## Платформа .Net Framework

**Платформа .Net Framework** представляет собой совокупность программных модулей в рамках которых в ОС Windows выполняются **.Net** приложения   
**.Net Framework** - это инсталляционный пакет который автоматически устанавливается при установке ОС Windows, также есть в свободном доступе на сайте Microsoft

**CLR** - общеязыковая среда исполнения

1. Отвечает за компиляцию и выполнение **.Net** программ.
2. Управляет ресурсами.
3. Генерирует нужное исключение.

Компоненты CLR

1. **Сборка** - формируется после компиляции состоит из 4ех компонентов
   * **Метаданные** - инфа о типах
   * **Манифест** - тех инфа о сборке
   * **Байт-код** (высокоуровневый ассемблер) в такой вид переводится откомпилированная программа в среде **.Net**, в отличие от низкоуровнего exe файла на уровне байт кода существует понятие объекта
   * **Ресурсы** - это доп файлы (картинки, звук), которые подключаться извне
2. **VES** (Virtual Execution System или Виртуальные процессор) - центральные понятие **CLR**. Единицей компиляцией для **VES** является сборка.
3. **JIT** компиляция (Just In Time) запускается **VES** процессором, компилирует методы которые вызваны с дальнейшем кешированием, после чего создается exe файл и отправляется к реальному процессору.
   * **Кэширование**. При **кэшировании** память выделяется только при первом вызове метода и не уничтожается после использования, при последующих вызовах будет использоваться та же функция. Таким образом первое выполнение **.Net** программ медленее происходит чем последующие вызовы.

Компоненты платформы .Net Framework

1. **FSL** - все классы .Net
2. **BSL** - подмножество **FCL**. Необходимый минимум для всех **.Net** совместимых языков и технологий
3. **CTS** - описание всех типов
4. **CLS** - это минимальные требование к языкам программирования для поддержки **.Net**

****

**C#** полностью объектно ориентированный язык, все должно находится в рамках класса

## Типы данных .Net

Все типы в **.Net** (даже те которые мы сами создаем и не реализуем явное наследование) наследуются от класса System.Object (Наследование).

### Все типы в **.Net** делятся на две категории:

1. **Ссылочные**
2. **Значимые**

| **Ссылочные** | **Значение** |
| --- | --- |
| class | struct, enum |
| Способ передачи в метод по ссылке | Способ передачи по значению |
| Размещение в куче | Размещение в стеке если не являются полем класса |
| Наследуются | Не наследуются |
| Конструктор по умолчанию переопределяется | Конструктор по умолчанию не переопределяется |

Все примитивные типы (int, double) в **.Net** являются структурами.

**Алиас** - это сокращенное название для значимых типов. Например для структуры Int32 **алиасом** является int.

Конструктор по умолчанию для значимых типов вызывается явно с помощью оператора **new**. При этом к выделению памяти в куче **new** не имеет никакого отношение

int a;

Console.Write(a); // Ошибка потому что конструктор по умолчанию не вызывается.

int a = new int();// Выхывается констурктор по умолчание.

Console.Write(a); // Будет 0.

Для ввода с клавиатуры используется метод **Console.ReadLine()**, который возвращает только строку.   
При необходимости мы используем класс **Convert** для получения желаемого типа. Или метод **Parse**, которые есть у любой структуры.

## Массивы в .Net

Все **массивы** - классы, наследуются от одного базового класса **SystemArray**. Так как **массив** это ссылочный тип данных, память выделяется в куче. Ссылочный **массив** имеет свои преимущества и недостатки

1. Преимущество - при наследовании **массив** получает необходимую функциональность от базового класса Array.
2. От того, что **массивы** размещаются в куче замедляется доступ к ним.

### Типы массивов в .Net

1. **Одномерные**
2. **Многомерные**
3. **Зубчатые** - разновидность многомерных **массивов**, каждая строка которой может содержать разное количество элементов. Экономит память, но уменьшается скорость обращения.

### Основные классы метода Array

1. **IndexOf** - поиск первого вхождения числа в массиве
2. **LastIndexOf** - поиск последнего вхождения числа в массиве
3. **Sort** - сортировка по возрастанию, реализация по быстрой сортировки
4. **Reverse** - переворачивает массив
5. **Copy** - копирует один массив в другой, параметры (откуда, куда, сколько)
6. **Clear** - обнуление массива
7. **GetLength** - Количество строк двумерного массива

## Nullable типы и null для значимых типов

В **.Net** есть null значения, в основном это значение для ссылочных типов и означает, что ссылка никуда не указывает. Часто возникает необходимость подобное поведение реализовать и для значимых типов. Обычный значимый тип (int) не может содержать в себе null. Поэтому более новая версия С# добавили **nullable типы**.

**GetValueOrDefault()** - Возвращает 0 если nullable тип содержит 0

**Оператор ??** - возвращает левый операнд если значение != null.

**Строка** - это ссылочный тип данных. Строки в **.Net** представляют собой набор **unicode** символов. Класс **System.String** позволяет создавать не изменяемые строки.

Индексатор для объекта **string[]** только в режиме чтения, а для **StringBuilder** на чтение и запись.

Из-за частого пересоздания объекта класса **string** теряем в производительности, а также в момента пересоздания не сразу освобождается память. Это проблему решает класс **StringBuilder**.

Свойство **Capacity** возвращает объем выделенной памяти в виде допустимые количество элементов.

## Классы в .Net

Начиная с версии C# 7.0 появилось понятие **“Top level Statements”** основная идея этого подхода - что в небольших приложениях основной класс и метод main можно явно не писать. При этом все классы и структуры должны создаваться после использования объекта. После компиляции в **IL коде** неявно создается пространство имен, класс и статический метод main и код подставляется в **IL код**.

**Класс** - это пользовательский тип данных, который позволяет создавать ссылочные типы. Может содержать поля, свойства и методы.

**Свойства в .Net** - это аксесоры, которые управляют доступом к конкретному полю класса.

В базовом классе **System.Object** есть виртуальный метод **ToString()**. Метод выполняет преобразование любого типа в строку. По умолчанию этот метод возвращает название типа. При необходимости переопределив **ToString() д**ля класса мы задаем формат строки для нашего класса.

**ToString()** не принимает параметров, возвращает строку определенного формата. При переопределении метода в .**Net** используется ключевое слово **override**

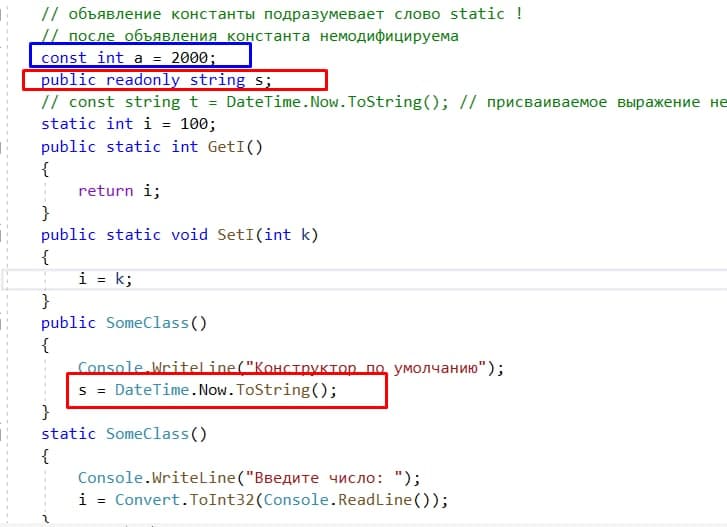
## Рефлекторы и дотфуксатор

**Рефлекторы** это специальные программы, которые позволяют исследовать сборку, а также существует возможность обратной компиляции кода.

**Дотфуксатор** защищает программные продукты от возможности обратной инженерии (процесс воссоздания продукта по его экземпляру).

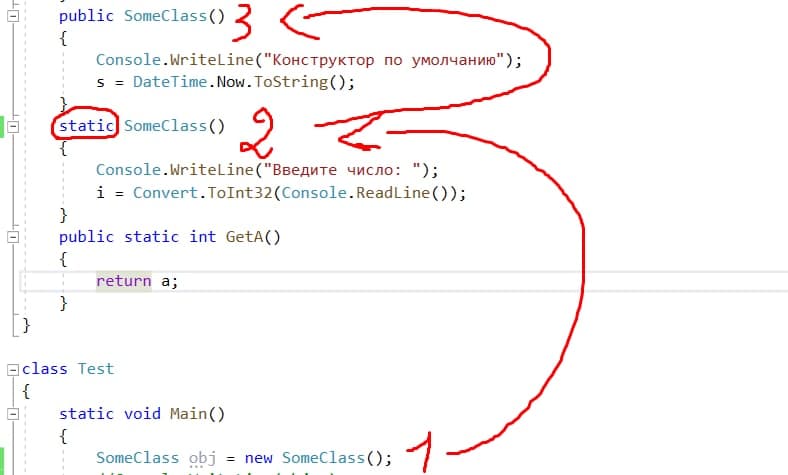
**Read only** поля можно инициализировать в конструкторе

**Константные поля** инициализируются сразу же при объявлении



В C# есть **статические конструкторы**. Создается без модификатора доступа

Статический конструктор вызывается при создании объекта, но до вызова экземплярного конструктора.



**Auto propartice** - автоматические свойства.

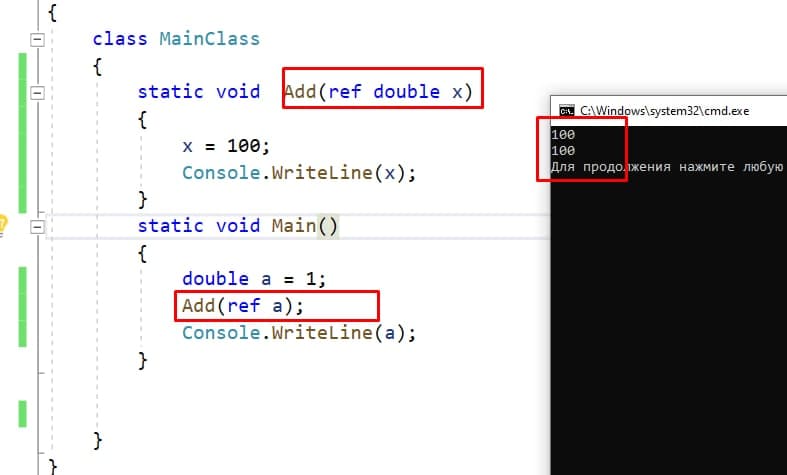
public int Age { get; set; } = 25 //private int\_Age = 25;

На основе автоматических свойств IL коде создаются инкапсулированные поля класса.

