41**

[**===30. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов.**](#_d5ybnbgh036x) **43**

[**===31. Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы.**](#_thhomwlifctz) **44**

[**===32. Ковариантность интерфейсов. Контравариантность интерфейсов**](#_j96ls140y8wt) **46**

[**===33. Стандартные интерфейсы .NET. Назначение и применение.**](#_t48awy2ddmxv) **48**

[**===34. Исключительные ситуации. Генерация и повторная генерация исключений.**](#_ewec33w6eul4) **50**

[**===35. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений**](#_ndn4ndy97317) **51**

[**===36. Обобщения (generics). Свойства обобщений.**](#_gfngvgxe1tr1) **53**

[**===37. Концепция ограничений обобщений. Статические члены обобщений.**](#_w7qbmyvasi5y) **55**

[**===38. Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты.**](#_7zkw8kq8i699) **58**

[**===39. Анонимные функции. Лямбда-выражения.**](#_jr1ez2j2k1ns) **60**

[**===40. Обобщённые делегаты .NET. Action, Func, Predicate**](#_ckt8hlvf6d4) **62**

[**===41. События и делегаты.**](#_f1wf07wd8vjp) **63**

[**===42. Стандартные коллекции .NET. Типы коллекций.**](#_1u6hl884n2tp) **65**

[**===43. Стандартные интерфейсы коллекций.**](#_3tucywn9cyni) **67**

[**===44. IEnumerable и IEnumerator**](#_fi8us788c9bv) **69**

[**===45. LINQ to Objects. Синтаксис. Форма. Возврат результата. Грамматика выражений запросов. Отложенные и не отложенные операции.**](#_r63o2b7f2pm8) **69**

[**===46. LINQ to Objects. Операции Where, Select, Take, OrderBy, Join, GroupBy**](#_ubdve0sx3jgf) **73**

[**===47. Рефлексия. System Type.**](#_41ecczxgjq35) **77**

[**===48. Классы для работы с файловой системой.**](#_lhpsvmqqyxh8) **78**

[**===49. Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы.**](#_3zh4cicqfn0l) **80**

[**===50. Классы адаптеры потоков.**](#_4qy9nlw3s8wu) **82**

[**===51. Сериализация. Форматы сериализации.**](#_bl7a1epyecag) **85**

[**===52. Сериализация контрактов данных. интерфейс ISerializable.**](#_kmh5rs4e93o8) **88**

[**===53. Атрибуты. Создание собственного атрибута.**](#_8zkew264v0mm) **91**

[**===54. Процесс. Домен приложений. Поток выполнения.**](#_949jjdrjbsql) **93**

[**===55. Создание потоков , классы приоритетов. Состояния потоков**](#_hyvcaqg54ntl) **96**

[**===56. Синхронизация потоков. Lock. Monitor. Мutex. Semaphore**](#_ampxa5g702ix) **99**

[**===57. Библиотека параллельных задач TPL. Класс Task. Состояние задачи.**](#_oluv7xv50tyg) **104**

[**===58. Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения.**](#_xttpk6gc4db0) **106**

[**===59. Параллелизм при императивной обработке данных. Класс Parallel**](#_cis53sjzzzlj) **108**

[**===60. Асинхронные методы. async и await**](#_h9j54ilahosy) **111**

[**===61. Проектирование отношений. Агрегация, композиция и ассоциация**](#_ayhzpsv76cru) **112**

Вопросы

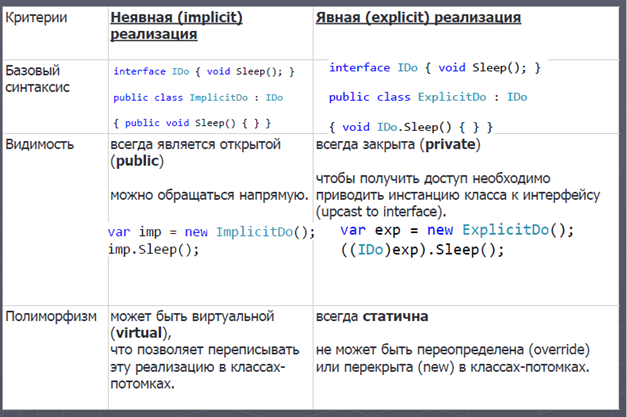
# ===31. Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы.

