[1. **Состав .NET Framework. Структура среды выполнения CLR.** 4](#_Toc123507125)

[2. **Структура управляемого модуля - portable executable (PE). Понятие и исполнение сборки. CIL.** 5](#_Toc123507126)

[3. **CTS (Common Type System). Типы данных C#. Ссылочные и типы значений.** 6](#_Toc123507127)

[4. **Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, nullобъединение** 7](#_Toc123507128)

[5. **Тип данных String: операции, литералы, пустые и нулевые строки, форматированный вывод.** 8](#_Toc123507129)

[6. **Неявная типизация – назначение и использование.** 9](#_Toc123507130)

[7. **Массивы C# одномерные, прямоугольные и ступенчатые.** 10](#_Toc123507131)

[8. **Понятие кортежей. Свойства, создание** 11](#_Toc123507132)

[9. **Принципы объектно-ориентированного программирования.** 12](#_Toc123507133)

[10. **Класс. Элементы класса. Свойства и индексаторы.** 13](#_Toc123507134)

[11. **Класс. Константы. Поля только для чтения. Инициализаторы класса.** 15](#_Toc123507135)

[12. **Спецификаторы доступа C#. Видимость типов. Доступ к членам типов.** 16](#_Toc123507136)

[**Спецификаторы доступа:** 16](#_Toc123507137)

[**Доступ к членам типов.** 16](#_Toc123507138)

[13. **Класс. Конструкторы и их свойства. Деструкторы** 17](#_Toc123507139)

[14. **Класс и методы System.Object.** 18](#_Toc123507140)

[**Методы System.Object:** 18](#_Toc123507141)

[15. **Статические методы и статические конструкторы класса.** 20](#_Toc123507142)

[16. **Статические классы. Методы расширения и правила их определения.** 21](#_Toc123507143)

[17. **Анонимные типы.** 23](#_Toc123507144)

[18. **Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы.** 24](#_Toc123507145)

[19. **Перегрузка методов и операторов. Правила перегрузки операторов.** 26](#_Toc123507146)

[20. **Операции преобразования типа. Явная и неявная форма. Ограничения.** 27](#_Toc123507147)

[21. **Вложенные типы. Вложенные объекты** 28](#_Toc123507148)

[22. **Правила наследования C#.** 28](#_Toc123507149)

[23. **Сокрытие имен при наследовании.** **Обращение к срытым членам** 29](#_Toc123507150)

[24. **Использование операций is и as** 30](#_Toc123507151)

[25. **Полиморфизм. Виртуальные методы, свойства и индексаторы. Правила переопределения.** 31](#_Toc123507152)

[26. **Понятие раннего и позднего связывания.** 31](#_Toc123507153)

[27. **Абстрактные классы и методы. Бесплодные классы.** 32](#_Toc123507154)

[28. **Структур в C#.** 33](#_Toc123507155)

[29. **Интерфейсы. Свойства интерфейсов. Реализация интерфейсов.** 35](#_Toc123507156)

[30. **Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы.** 37](#_Toc123507157)

[31. **Ковариантность интерфейсов. Контравариантность интерфейсов** 39](#_Toc123507158)

[32. **Стандартные интерфейсы .NET. Назначение и применение.** 41](#_Toc123507159)

[33. **Исключительные ситуации. Генерация и повторная генерация исключений.** 42](#_Toc123507160)

[34. **Исключительные ситуации. Варианты обработки исключений. Фильтры исключений** 44](#_Toc123507161)

[35. **Обобщения (generics). Свойства обобщений.** 46](#_Toc123507162)

[36. **Концепция ограничений обобщений. Статические члены обобщений.** 48](#_Toc123507163)

[37. **Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты.** 52](#_Toc123507164)

[38. **Анонимные функции. Лямбда-выражения.** 54](#_Toc123507165)

[39. **Обобщённые делегаты .NET. Action, Func, Predicate** 55](#_Toc123507166)

[40. **События и делегаты.** 57](#_Toc123507167)

[41. **Стандартные коллекции .NET. Типы коллекций.** 59](#_Toc123507168)

[42. **Стандартные интерфесы коллекций.** 60](#_Toc123507169)

[43. **IEnumerable и IEnumerator** 64](#_Toc123507170)

[44. **LINQ to Objects. Синтаксис. Форма. Возврат результата. Грамматика выражений запросов. Отложенные и неотлаженные операции.** 66](#_Toc123507171)

[45. **LINQ to Objects. Операции Where, Select, Take, OrderB, Join, GroupBy** 68](#_Toc123507172)