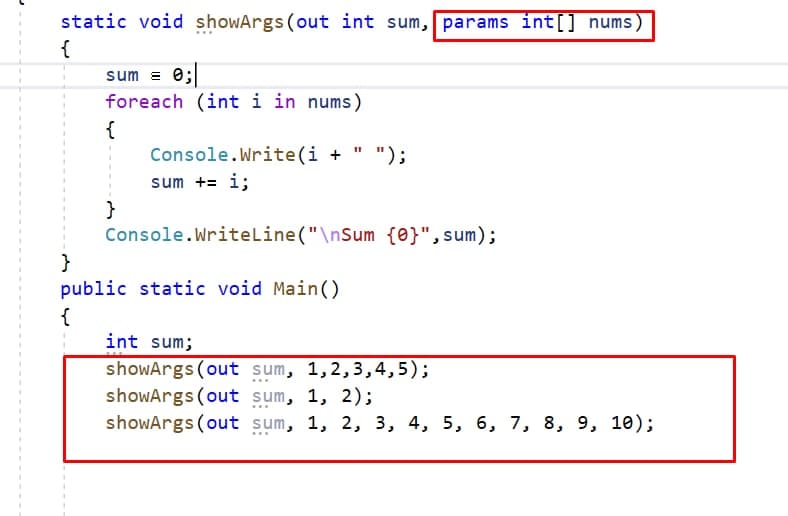
### Ref Out параметры

**Ref** - реализует передачу по ссылке. Входящий параметр должен быть инициализирован.

**Out** - передача по ссылке. Входящий объект может быть не инициализирован. При этом обязательно нужно инициализировать внутри метода после передачи.



Можно создать метод с произвольным количеством параметров. Для этого используется ключевое слово **params**.



Оператор **nameof** позволяет получить название типа через объект

## Модификаторы доступа в .Net

### Список модификаторов

**private** - доступ только в классе

**private protected**

**internal** - доступ только в текущем проекте

**protected internal**

**public**

Модификатор доступа для классов по умолчанию - **internal**. А для членов класса (поля и методы) модификатор по умолчанию **private**.



## Перегрузка операций

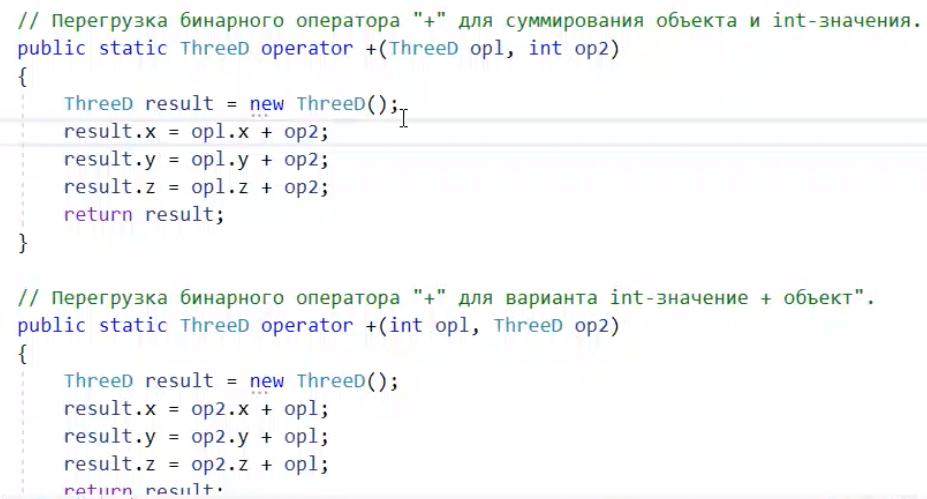
Перегрузка операций в **.Net** осуществляется с помощью открытых статических методов класса (public, static)

### Не перегружаемые операторы

* [], ()
* new
* is, as
* sizeof, typeof
* тернарные операции
* оператор присваивания

При перегрузке **бинарных** операций статический метод принимает 2 операнда (Потому что нет указателя this.).

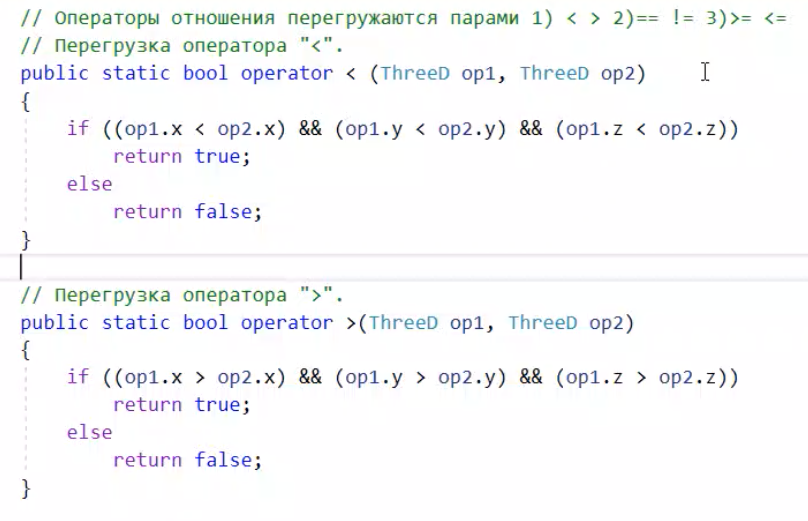
При перегрузке **унарных** операций 1 операнд.

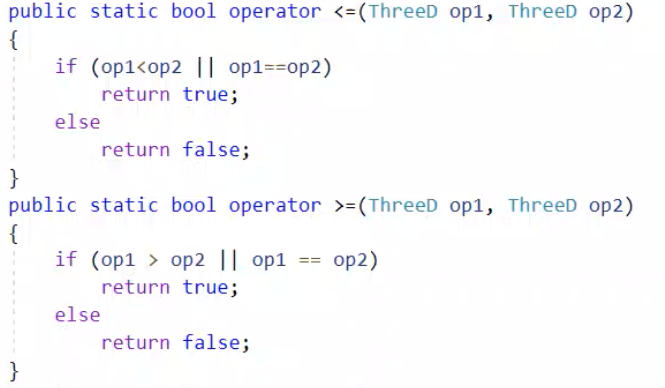


### Перегрузка логических операций

Логические операции перегружаются только парами

* < и >
* == и !=
* <= и >=



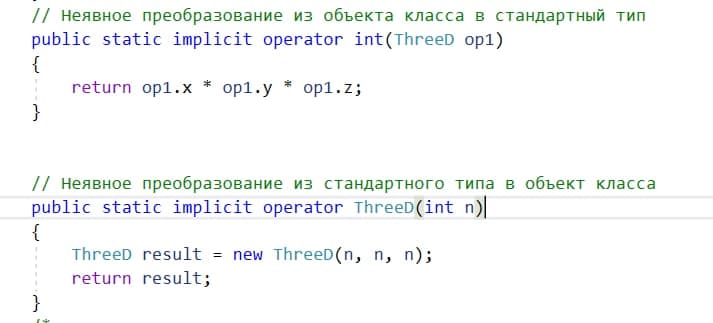


Операторный метод всегда должен возвращать **bool**.

### Перегрузка операторов преобразований

**Explicit** позволяет реализовать явное преобразование

**Implicit** позволяет реализовать неявное преобразование



Внутри класса нельзя одновременно реализовать и **explicit**, и **implicit** преобразование с одинаковыми сигнатурами.

**Implicit** срабатывает при явно и при неявном преобразовании.

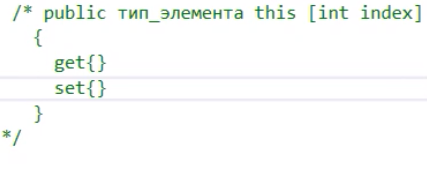
## Индексатор

**Индексатор** позволяет обращаться к элементам массива, который инкапсулирован в классе.

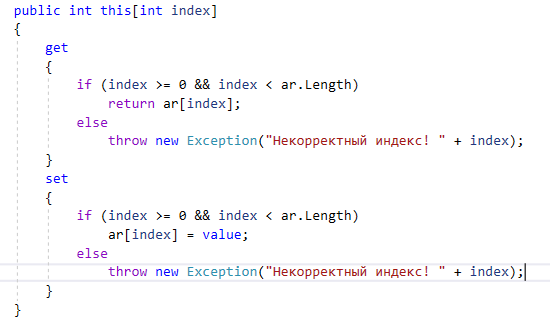
### Индексаторы бывают:

* **Одномерным** - для одномерного массива
* **Многомерным** - для многомерного массива.

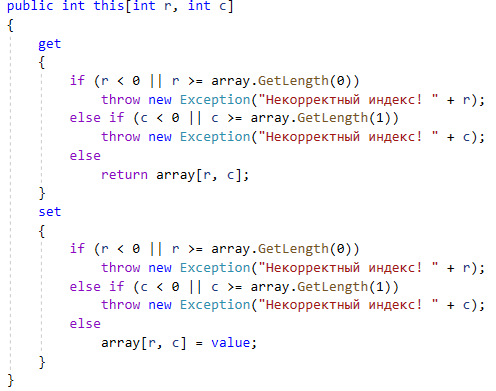
**Синтаксис** для одномерного **индексатора**.



При использовании вместо **this** подставляется наш объект



При реализации **многомерного индексатора** в [ ] указывается индекс строки и индекс столбца



### Namespace

Можно подключить глобально такие классы как

* **Console**
* **Convert**

И при использовании в коде явно не прописывать.

## Иерархии, исключение в .Net

**Исключение** в .Net являются объектами.

**BSL** содержит ряд классов, которые отвечают за обработку **CLR**.

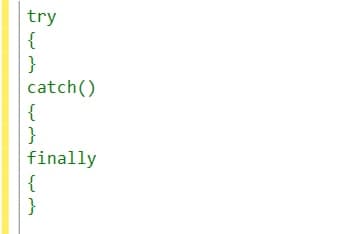
**CLR** автоматически генерируют самые распространенные **исключения**. Например при делении на 0 создается объект класса **DivideByZeroException**. При выходе за границы массива генерируется исключение типа **IndexOutOfRangeException**.

Все классы исключений являются потомками базового класса Exception.

У базового класса **Exception** есть два потомка **SystemException и ApplicationException**.