**Интерфейсы** - задается набор абстрактных методов, свойств, событий и индексаторов, которые должны быть реализованы в производных классах

[атрибуты] [спецификаторы] interface Iимя\_интерфейса [ : предки] Тело интерфейса[; ]

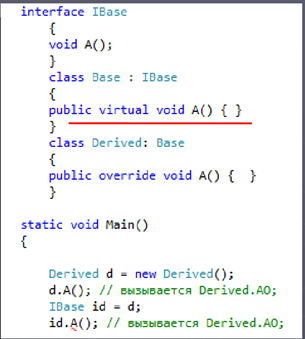
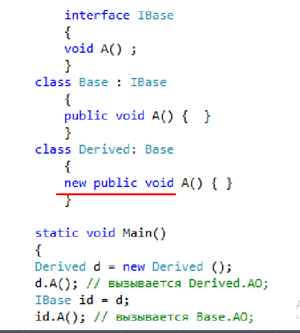
При реализации члена интерфейса имеется возможность указать его имя полностью вместе с именем самого интерфейса. В этом случае получается явная реализация члена интерфейса, или просто **явная реализация.**

Для явной реализации интерфейсного метода могут быть две причины. Во-первых, когда интерфейсный метод реализуется с указанием его полного имени, то такой метод оказывается доступным не посредством объектов класса, реализующего данный интерфейс, а по интерфейсной ссылке. Следовательно, явная реализация позволяет реализовать интерфейсный метод таким образом, чтобы он не стал открытым членом класса, предоставляющего его реализацию. И во-вторых, в одном классе могут быть реализованы два интерфейса с методами, объявленными с одинаковыми именами и сигнатурами. Но неоднозначность в данном случае устраняется благодаря указанию в именах этих методов их соответствующих интерфейсов.



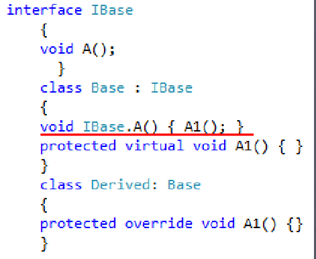
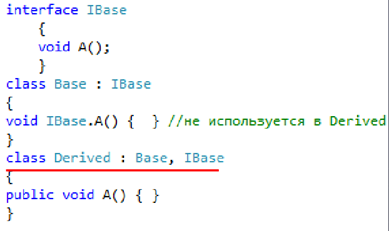
**Работа с объектами через интерфейсы (преобразования)**

Класс наследует все методы своего предка (интерфейсы). Он может переопределить ( new), но обращаться к ним - через объект класса. Если использовать для обращения ссылку на интерфейс, вызывается не переопределенная версия



Если интерфейс реализуется с помощью виртуального метода класса, после его переопределения в потомке любой вариант обращения (через класс или через интерфейс) приведет к одному и тому же результату

Метод интерфейса, реализованный явным указанием имени, объявлять виртуальным запрещается. При необходимости переопределить в потомках его поведение



Существует возможность повторно реализовать интерфейс, указав его имя в списке предков класса наряду с классом предком, уже реализовавшим этот интерфейс.

# ===32. Ковариантность интерфейсов. Контравариантность интерфейсов

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

Имеется три возможных варианта поведения:

* **Ковариантность:** позволяет использовать более конкретный тип, чем заданный изначально. Это средство, разрешающее методу возвращать тип, производный от класса, указанного в параметре типа.
* **Контравариантность:** позволяет использовать более универсальный тип, чем заданный изначально. Это средство, разрешающее методу использовать аргумент, тип которого относится к базовому классу, указанному в соответствующем параметре типа.
* **Инвариантность:** позволяет использовать только заданный тип

По умолчанию все обобщенные интерфейсы, например, IAccout<T> являются **инвариантными.**

Параметр **ковариантного типа** объявляется с помощью ключевого слова **out**, которое предваряет имя этого параметра.