[46. **Рефлексия. System Type.** 71](#_Toc123507173)

[47. **Классы для работы с файловой системой.** 74](#_Toc123507174)

[48. **Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы.** 75](#_Toc123507175)

[49. **Классы адаптеры потоков.** 77](#_Toc123507176)

[50. **Сериализация. Форматы сериализации.** 78](#_Toc123507177)

[51. **Сериализация контрактов данных. интерфейс ISerializable.** 80](#_Toc123507178)

[52. **Атрибуты. Создание собственного атрибута.** 81](#_Toc123507179)

[53. **Процесс. Домен приложений. Поток выполнения.** 82](#_Toc123507180)

[54. **Создание потоков , классы приоритетов. Состояния потоков** 83](#_Toc123507181)

[55. **Синхронизация потоков. Lock. Monitor. Мutex. Semaphore** 85](#_Toc123507182)

[56. **Библиотека параллельных задач TPL. Класс Task. Состояние задачи.** 87](#_Toc123507183)

[57. **Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения.** 90](#_Toc123507184)

[58. **Параллелизм при императивной обработке данных. Класс Parallel** 93](#_Toc123507185)

[59. **Асинхронные методы. async и await** 94](#_Toc123507186)

[60. **Проектирование отношений. Агрегация, композиция и ассоциация** 95](#_Toc123507187)

[61. **Антипаттерны проектирования. Рефакторинг. Методы рефакторинга.** 96](#_Toc123507188)

[62. **Чистый код. Требования к именам, функциям, форматированию.** 98](#_Toc123507189)

[63. **Чистый код. Требования к классам и объектам. Закон Деметры. DTO. Избыточный код.** 100](#_Toc123507190)

# **Состав .NET Framework. Структура среды выполнения CLR.**

**Microsoft.NET (.NET Framework) –** программная платформа. Обеспечивает совместное использование разных языков программирования, а также безопасность, переносимость программ и общую модель программирования для платформы Windows. Языки, используемые на платформе: C++/CLI, C#, Visual Basic, F#, Iron Pithon, Iron Ruby, ассемблер IL. Решает проблемы, такие как: интеграция языков, работа на многих платформах, упрощенное повторное использование кода, автоматическое управление памятью, проверка безопасности типов, единый принцип обработки сбоев, взаимодействие с существующим кодом, проблеммы с версиями.

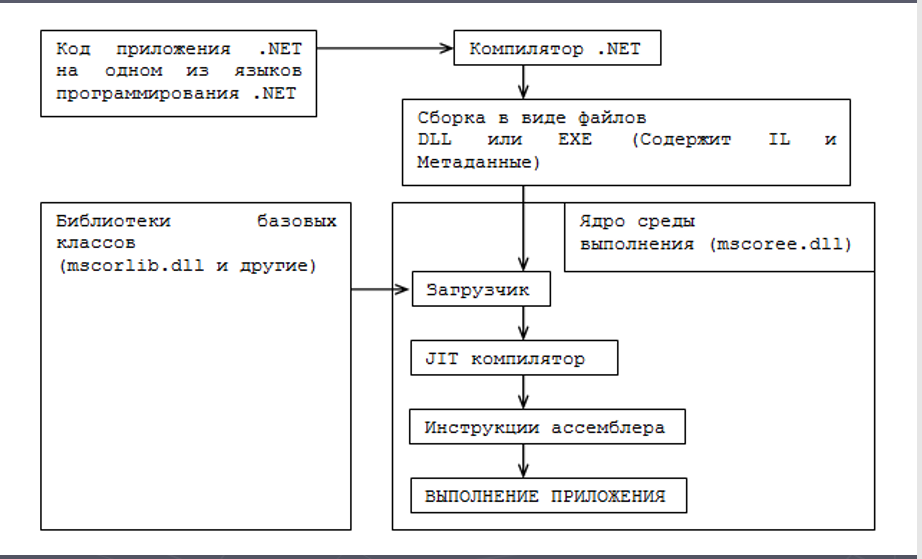
**Содержит следующие основные компоненты:**

***СLR (Common Language Runtime)*** – общеязыковая среда исполнения, виртуальная машина на которой исполняются все приложения, работающие в среде .NET. Реализация CLI VES компанией Microsoft. Компилятор JIT(Just in Time); позволяет обьектам написанном на одном языке быть равнопровными с обьектами написанными на другом.

***MSIL(Microsoft IL)*** – реализация CLI CIL компанией Microsoft.

***FCL (Framework Class Library)/BCL*** – реализация CLI BCL компанией Microsoft. Можно рассматривать, как API CRL; стандартная библиотека классов платформы .NET Framework.