**ApplicationException** это базовый класс для пользовательских исключений, для более детальной обработки нестандартных ошибок

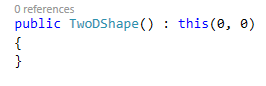
Синтаксис



**Finally** выполняется гарантировано в отличии от catch. Например в **Finally** можно прописать явное освобождение ресурса

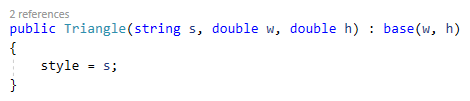
## Наследование

При делегировании между конструкторами используется ключевое слово **this**.



**.Net** поддерживает только одиночное наследование.

Для вызова методов базового класса и конструктора используется слово **base**.



**Абстрактный** метод обязательно необходимо переопределить в потомках.

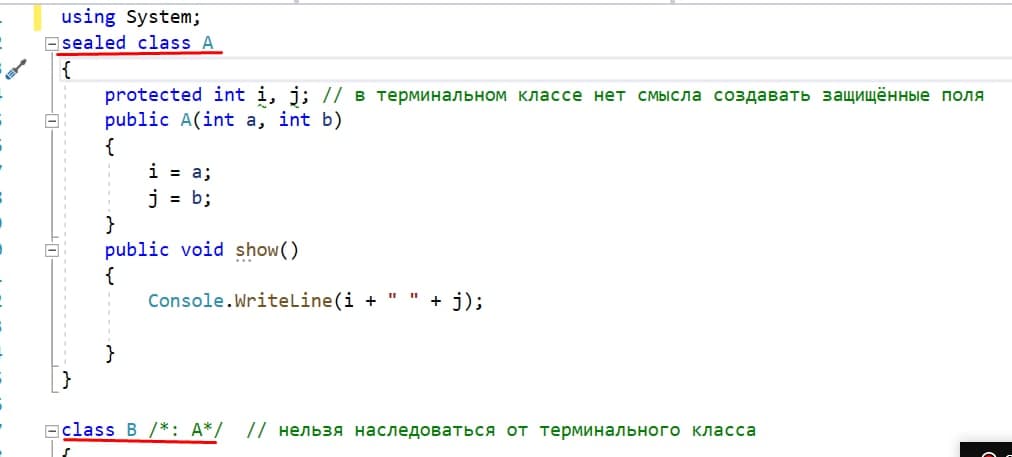
Объявляется с помощью слова **abstract** и не имеет собственного поведения в базовом классе.

Класс можно сделать **абстрактным**. **Абстрактный** метод можно создать только в абстрактном классе.

## Терминальный класс (sealed class)

**Терминальный класс** не может иметь собственных потомков.

**Терминальный** метод нельзя переопределить.



## Интерфейс

**Интерфейсы** в **.Net** относятся к ссылочным типам.

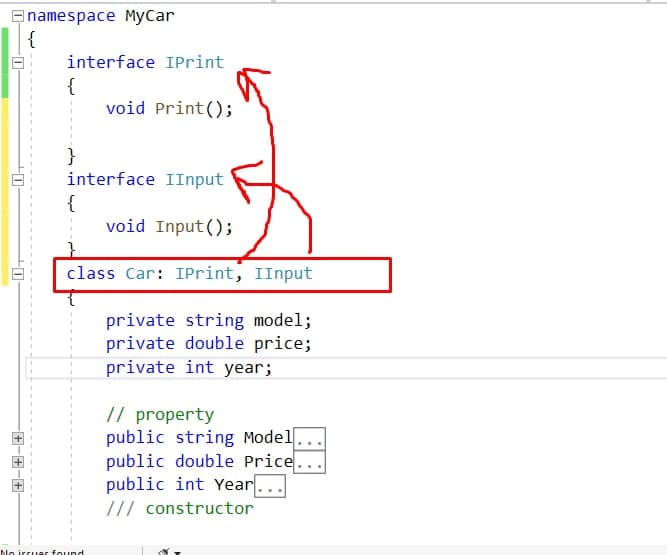
**Интерфейсы** создаются для реализации в разных иерархиях вне зависимости от типа.

**Интерфейсные** ссылки более абстрактные сущности чем ссылки базового абстрактного класса.

С точки зрения архитектуры классов - **интерфейсы** не должны содержать поля.

**Интерфейсы** не наследуются, а реализуются.

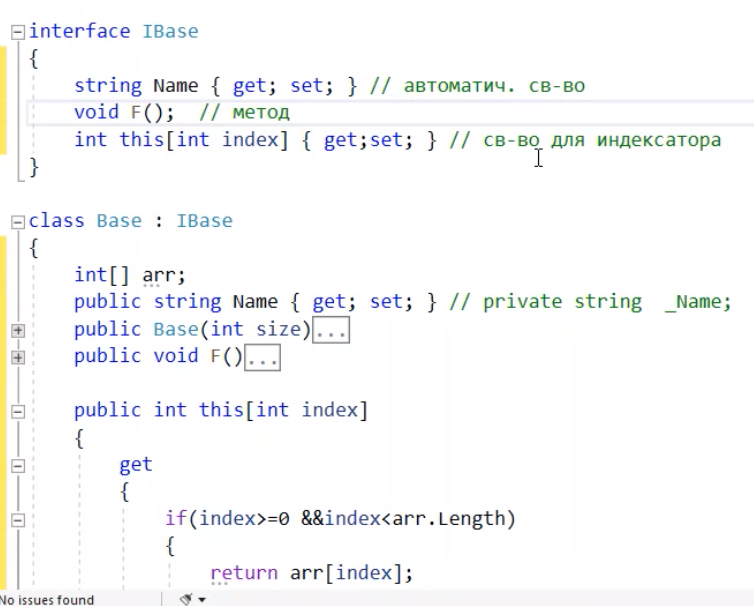
Класс может наследовать данные только от одного родителя и при этом может реализовать несколько интерфейсов.



### Интерфейсные свойства

Помимо прототипов **интерфейсы** могут содержать в себе автоматические свойства и свойства для индексатора.

**Интерфейс** - это контракт (набор требований) для будущих классов, которые будут его реализовать. Условия необходимо соблюдать.



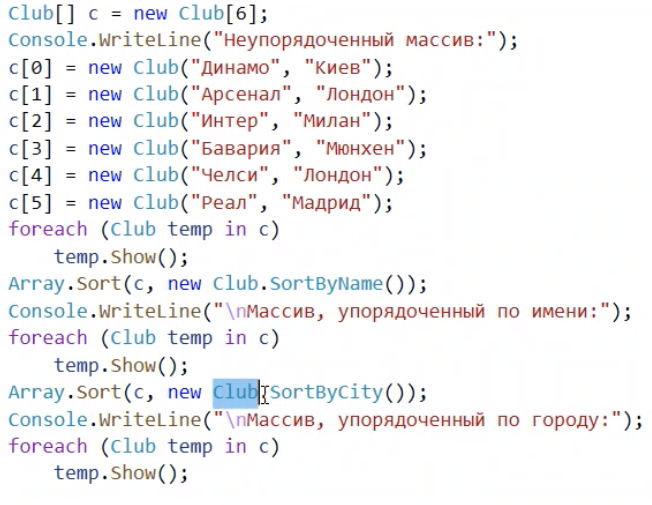
### Абстрактный класс

1. **Абстрактный класс** не может иметь собственных объектов. Является базовой минимальной основой для своих потомков.
2. **Абстрактный класс** - это заготовка для своих производных классов.
3. **Абстрактный класс** имеет минимальное базовое поведение (поля, реализованные методы).

### Стандартные интерфейсы

**.Net** представляет набор стандартных **интерфейсов** для решения часто возникающих проблем.

**IComparer**. Интерфейс содержат метод compare реализация которого в классе позволяет сравнивать между собой переданные компоненты.

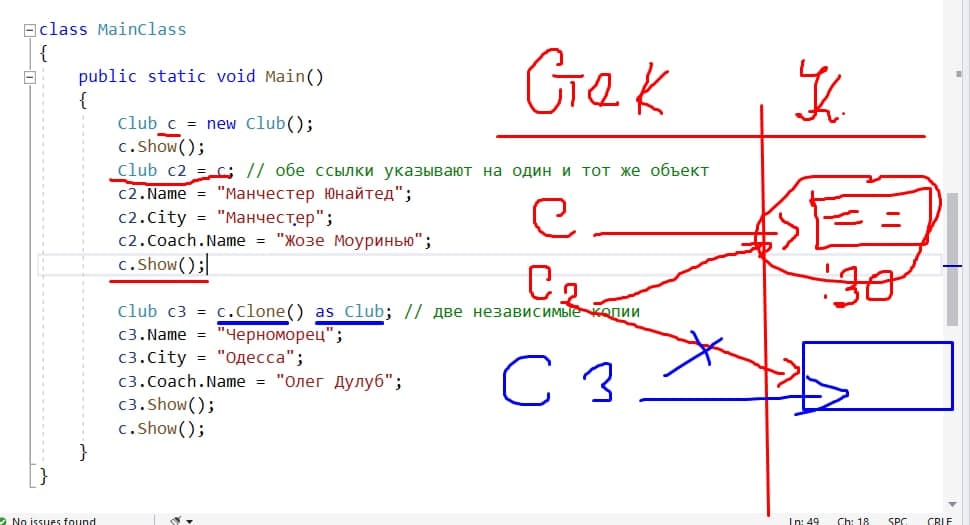


**IComparable**. Содержит 1 метод **CompareTo.**

**ICloneable**.Выполняет глубокое копирование, создает клон объекта.

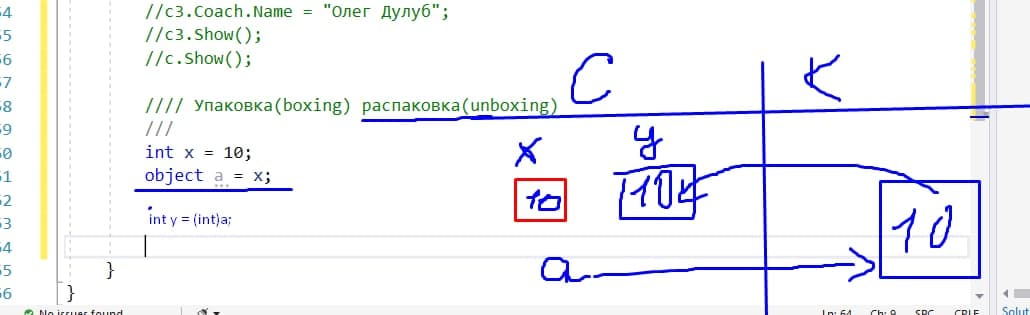
### Упаковка. Распаковка

**Упаковка** - это неявное преобразование любого значимого типа в object.   
При **упаковке** в куче выделяется место для object и записывается туда само значение справа от равно ( = ).