// В этом обобщенном интерфейсе поддерживается ковариантность,

public interface IMyInfo<out Т> {

T GetObject();

}

В данном контексте ключевое слово out обозначает, что обобщенный тип T является ковариантным. А раз он ковариантный, то метод GetObject() может возвращать ссылку на обобщенный тип T или же ссылку на любой класс, производный от типа Т.

На применение ковариантности накладываются некоторые ограничения. Ковариантность параметра типа может распространяться только на тип, возвращаемый методом. Следовательно, ключевое слово out нельзя применять в параметре типа, служащем для объявления параметра метода. Ковариантность оказывается пригодной только для ссылочных типов. Ковариантный тип нельзя использовать в качестве ограничения в интерфейсном методе. Так, следующий интерфейс считается недопустимым:

public interface IMyCoVarGenIF2<out Т> {

void M<V>() where V:T; // Ошибка, ковариантный тип T нельзя использовать как ограничение }

Параметр **контравариантного типа** объявляется с помощью ключевого слова **in**, которое предваряет имя этого параметра.

// Это обобщенный интерфейс, поддерживающий контравариантность.

public interface IMyContraVarGenlFc<in Т> {

void Show(T obj);

}

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

Контравариантность оказывается пригодной только для ссылочных типов, а параметр контравариантного типа можно применять только к аргументам методов. Следовательно, ключевое слово in нельзя указывать в параметре типа, используемом в качестве возвращаемого типа.

# ===33. Стандартные интерфейсы .NET. Назначение и применение.

Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки.

**Сравнение объектов (интерфейс IComparable)**

Интерфейс IComparable содержит всего один метод CompareTo(),возвращающий результат сравнения двух объектов – текущего и переданного ему в качеству параметра:

interface IComparable

{

int CompareTo(object obj)

}

Метод должен возвращать:

* 0, если текущий объект и параметр равны
* отрицательное число, если текущий объект меньше параметра;
* положительное число, если текущий объект больше параметра.

**Сортировка объектов по различным критериям (интерфейс IComparer)**

Данный интерфейс определен в пространстве имен System.Collections. Он также содержит один метод Compare(),возвращающий результат сравнения двух объектов, переданных ему в качестве параметров.

interface IComparer

{

int Compare(object obj1,object obj2)

}

Принцип применения этого интерфейса состоит в том, что для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий данный интерфейс. Объект этого класса передается в стандартный метод сортировки массива в качестве второго аргумента.

**Клонирование объектов (интерфейс ICloneable)**

**Клонирование** - создание копии объекта. Копия объекта принято называть клоном. При присваивании одного экземпляра другому копируется ссылка, а не сам объект. В случае если крайне важно скопировать в другую область памяти поля объекта͵ можно воспользоваться методом MemberwiseClone(), который любой объект наследует от класса object. При этом объекты, на которые указывают поля объекта͵ в свою очередь являющиеся ссылками, не копируются. Это принято называть *поверхностным клонированием*.

Важно заметить, что для создания полностью независимых объектов крайне важно глубокое копирование, когда в памяти создается дубликат всего дерева объектов, то есть объектов, на которые ссылаются поля объекта͵ поля полей, и т.д. Алгоритм глубокого копирования сложен, требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.

Объект, имеющий собственные алгоритмы клонирования, должен объявляться как производный интерфейса ICloneable и переопределять его единственный метод Clone().