Структура среды выполнения **CLR:**



# **Структура управляемого модуля - portable executable (PE). Понятие и исполнение сборки. CIL.**

Portable Executable (PE, «переносимый исполняемый»). Формат PE представляет собой структуру данных, содержащую всю информацию, необходимую PE-загрузчику для отображения файла в память. Исполняемый код включает в себя ссылки для связывания динамически загружаемых библиотек, таблицы экспорта и импорта API функций, данные для управления ресурсами и данные локальной памяти потока (TLS). В операционных системах семейства Windows NT формат PE используется для EXE, DLL, SYS (драйверов устройств) и других типов исполняемых файлов.

Секция данных **CLR** содержит два важных сегмента: сегмент метаданных и сегмент кода промежуточного языка (IL): Метаданные содержат информацию, относящуюся к сборке, включая манифест сборки. Манифест подробно описывает сборку, включая уникальный идентификатор, имя сборки, номер версии, языковые и региональные параметры.

**Код промежуточного языка** (Intermediate Language — IL) — код,содержащий набор инструкций , не зависящих от платформы .иными словами , после компиляции исходного кода он преобразует не в код для платформы , а в промежуточный код на языке il.

**Сборка** является базовой структурной единицей в .NET, на уровне которой проходит контроль версий, развертывание и конфигурация приложения.

Когда мы создаем приложение в результате компиляции в Visual Studio или в консоли, результатом этой работы является файл exe или dll (в зависимости от выбранных настроек), который называется сборкой приложения

**Исполнение сборки:**

\*CLR ищет типы данных и загружает во внутренние структуры

\*Для каждого метода CLR заносит адрес внутренней ЦЛР функции JitCompiler

\*JitCompiler ищет в метаданных соответствующей сборки IL- код вызываемого метода, проверяет и компилирует IL-код в машинные команды

\*Они хранятся в динамически выделенном блоке памяти.

\*JitCompiler Заменяет адрес вызываемого метода адресом блока памяти, содержащего готовые машинные команды

\*JitComplier передает управление коду в этом блоке памяти

# **CTS (Common Type System). Типы данных C#. Ссылочные и типы значений.**

CTS (Common Type Systems) - спецификация типов, которые должны поддерживаться всеми языками, ориентированными на CLR. Microsoft выпустил несколько компиляторов соответствующих этой спецификации: С++/CLI (C++ с управляемыми расширениями), С#, VB .NET, Jscript.

Типы данных C# поддерживает общую систему типов (CTS): для объявления того или иного встроенного типа данных из CTS обычно предусмотрено свое уникальное ключевое слов.

**Система типов С#:**

***\*Типы значения –*** Структуры, перечисления, простые типы

***\*Типы ссылки –*** классы, строки, массивы, делегаты, интерфейсы

**Типы значений** (value types)

1) Определяются struct или emun

2) Размещение в стеке потока

3) поля экземпляра размещаются в самой переменной

4) не обрабатываются cборщиком мусора. Типы значения, структуры, перечисления, простые типы.

**Ссылочные типы** (reference types). определяются class (в куче)

# **Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, nullобъединение**

Преобразование типа значений к ссылочному типу сопровождается неявной **операцией упаковки** (boxing) — помещение копии типа значений в класс-обёртку, экземпляр которого сохраняется в куче. Упаковочный тип генерируется CLR и реализует интерфейсы сохраняемого типа значения. Преобразование ссылочного типа к типу значений вызывает **операцию распаковки** (unboxing) — извлечение из упаковки копии типа значения и помещение её в стек.

Одно из отличий ссылочных типов от типов значений состоит в том, что переменные ссылочных типов могут принимать значение null.

Для этого надо использовать знак вопроса ? после типа значений. Например:

int? z = null;

bool? enabled = null;

Преобразование типов Nullable: явное, неявное, неявные расширяющие преобразования, явные сужающие преобразования

**Null-объединение**

Оператор ?? называется оператором **null-объединения**. Он применяется для установки значений по умолчанию для типов значений и ссылочных типов, которые допускают значение null. Оператор ?? возвращает левый операнд, если этот операнд не равен null. Иначе возвращается правый операнд. При этом левый операнд должен принимать null.  
При проверке объектов на равенство следует учитывать, что они равны не только, когда они имеют ненулевые значения, которые совпадают, но и когда оба объекта равны null

int? x = null;

int y = x ?? 100;  // равно 100, так как x равен null

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

int? z = 200;

int t = z ?? 44; // равно 200, так как z не равен null

**Преобразование**

**int? x1 = null;**

**if (x1.HasValue) {**

**int x2 = (int)x1;//явное**

**int? x3 = x2; //неявное к T?**

**long? x4 = x2; //неявное расширяющее**

**}**

# **Тип данных String: операции, литералы, пустые и нулевые строки, форматированный вывод.**

**Создание строк**

Создавать сроки можно, как используя переменную типа string и присваивая ей значение, так и применяя один из конструкторов класса String.