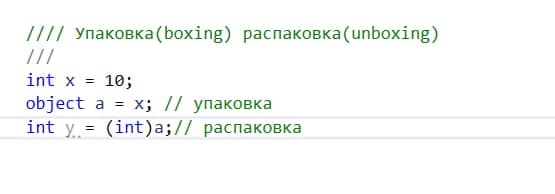


**Распаковка** - это неявное преобразование object в значимый тип.

При **распаковке** в стеке выделяется место для значимого типа и значение переносится из кучи в стэк.



**Упаковка и распаковка** очень сильно бьет по производительности.

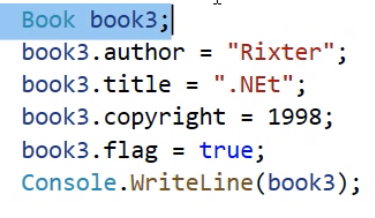


## Структуры

**Структуры** позволяют создавать значимые типы данных.

* **Структуры** не наследуются.
* **Структуры** могут реализовать интерфейс
* Нельзя явно переопределить конструктор по умолчанию так как у него собственное поведение.
* В теле конструктора по умолчанию происходит обнуления полей **структуры**.

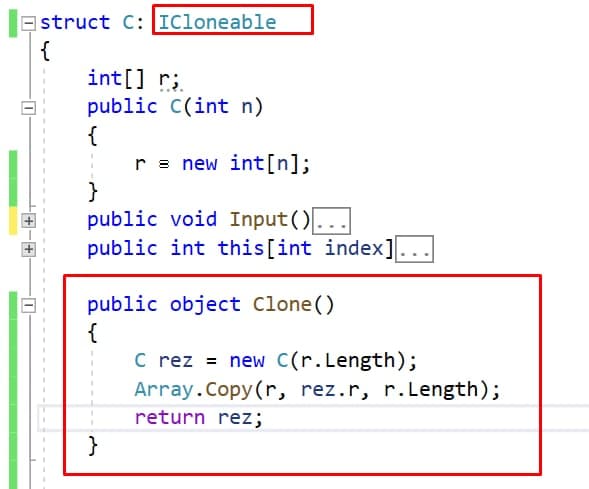
Оператор new для значимых типов вызывает конструктор.



Book boo3; - конструктор по умолчанию не вызывается. Память для объекта вызывается, но объект можно использовать только после явной инициализации.

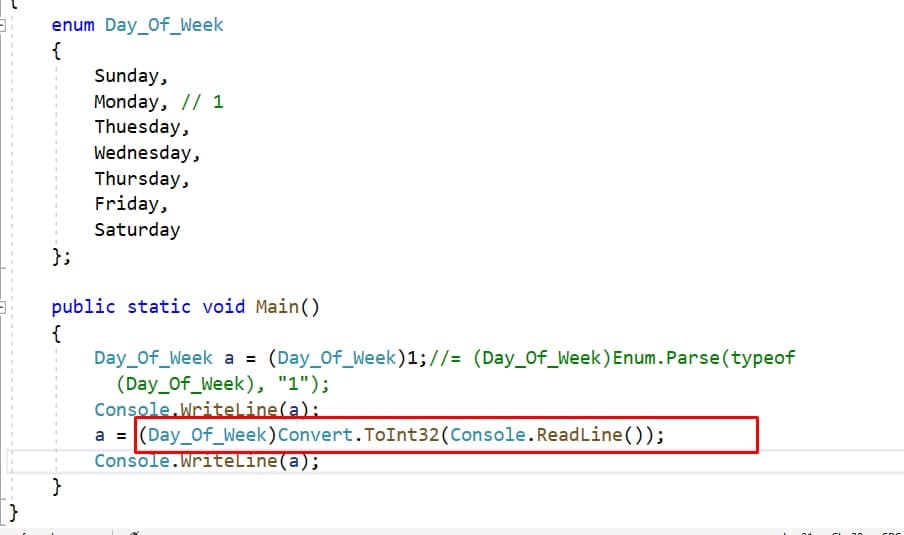
Book boo2 = new Book(); - Явный вызов конструктора по умолчанию.

Если значимый тип является полем структуры, то нельзя инициализировать при создании, кроме Const. Потому что память выделяется в стеке и за начальную инициализацию отвечает конструктор по умолчанию структуры.



### Перечисления Enum

Перечисления позволяют создавать набор именованных целочисленных констант. Все члены перечисления начинаются с 0



## Коллекции в BCL

**Коллекция** - это класс, который размещает совокупность каких либо данных, объектов в оперативной памяти по спец принципу.

**Коллекции** в BCL бывают двух категорий

### Обобщенные

**Проблема**: при использовании **обобщенных** коллекций на упаковку и распаковку отводится много времени. Для решения этой проблемы были созданы **обобщенные** коллекции, в которых тип определяется на этапе компиляции программы.

* 1. **List**<T> - массив
  2. **LinkedList**<T> - двусвязный список (односвязного нет в BCL
  3. **Stack**<T> - массив
  4. **Queue**<T> - массив
  5. **PriorityQueue**<T> - массив
  6. **Dictionary**<T> - массив
  7. **SortedList**<T> - сортированный словарь с массивом
  8. **SortedDictionary** - словарь с деревом

### Необобщенные.

Все элементы это object.

* 1. **ArrayList** - динамическая структура (массив)
  2. **Stack** - идеология LIFO (массив)
  3. **Queue** - идеология FIFO (массив)
  4. **SortedList** - сортировка по ключам (массив). Содержит два массива (ключей и значений). Так как сравнение идет по ключам, ключи должны быть однотипные.
  5. **HashTable** - словарь (key, value), внутри массив. Представляет собой совокупность объектов **DictionaryEntry**. У каждого объекта есть ключ значение.

## Делегаты

**Делегат** - это ссылочный тип, который представляет собой указатель на метод.

**Пространство** System.Delegate и System.MulticastDelegate.

**Синтаксис**

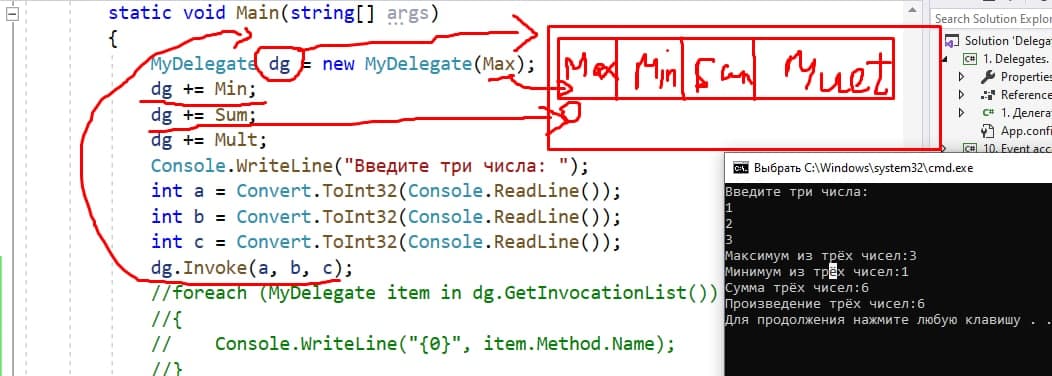
### 

**Делегат** (тип делегата) можно объявить за пределами класса или структуры.

Метод **Invoke()** вызывает метод адрес который содержиться в **делегате**.

Каждый объект **делегата** содержит в себе массив адресов, с помощью операции **+=** адрес метода добавляется в массив, **-=** адрес удаляется из массива.

Для получение инкапсулированного массива адресов используется метод **GetInvocationList()**.



**Мультикастинг** - представляет собой цепочку вызовов с помощью одного объекта **делегата**.

## Событие

**Событие** - это объект, который инициирует определенное действие и на это действие реагируют подписчики (наблюдатели).

Для создание объекта **события** (event) обязательно нужен **делегат**.

**Синтаксис**

[модификатор доступа] event ИмяДелегата ИмяСобытия.

С помощью **делегата** определяется сигнатура подписчиков.

**События**, как и **делегаты**, содержат в себе массив наблюдателей (Invocation List).

В большинстве случаев **события** можно заменить **делегатами**. Но с технической точки зрения, есть отличия между ними:

* Объект **делегата** может существовать без другого объекта
* **События** не может существовать без объекта инициатора.

Список наблюдателей **событий** контролируется с помощью двух операция   
**+=** и **-=** (**=** запрещено)

Есть понятие **“Событийное свойство”**, они позволяют контролировать доступ к объекту **event** если тот инкапсулирован в классе. Также благодаря **событийным свойствам** можно явно создать список наблюдателей и реализовать доступ к этому списку.