**Перебор объектов (интерфейс IEnumerable) и итераторы**

Оператор foreach является удобным средством перебора элементов объекта. Массивы и все стандартные коллекции библиотеки .NET позволяют выполнять такой перебор благодаря тому, что в них реализованы интерфейсы **IEnumerable и IEnumerator.** Для применения оператора foreach к пользовательскому типу данных требуется реализовать в нем эти интерфейсы.

**Интерфейс IEnumerable** ( перечислимый ) определяет всего один метод — GetEnumerator, возвращающий объект типа IEnumerator ( перечислитель ), который можно использовать для просмотра элементов объекта.

**Интерфейс IEnumerator** задает три элемента:

* свойство Current, возвращающее текущий элемент объекта;
* метод MoveNext, продвигающий перечислитель на следующий элемент объекта;
* метод Reset, устанавливающий перечислитель в начало просмотра.

**Итератор** представляет собой блок кода, задающий последовательность перебора элементов объекта. На каждом проходе цикла foreach выполняется один шаг итератора, заканчивающийся выдачей очередного значения. Выдача значения выполняется с помощью ключевого слова yield.

# ===34. Исключительные ситуации. Генерация и повторная генерация исключений.

**Structured Exception Handling** (SHE) - унифицированный подход для языков ориентированных на платформу .NET

System.Exception

Исключительная ситуация **exception** - это состояние ошибки, обнаруженное в программе в ходе ее выполнения (деление на ноль, невозможность выделения памяти при создании нового объекта и т.д. ). Исключение генерируется оператором throw( <выражение>), где аргумент - объект типа исключение.

if (b==0) throw new Exception("Zero devision");

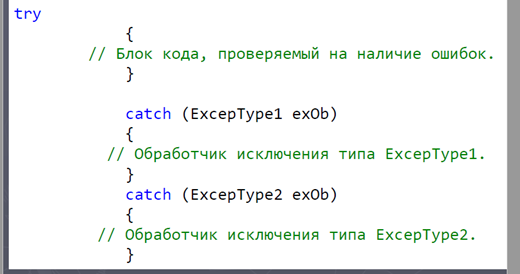
else c = a / b;

ГЕНЕРИРОВАНИЕ И РАСПОЗНАВАНИЕ ИСКЛЮЧЕНИЙ

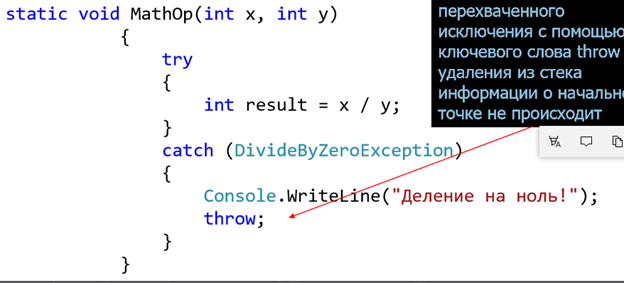
* **try** – контролируемый блок
* **throw** - генерация исключительной ситуации внутри try
* **catch** – обработчики исключений, идут за try (может быть несколько)
* **finally** - код, очищающий ресурсы и др. действия (выполняется всегда) (только 1)

Если код в блоке try не порождает исключение, CLR никогда не переходит к выполнению кода в соответствующем блоке catch.

Так как тип Exception является базовым классом для всех исключений, то выражение catch (Exception ex) будет обрабатывать практически все исключения.



**Повторная генерация исключения** - создание нового объекта посредством повторного использования старого с помощью оператора throw без параметров. При повторном вызове перехваченного исключения с помощью ключевого слова throw удаления из стека информации о начальной точке не происходит.



# ===35. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений

См. Вопрос 34. (начало по исключениям)

try {…throw выражение;..}

catch(объявление исключения)

{ операторы обработчика }

catch(объявление исключения)

{ операторы обработчика }

catch(объявление исключения)

{ операторы обработчика }

**try-catch, try-finally, try-catch-finally**

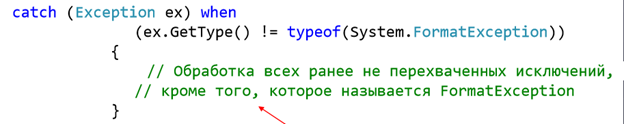
Поиск подходящего блока catch в CLR осуществляется сверху вниз, поэтому наиболее конкретные обработчики должны находиться в начале списка.

Сначала следуют потомки с наибольшей глубиной наследования, потом — их базовые классы (если таковые имеются) и, наконец, — класс System.Exception

Механизм:

1. Исключение не произошло
   1. try выполняем до конца
   2. catch пропускаем
   3. finally выполняем
2. Исключение произошло
   1. выполнение try
   2. ищем блок catch на соответствие по типу исключения
      1. если нет catch
         1. разматывает стек, локальные объекты, выходят из области видимости
         2. снова генерируется исключение в точке вызова метода
         3. если блока не найдено, то смс необработанное исключении и вып-ние программы stop
      2. catch найден
         1. Передается управление ближайшему catch-обработчику, совместимому с типом выброшенного исключения
         2. объект-исключения передается, если это предусмотрено, обработчику в качестве параметра.
         3. 2.3. переходим в finally
   3. если нет finally
      1. выполнение программы продолжается начиная с позиции, след. за последним обработчиком данного блока try

**Фильтр исключения** позволяет указать дополнительные условия, при которых используется обработчик исключения. Эти условия принимают форму булева выражения, перед которым ставится ключевое слово **when**.



int x = 1;

int y = 0;

try

{ int result = x / y; }

catch(Exception ex) when (y==0)

{ Console.WriteLine("y не должен быть равен 0"); }

catch(Exception ex)

{ Console.WriteLine(ex.Message); }

В данном случае будет выброшено исключение, так как y=0. Здесь два блока catch, но поскольку для первого блока указано условие с помощью ключевого слова when, то сработает первый блок catch. Если бы y не было бы равно 0, то сработал бы второй блок catch.

# ===36. Обобщения (generics). Свойства обобщений.

**Обобщение** - параметризированный тип. Определены для CLR – поддержка разных языков.



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

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

**Недостатки использования object:**

* InvalidCastException - два типа не совместимы друг с другом.
* вероятность дополнительного потребления памяти и процессорного времени, если в ходе выполнения потребуется преобразовывать

**Свойства:**

1. Обобщенный тип может содержать другой обобщенный тип

public class B<T>

{

private A<T> one;

private A<int> two;

}

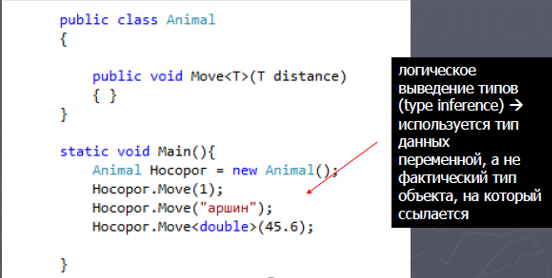
1. Обобщенные типы перегружаются на основе количества параметров (арность)

public class A {}

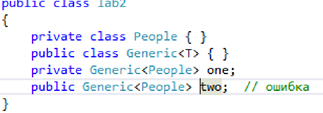
public class A<T> {}

public class A<T, U> {}

1. Обобщенными могут быть классы, структуры , интерфейсы, делегаты, методы public void Method <R> (A<R> iA, B<R,T> iB)



1. Могут содержать статические типы
2. Доступность конструируемых типов определяется на основе пересечения доступности обобщённого типа и типа в списке аргументов



1. могут использовать несколько универсальных параметров одновременно
2. поддерживает механизм ограничений

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

T id = null;

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

class Account<T>

{

T id = default(T);

}

# ===37. Концепция ограничений обобщений. Статические члены обобщений.

В CLR существует механизм ограничений (constraints) инструмент определения обобщенного типа с указанием допустимых для него аргументов типа.