**Событийное свойство** представляет собой конструкцию, которая создается по следующему синтаксису



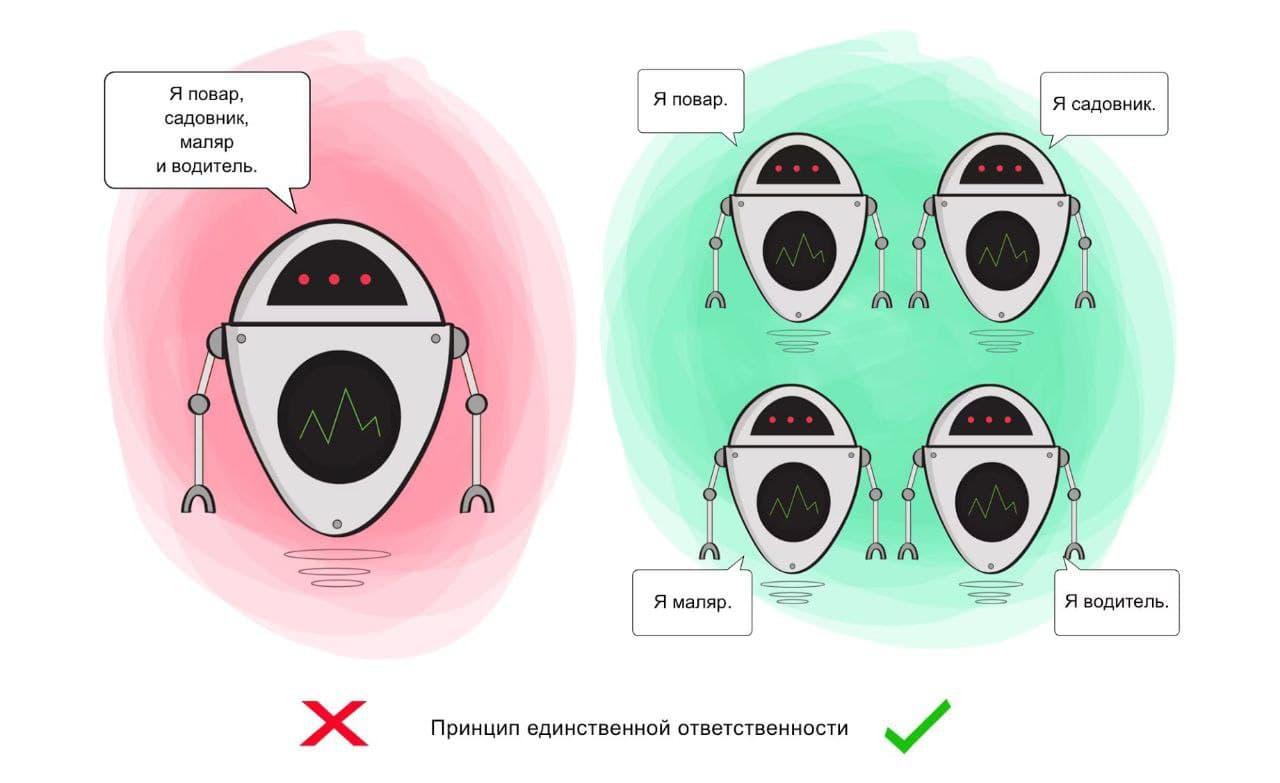
## SOLID (Солид)

* S. Принцип единой ответственности (SRP)
* O. Принцип открытости/закрытости (OCP)
* L. Принцип заменяемости Барбары Лисков (LCP)
* I. Принцип разделения интерфейсов (ISP)
* D. Принцип инверсии зависимостей (DIP)



### SRP

**SPR** - Каждый класс должен отвечать только за одну операцию. У класса должен быть 1 мотив для изменения.



Для максимальной гибкости необходимо исключать следующие правила

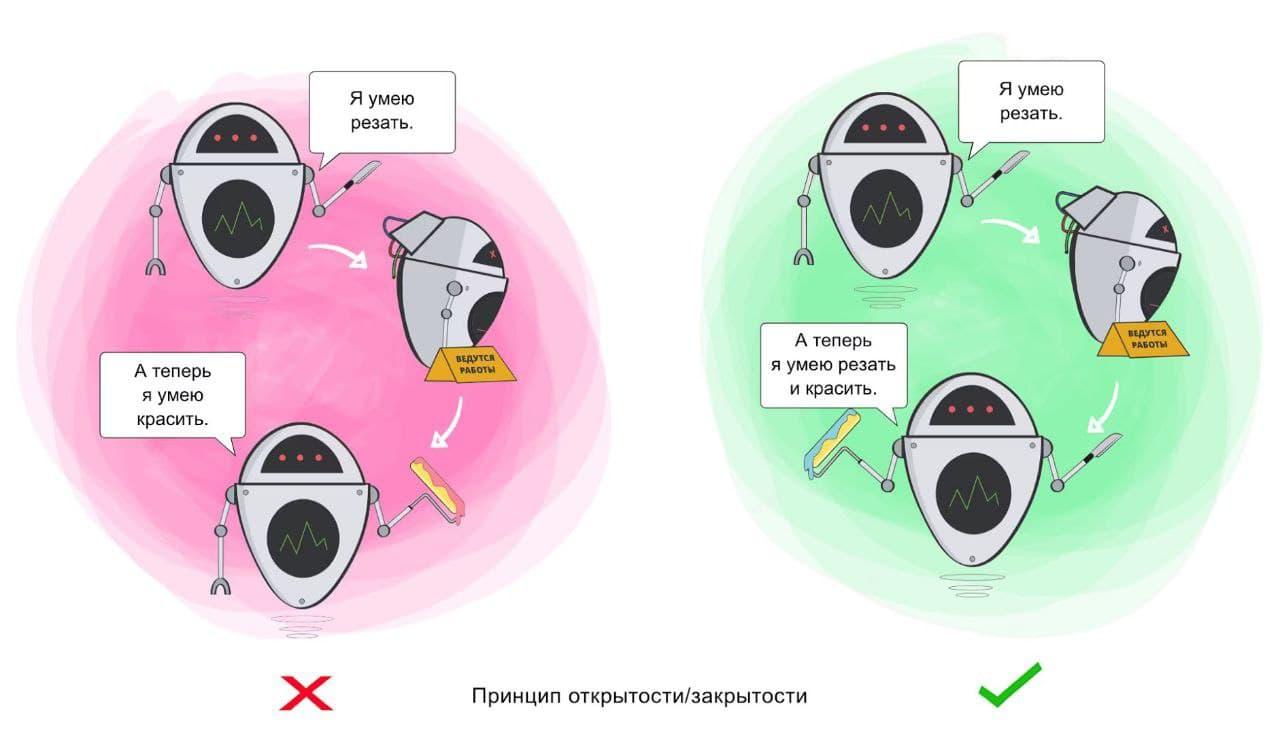
* Сильное сцепление
* Слабая связанность

Все, что создается внутри класса, максимально сильно взаимосвязано (поля, методы).

Слабую связанность необходимо реализовать между объектами разных классов.

### OCP

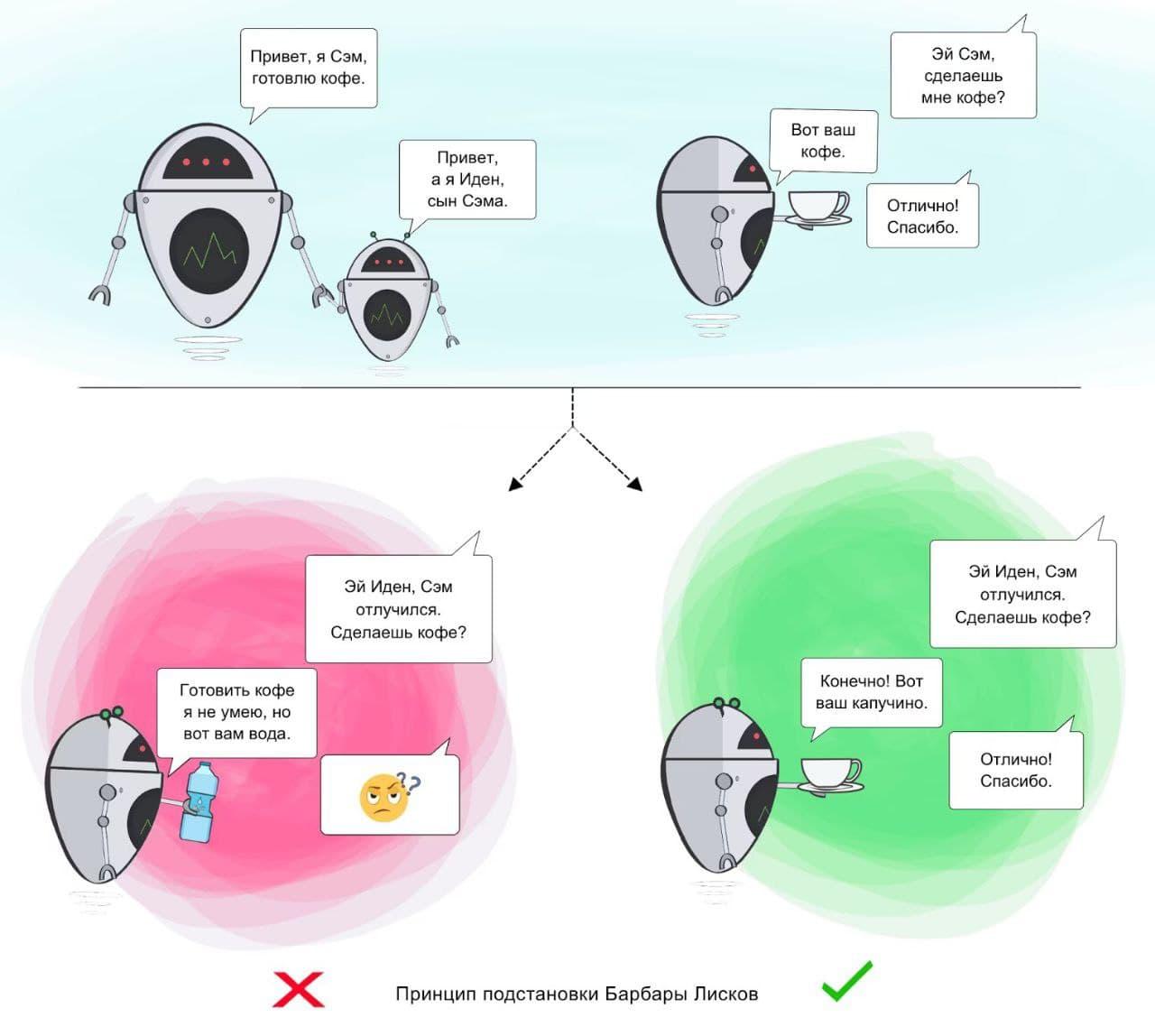
**OCP** - Программные сущности (классы) должны быть открыты для расширения но закрыты для изменения. Это означает, что такие сущности могут позволять менять свое поведение без изменения их исходного кода.



### LCP

Принцип подстановки **Барбары Лисков** гласит - должна быть возможность вместо базового типа подставить любой производный тип. Принцип нарушается если внутри функции или метода от абстрактного типа переходим типам потомков.

Например в **С++** это можно сделать с помощью **dynamic\_cast<>**

****

Функции, которые используют указатели на базовые классы должны иметь возможность использовать объекты произвольных классов не зная об это.

Необходимо использовать абстракцию на тип без лишних проверок на них.

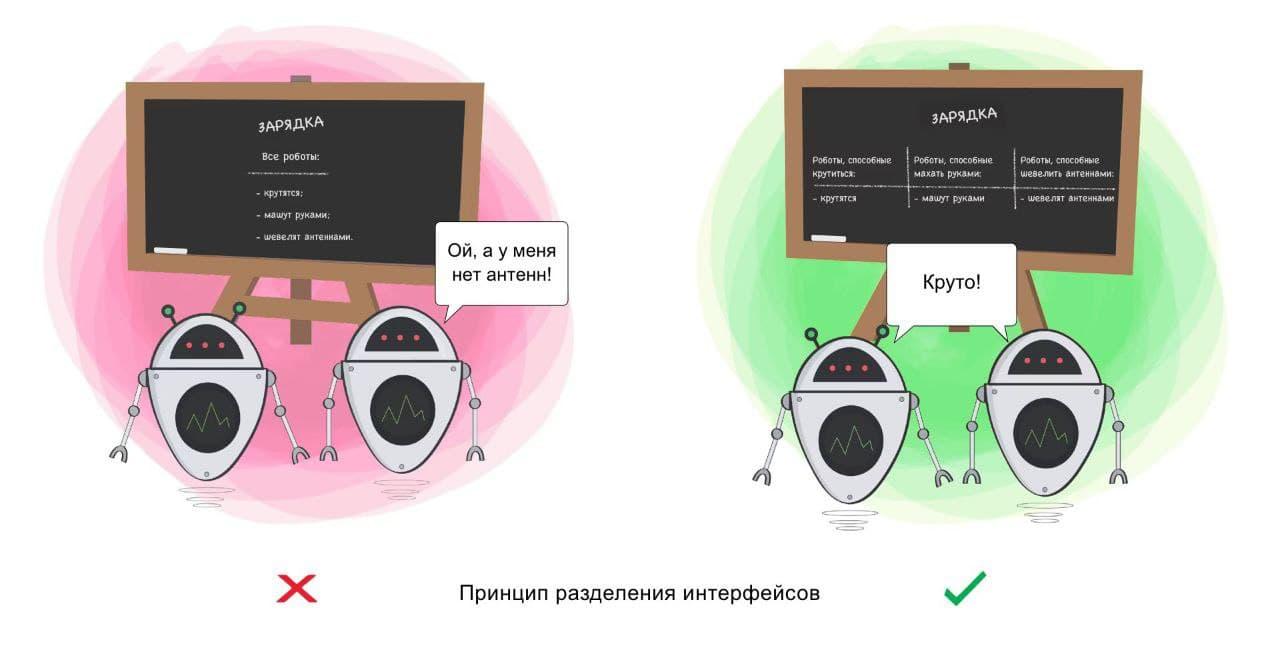
В **.Net** вместо dynamic\_cast<> можно использовать **is**.

### ISP

**ISP** - означает, что много специализированных интерфейсов лучше чем один универсальный.

Классы не должны зависеть от методов, которые они не используют.

Слишком толстые **интерфейсы** необходимо разделять на маленькие и специфические. При изменения метода интерфейса не должна меняться классы, которые этот метод не используют.



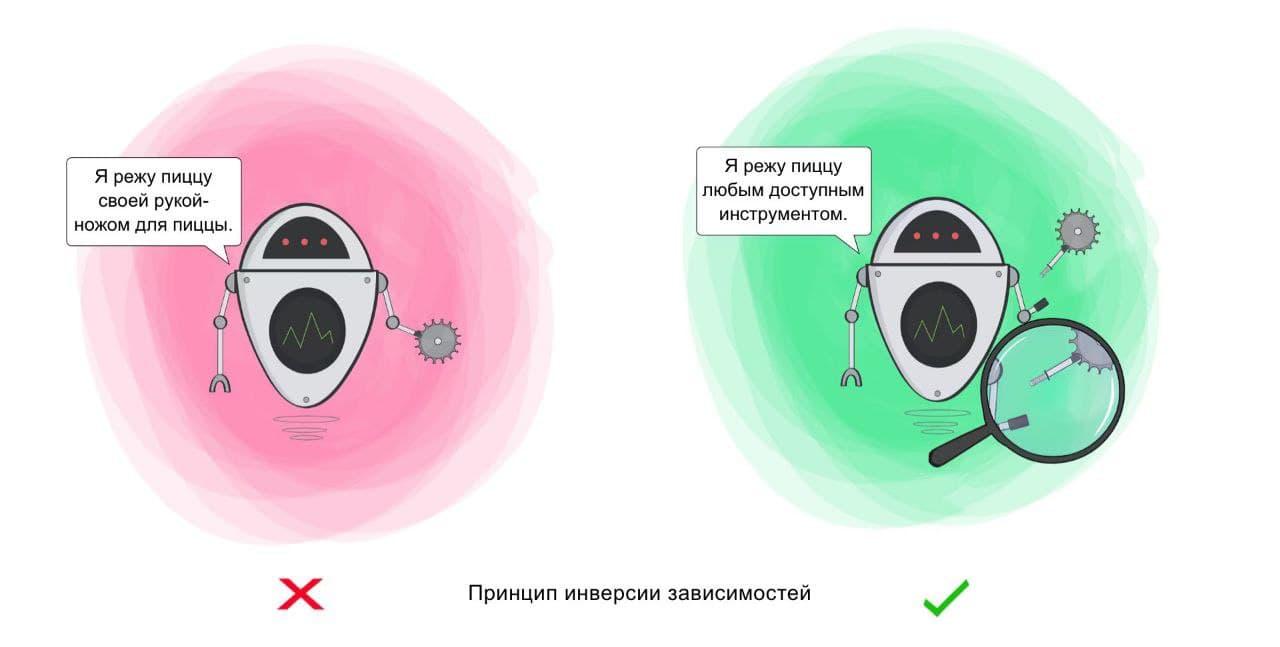
### DIP

**DIP** используется для минимизации зависимостей между классами

Классы верхних уровней не должны зависеть от классов нижних уровней. Оба должны зависеть от абстракции. Абстракции не должны зависеть от абстракции, а наоборот, детали должны зависеть от абстракции.

При проектировании можно выделить два уровня классов.

* **Нижнего уровня** - это базовые операции с объектами высокого уровня
* **Высокого уровня** - содержат сложную логику программы, которые опираются на классы нижнего уровня для простых операций.



## Сборщик мусора. Garbage collector

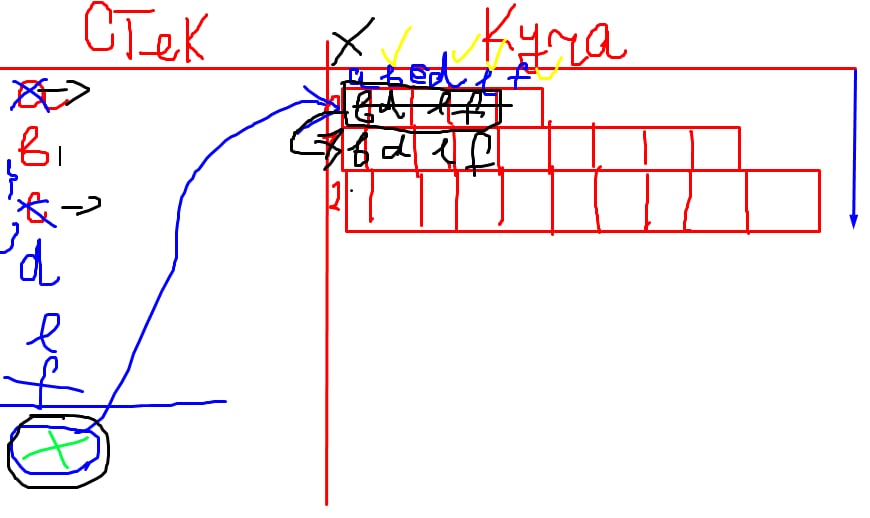
**Управляемая куча** представляет собой набор поколений, их 3 (0, 1, 2).

**0 поколение** имеет минимальный размер среди всех поколений. Все объекты ссылочных типов сначала попадают в **0 поколение**.

Когда **0 поколение** заполняется и в коде создается новый объект ссылочного типа впервые запускается алгоритм сборки мусора.

### Алгоритм сборщика мусора:

1. **Отслеживание ссылок**
   1. Проверяет на каком участке есть активные ссылки
2. **Маркировка.** 
   1. Участки, которые имеют активные ссылки в стеке маркируются
   2. После проверки всех объектов, куча содержит набор маркированных и немаркированных объектов
   3. Маркированные объекты переживут уборку мусора потому что на них ссылается хотя бы 1 объект в стеке. Немаркированные недостижимы потому что в приложении не существует корня через которое приложение могло бы к ним обратиться, в стеке нет указателя
3. **Сжатие** 
   1. После сжатия все достижимые участки располагаются строго последовательно в текущем поколении.
4. **Перенос** 
   1. Перенос достижимых объектов в следующее поколение
5. **Выделение памяти** 
   1. Выделение памяти для нового объекта в 0 поколении



**Вопрос?**   
Сколько объектов в 0 поколении после первого запуска сборщика мусора?

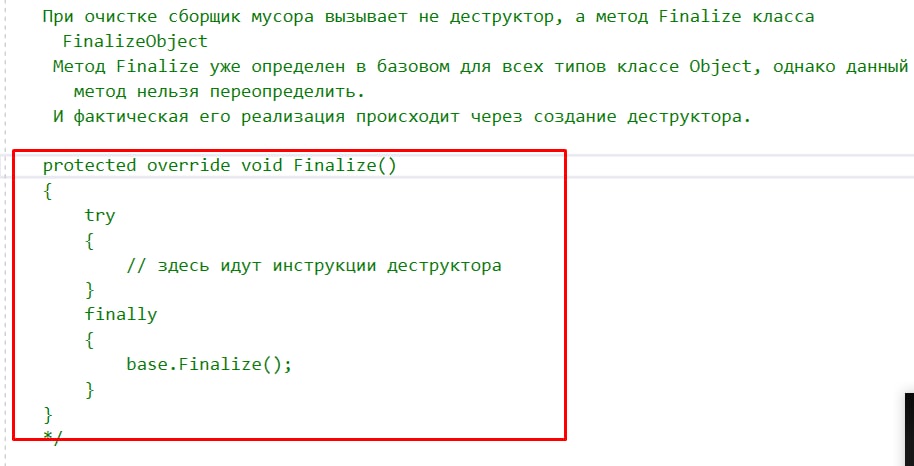
**Ответ**: В 0 поколении только новый объект. Все старые достижимые объекты перешли в следующее поколение

**Предположения**

1. Чем младше объект тем короче его время жизни
2. чем старше объект тем длиннее его время жизни
3. уборка мусора в части кучи выполняется быстрее чем во всей куче

**Алгоритм сборщика мусора** корректно работает только с простыми объектами. Объекты которые содержат в себе ссылки на файлы или БД не будут корректно уничтожены из управляемой кучи.