private static T Min<T>(T o1, T o2) {

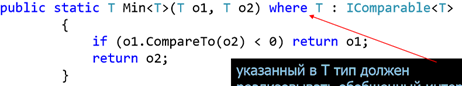
if (o1.CompareTo(o2) < 0) return o1;

return o2; //ошибка

}

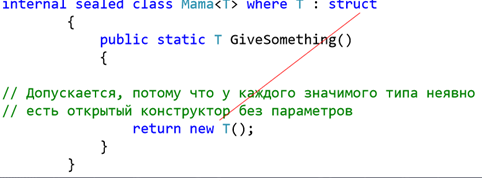
Ограничение сужает перечень типов, которые можно передать в обобщенном аргументе, и расширяет возможности по работе с этими типами. Указанный в T тип должен реализовывать обобщенный интерфейс IComparable того же типа (T).

**Ограничение на интерфейс**

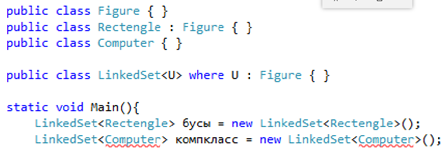


**Ограничение типа значения**

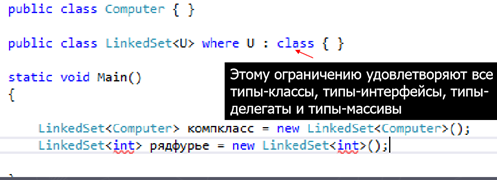
Гарантирует компилятору, что указанный аргумент типа будет иметь значимый тип, но значимые типы с поддержкой null (System.Nullable<T>) не подходят под это ограничение.

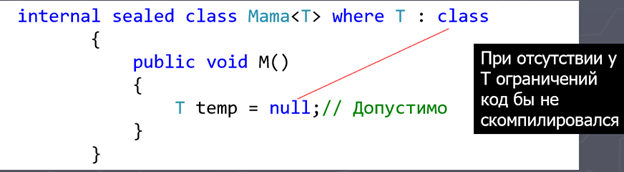


**Ограничение на базовый класс**

****

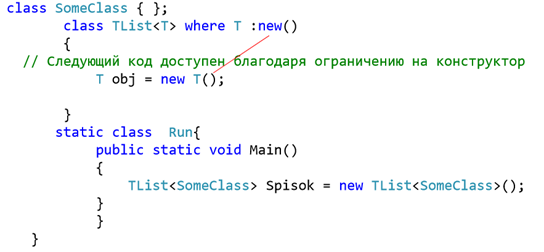
**Ограничение ссылочного типа**

****

****

**Ограничение на конструктор**

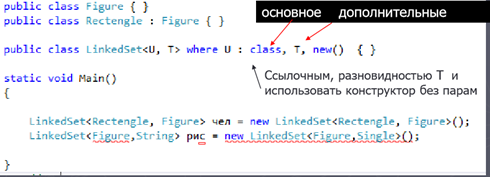
Гарантирует компилятору, что указанный аргумент-тип будет иметь не абстрактный тип, имеющий открытый конструктор без параметров.



**Особенности последним** по порядку

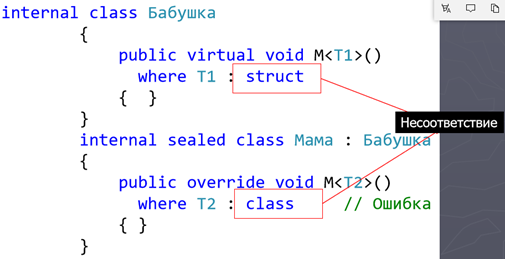
Ограничение new () позволяет конструировать объект, используя только конструктор без параметров. Ограничение new() нельзя использовать одновременно с ограничением типа значения.