Для корректного удаления таких объектов **.Net** предоставляет несколько способов:

1. Интерфейс IDisposable. Есть метод Dispose().   
   При использовании using для выделенных объектов, автоматически будет вызываться метод DIspose() до того как освободиться место в управляемой куче
2. Использование финализатора. При создании “деструктора” в классе происходит не явное переопределение метода Finalize() для текущего класса. В блок try этого метода попадает тело “деструктора”  
   

При использовании **using** Dispose() вызывается автоматически при выходе из области **using**

class System.GC позволяет явно вызвать сборку мусора в указанном поколении

## Файловые потоки

class **StreamReader** и **StreamWriter**. Данные классы по умолчанию работают с текстовым форматом (текст файлы).   
**StreamReader** - чтение файлов   
**StreamWriter** - запись в файл

Метод **Peek()** - возвращает следующий доступный символ или -1 если нет символа

**FileStream** - байтовый класс. Обеспечивает доступ на уровне байт

class **BinaryReader** и **BinaryWriter** - это классы обертки для работы в бинарном формате.

**BinaryReader** содержит ряд методов для чтения данных определенного типа.

### Работа с директориями и папками

Для работы с директориями используются классы **Directory** и **DirectoryInfo**

Для работ с файлами используются классы **File** и **FileInfo**

Классы **Directory** и **File** - статические

## Регулярные выражения

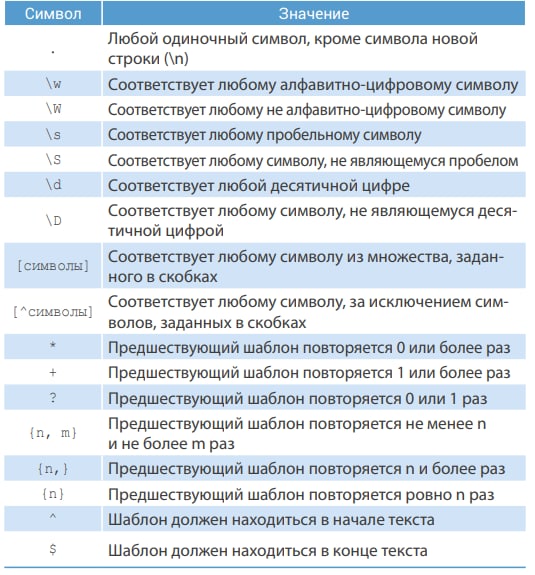
**Регулярное выражение** представляет собой шаблон, с помощью которого можно создать строку определенного формата для соответствия с конкретной строкой

**Например**. Можно создать регулярное выражение для поиска email адресов указанного формата.

В **.Net регулярное выражение** является объектом, class **Regex**. Пространство имен System.Text.RegularExpression.

Конструктор класса **Regex** принимает шаблон регулярного выражения в виде строки и дополнительный параметр перечисление **RegexOptions**.

Метод **IsMatch()** получает в качестве параметра строку и сопоставляет с шаблоном. Если строка соответствует шаблону - возвращается true.



## XML

**XML** - расширяемый язык разметки, предназначенный для описания структурированных данных.

Текстовые документы не имеют в себе структуру хранение данных. Соответственно для работы с текстовыми файлами часто используется позиционирование в файле.

**XML** применяется:

1. Для хранение данных
2. Для обмена данными между программами
3. Для создания на его основе более специализированных языков разметки

Цель создания **XML** - обеспечение совместимости при передаче структурированных данных между разными системами обработки информации

Основные конструкции **XML**

1. Элементы
2. Атрибуты
3. Комментарии

**XML** парсер - это специальный анализатор XML документа

**Пространство имен** - System.Xml

**XmlTextReader** - предоставляет средство чтения, прямой доступ к данным XML

**XmlTextWriter** - Предоставляет средство записи

## Обобщения. Generic

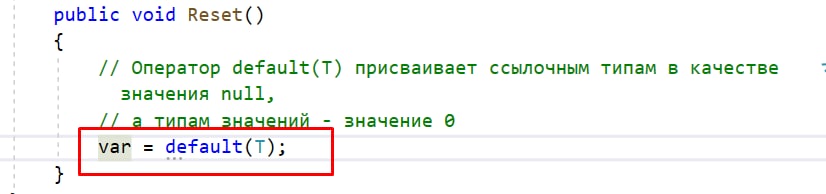
Проблема.

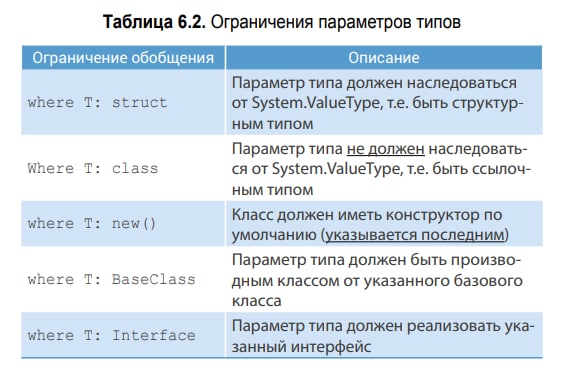
1. При использовании не обобщенных коллекций постоянно приходиться использовать упаковку и распаковку
2. Безопасность типов

Преимущества обобщений

1. Производительность
2. Безопасность типов

Оператор default принимает тип и возвращает значение по умолчанию для этого типа

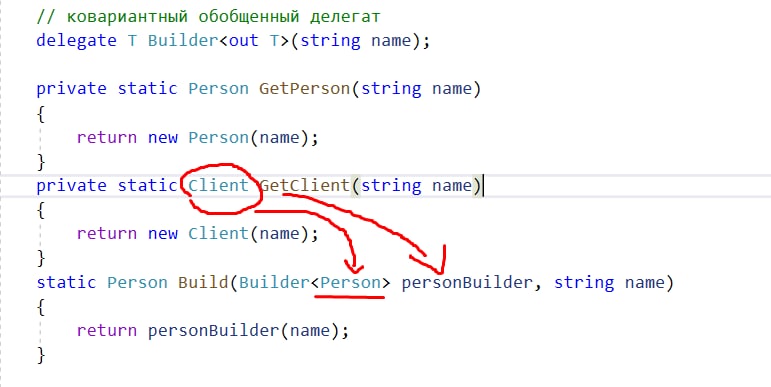




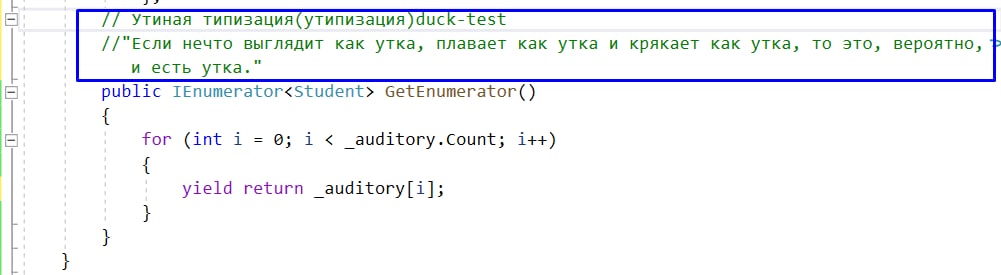
## Утипизация

**Вариантность** - перенос наследования исходных типов на производные от них типы. Если метод получает в качестве параметров базовый тип, то метод также может получить любой производный тип. Тоже самое касается к типам возвращаемого значения

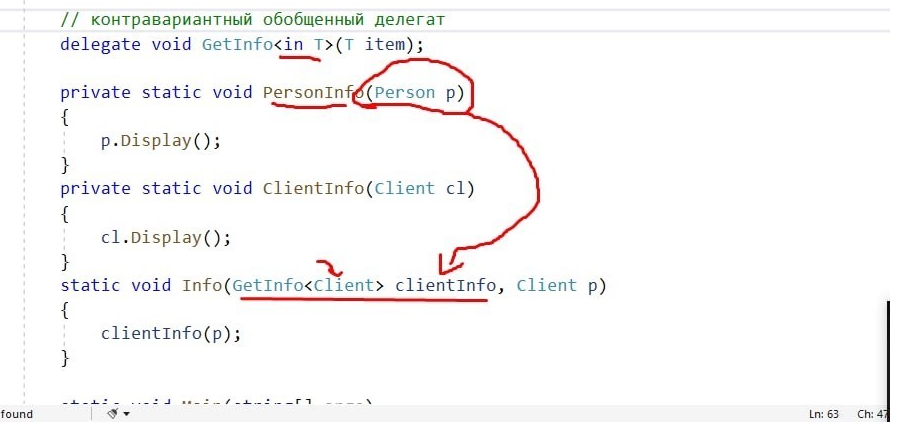
**Ковариантность** - перенос наследования исходных типов на производные от них типы в прямом порядке. Позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата. **Ковариантность** позволяет использовать более конкретный тип, чем заданный изначально



**Утипизация.** если нечто выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, утка



**Контравариантность** - перенос наследования исходных типов на производные от них типы в обратном порядке. Позволяет присвоить делегату метод, типом параметра которого служит класс, являющимся базовым для класса, указываемого в объявлении делегата. **Контравариантность** позволяет использовать более универсальный тип, чем заданный изначально



**Утипизация** - это возможность определить объект не по типу а по поведению.

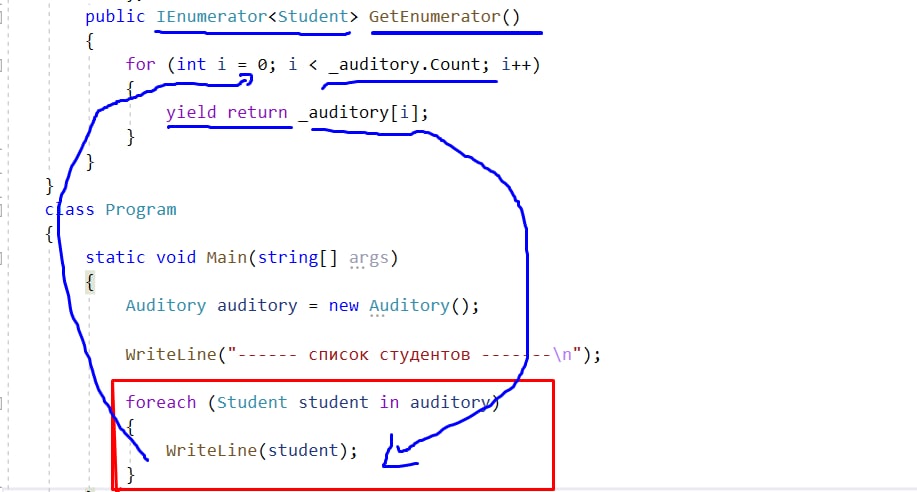
## Итератор

**Понятие итератор**. Для того, чтобы класс коллекция мог перечислять свои элементы при помощи foreach он должен реализовать:

1. Свой итератор
2. Интерфейс IEnumerable

С тех. точки зрения, итератор позволяет реализовать перебор коллекции. Для реализации итератора используется ключевое слово **yield**, которая предназначена для возврата значений в конструкцию foreach.