**Ограничение на связь параметров**

****

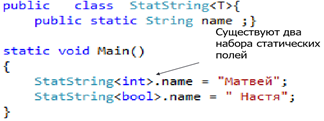
**Наследование**

При переопределении виртуального обобщенного метода в переопределяющем методе должно быть задано то же число параметров-типов, а они, в свою очередь, наследуют ограничения, заданные для них методом базового класса.



**Статические члены**

В CLR размещает статические поля типа в самом объекте-типе , каждый закрытый тип имеет свои статические поля



# ===38. Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты.

**Делегат** - это объект, предназначенный для хранения ссылок на методы. Более того, метод можно вызывать по этой ссылке. Иными словами, делегат позволяет вызывать метод, на который он ссылается.

Синтаксис:

**delegate возвращаемый\_тип имя (список\_параметров);**

Как только будет создан экземпляр делегата, он может вызывать и ссылаться на те методы, возвращаемый тип и параметры которых соответствуют указанным в объявлении делегата.

Пример:

delegate int **IntOperation** (int i, int j);

class Program {

static int Sum(int x, int y) { return x + y; }

static int Mul(int x, int y) { return x \* y; }

static void Main() {

**IntOperation** op1 = new **IntOperation**(Sum);

int result = op1(5, 10);

Console.WriteLine("Сумма: " + result);

op1 = new **IntOperation**(Mul); // меняем ссылку

result = op1(5, 10);

Console.WriteLine("Произведение: " + result);

}

}

В частности, объект делегата поддерживает три важных фрагмента информации:

* адрес метода, на котором он вызывается;
* аргументы (если есть) этого метода;
* возвращаемое значение (если есть) этого метода.

Делегат может хранить ссылки на несколько методов и вызывать их поочередно - сигнатуры всех методов должны совпадать.

Свойства делегатов:

* Тип данных
* Наследовать от делегата нельзя
* Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса (в любом месте, где может быть определен класс)
* может вызывать только такие методы, у которых тип возвращаемого значения и список параметров совпадают
* Может быть статический метод класса
* Имеет тот же синтаксис, что и вызов метод
* Если делегат хранит ссылки на несколько методов, они вызываются последовательно в том порядке, в котором были добавлены в делегат (цепочки (chaining))
* делегаты могут выполняться в асинхронном режиме, при этом не надо создавать второй поток, надо вместо метода Invoke использовать пару методов BeginInvoke/EndInvoke
* делегат можно вызывать как обычный метод
* делегаты могут быть параметрами методов

**Обобщенные делегаты**:

Синтаксис:

**delegate** возвр\_тип имя<список\_параметров\_типа> (список\_аргументов);

Список параметров типа следует непосредственно после имени делегата. Преимущество обобщенных делегатов заключается в том, что их допускается определять в типизированной обобщенной форме, которую можно затем согласовать с любым совместимым методом.

Обобщенные делегаты предоставляют более гибкий способ спецификации метода, подлежащего вызову в безопасной к типам манере.

Пример:

delegate T MyDel<T> (T obj1, T obj2); // обобщ. дел.

class MySum {

public static int SumInt(int a, int b) {return a + b; }

public static string SumStr(string s1, string s2) { return s1 + " " + s2;}

public static char SumCh(char a, char b){ return (char)(a + b); }

}

class Program {

static void Main() {

MyDel<int> del1 = MySum.SumInt;

Console.WriteLine("6 + 7 = " + del1(6,7)); //вывод: 6 + 7 = 13

MyDel<char> del3 = MySum.SumCh;

Console.WriteLine("'a' + 'c' = " + del3('a','c')); //вывод: ‘а’ + ‘с’ = A

}

}

# ===39. Анонимные функции. Лямбда-выражения.

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

**Анонимные функции** представляет собой безымянный кодовый блок, передаваемый конструктору делегата

Определение анонимных методов начинается с ключевого слова **delegate**, после которого идет в скобках список параметров и тело метода в фигурных скобках:

class Program

{

delegate void MessageHandler(string message);

static void Main(string[] args)

{

MessageHandler handler = delegate(string mes)

{

Console.WriteLine(mes);

};

handler("hello world!");

Console.Read();

} }

Свойства анонимных функций:

* Параметры должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе опустить параметры.
* может не содержать никаких параметров. Тогда он соответствует любому делегату, который имеет тот же тип возвращаемого значения.
* метод имеет доступ ко всем переменным, определенным во внешнем коде

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата. И через эту переменную делегата можно вызвать данный анонимный метод.