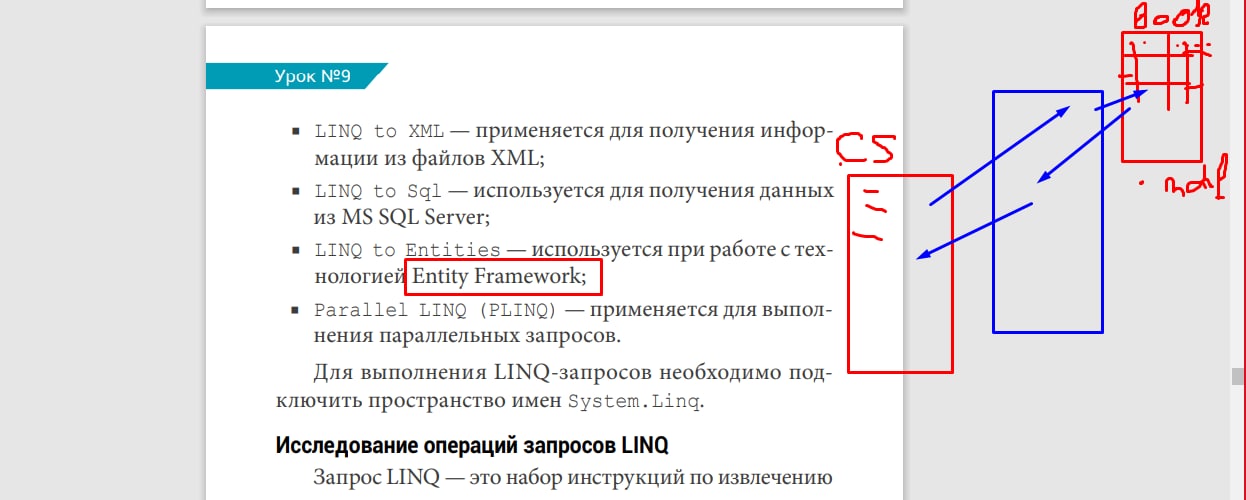
Название итератора обязательно **GetEnumerator**. Тип возвращаемого значения IEnumerator и тип возвращаемой коллекции.

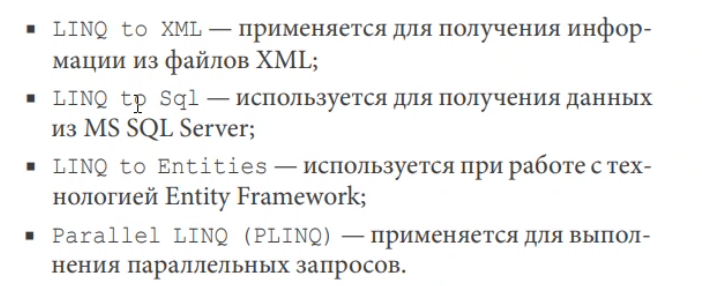


## LINQ

Пространство System.Linq

**LINQ** позволяет реализовать взаимодействие с хранилищами данных вне зависимости от типа, начиная от массивов и заканчивая базами данных





Особенности синтаксиса **LINQ**.

1. Запрос всегда начинается с оператора **from**
2. Создается переменная и направляется на источник данных с помощью оператора **in**
3. Оператор **select** обеспечивает возврат значений полученных при выполнении всех операторов расположенных между **from** и **select**

**LINQ to Object** - позволяет получать информация из различных коллекций

**orderby** ob- сортировка по возрастанию

**orderby** ob **descending** - сортировка по убыванию

Существуют методы расширения **LINQ**, который соответствуют операторам **LINQ**

**Проекция** позволяет спроектировать из текущего типа выборки в другой тип, а также **проекция** позволяет использовать несколько источников параллельно

**Агрегатные операции** (функции) всегда возвращают только 1 результат

## Атрибут. Сериализация объекта

Существует возможность добавления дополнительных сведений типах в метаданные, для этого используются **атрибуты**.

Пространство **System.Attribute**

Для создания собственного атрибута необходимо наследоваться от базового класса **System.Attribute**

Удобство **атрибутов** заключается в том, что можно внести инфу в сборку, а другой разработчик может достать эту инфу, и в зависимости от полученных данных менять поведения логику работы с классом.

Метод **GetType()** возвращает тип объекта.

Метод класса Type - **GetCustomAttribute()** возвращает список атрибутов для конкретного типа в виде массива Object.

**Сериализация** - процесс преобразования какого либо объекта в поток байтов.

**Десериализация** - процесс преобразования из байтов в объекта.

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом **Serializable**

## Рефлексия

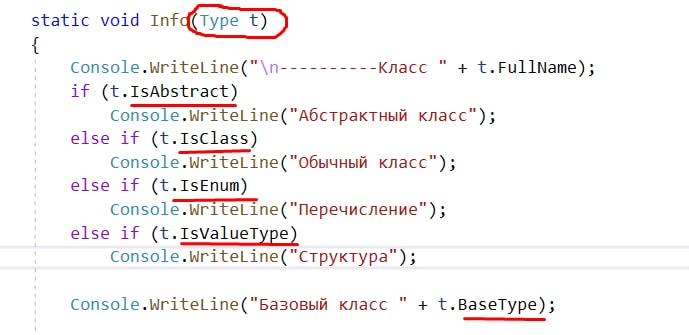
Рефлексия - представляет собой процесс выявления типов о время работы приложения

Пространство **System.Type**. Класс представляет изучаемый тип, инкапсулирует всю информацию о нем

Класс **Type** позволяет получить детальную информацию о типе.

Свойства:

1. IsAbstract - абстракт да\нет
2. IsClass - класс да\нет
3. IsEnum - перечисления да\нет
4. IsValueType - структура да\нет
5. BaseType - Возвращает базовый класс



Метод **GetMembers()** - возвращает коллекцию MemberInfo

1. DeclaringType - какой класс
2. MemberType - какой тип
3. Name - какое название

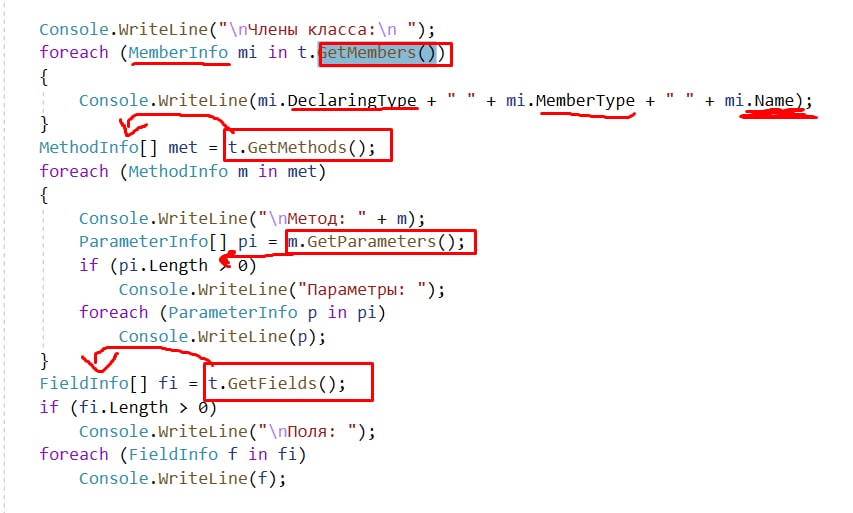
Метод **GetMethods()** - возвращает массив MemberInfo[]

Метод **GetParameters()** - возвращает массив ParameterInfo[]

Метод **GetFields()** - возвращает массив полей класса FieldInfo[]

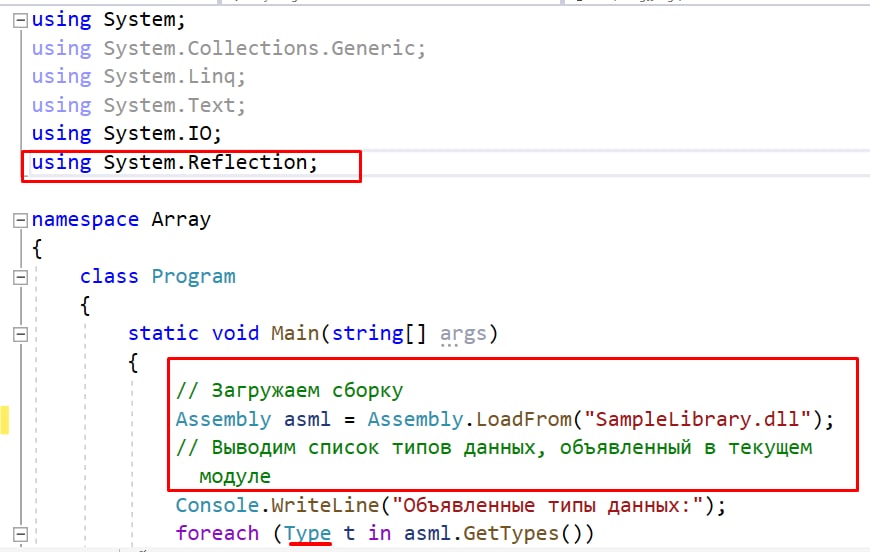
Метод **GetProperties()** - возвращает массив свойств класса PropertiesInfo[]

Метод **GetInterface()** - возвращает массив реализованных интерфейсов Type[]



## Позднее связывание. DLL

Пространство имен **System.Reflection**



class **Activator** - содержит ряд методов для создания объектов после получения типа.

## Логирование

**Логирование** - это процесс при котором состояние приложения фиксируется в отдельном лог файле

Существует большое количество готовых пакетов для логирования. Нужный пакет можно установить с помощью пакетного менеджера **NuGet**

Пакетный менеджер NuGet уже встроен в Visual Studio. Подключает выбранный пакет к текущему проекту.

После подключения пакета, с помощью **using** нужно подключить пространство имен