И важно отметить, что в отличие от блока методов или условных и циклических конструкций, блок анонимных методов должен заканчиваться точкой с запятой после закрывающей фигурной скобки.

Анонимные методы используются, когда надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется. Иногда такие методы нужны для обработки одного события и больше ценности не представляют и нигде не применяются.

**Лямбда-выражения -** упрощенная запись анонимных методов.

Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы.

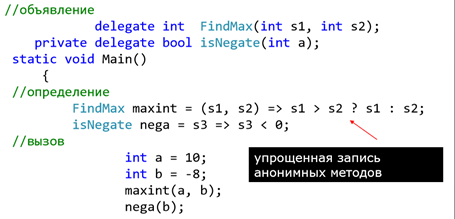
**Синтаксис:** параметр => выражение

i => i \* i;

(список\_параметров) => выражение

(x, y) => x + y;

Не надо указывать тип параметров, не надо использовать оператор return



При этом надо учитывать, что каждый параметр в лямбда-выражении неявно преобразуется в соответствующий параметр делегата, поэтому типы параметров должны быть одинаковыми. Кроме того, количество параметров должно быть таким же, как и у делегата. И возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата.

Список параметров может быть пустым

public delegate void D();

class Class1 { static void Main() {

D del = () => { Console.Write("Hello"); };

del();

} }

Можно передавать в качестве аргументов методу. Может принимать ссылку на метод. Лямбда-выражения можно передавать в качестве параметров методу.

# ===40. Обобщённые делегаты .NET. Action, Func, Predicate

**Делегаты** - объекты предназначенные для хранения ссылок на методы. Используются для поддержки событий.

Делегат может хранить ссылки на несколько методов и вызывать их поочередно - сигнатуры всех методов должны совпадать.

В .NET есть несколько встроенных делегатов, которые используются в различных ситуациях. И наиболее используемыми, с которыми часто приходится сталкиваться, являются Action, Predicate и Func.

Определенные в пространстве имен **System**

void Action<in T1, in Т2, in ТЗ ….in T16 >

TResult Func<out TResult>

TResult Func<in T1,……T16, out TResult>

Вместо определения собственных типов делегатов рекомендуется по мере возможности использовать обобщенные делегаты.

**Action.**

Делегат Action является обобщенным, принимает параметры и возвращает значение void:

public delegate void Action<T>(T obj)

Данный делегат имеет ряд перегруженных версий. Каждая версия принимает разное число параметров: от Action<in T1> до Action<in T1, in T2,....in T16>. Таким образом можно передать до 16 значений в метод.

Как правило, этот делегат передается в качестве параметра метода и предусматривает вызов определенных действий в ответ на произошедшие действия.

**Func.**

Еще одним распространенным делегатом является Func. Он возвращает результат действия и может принимать параметры. Он также имеет различные формы: от Func<out T>(), где T - тип возвращаемого значения, до Func<in T1, in T2,...in T16, out TResult>(), то есть может принимать до 16 параметров.

TResult Func<out TResult>()

TResult Func<in T, out TResult>(T arg)

TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)

TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3)

**Predicate.**

Делегат Predicate<T>, как правило, используется для сравнения, сопоставления некоторого объекта T определенному условию. В качестве выходного результата возвращается значение true, если условие соблюдено, и false, если не соблюдено:

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

Console.WriteLine(isPositive(20));

Console.WriteLine(isPositive(-20));

В данном случае возвращается true или false в зависимости от того, больше нуля число или нет.