*char[] a = { '0', '0', '0' }; // создание массива символов*

*string s; // инициализация отложена*

*string t = "qqq"; // инициализация строковым литералом*

*string u = new string(' ', 20); // с пом. конструктора*

*string v = new string(a); // создание из массива символов*

**Операции для строк** ► присваивание (=); ► проверка на равенство содержимого (==); ► проверка на неравенство (!=); ► обращение по индексу ([]); ► сцепление (конкатенация) строк (+) ► <,>, >=,<= - сравнивают ссылки!!!!!!!!

**Строки** равны, если имеют одинаковое количество символов и совпадают посимвольно. **Обращаться** к отдельному элементу строки по индексу можно только для получения значения, но не для его изменения. **Строки** типа string относятся к неизменяемым типам данных. **Методы**, изменяющие содержимое строки, на самом деле создают новую копию строки. Неиспользуемые «старые» копии автоматически удаляются сборщиком мусора.

**Строковые литералы**

*String path;*

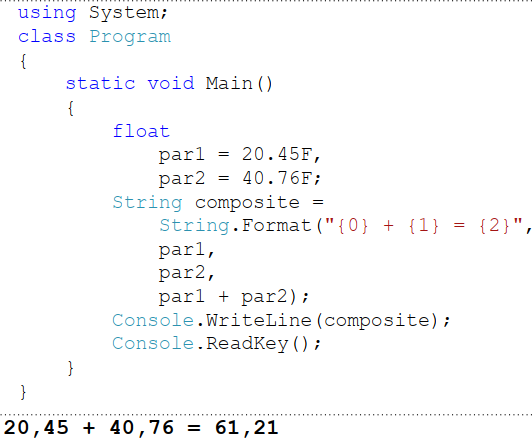
*path = "C:\\Windows\\regedit.exe"; //верно*

*path = @"C:\Windows\regedit.exe"; //верно, наиболее предпочтительно*

*path = "C:/Windows/regedit.exe"; //верно*

**Пустые строки и строки null**

**Пустая строка —** экземпляр объекта System.String, содержащий 0 символов:

**string s = "";**

Для пустых строк можно вызывать методы.

Строки со значениями null не ссылаются на экземпляр объекта System.String, попытка вызвать метод для строки null вызовет исключение NullReferenceException. строки null можно использовать в операциях объединения и сравнения с другими строками.

**Форматированный вывод. –>**

Таким образом, форматированный вывод это — преобразование значений различных типов в текстовую форму и вывод их с определённым выравниванием и определённым заполнением.

# **Неявная типизация – назначение и использование.**

Локальные переменные можно объявлять без указания конкретного типа. Ключевое слово var указывает, что компилятор должен вывести тип переменной из выражения справа от оператора инициализации. Выведенный тип может быть встроенным, анонимным. Оператор может применяться только в объявлении локальной переменной. Неявно типизированные локальные переменные не допскают множественного объявления. При объявлении обязательна инициализация.

Ранее мы явным образом указывали тип переменных, например, int x;. И компилятор при запуске уже знал, что x хранит целочисленное значение.

Однако мы можем использовать и модель неявной типизации:

var hello = "Hell to World";

var c = 20;

Console.WriteLine(c.GetType().ToString());

Console.WriteLine(hello.GetType().ToString());

Для неявной типизации вместо названия типа данных используется ключевое слово var. Затем уже при компиляции компилятор сам выводит тип данных исходя из присвоенного значения. В примере выше использовалось выражение Console.WriteLine(c.GetType().ToString());, которое позволяет нам узнать выведенный тип переменной с. Так как по умолчанию все целочисленные значения рассматриваются как значения типа int, то поэтому в итоге переменная c будет иметь тип int или System.Int32

Эти переменные подобны обычным, однако они имеют некоторые ограничения.

Во-первых, мы не можем сначала объявить неявно типизируемую переменную, а затем инициализировать.

Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной null.

# **Массивы C# одномерные, прямоугольные и ступенчатые.**

**Массив** представляет набор однотипных переменных. Объявление массива похоже на объявление переменной за тем исключением, что после указания типа ставятся квадратные скобки

**Одномерные массивы**

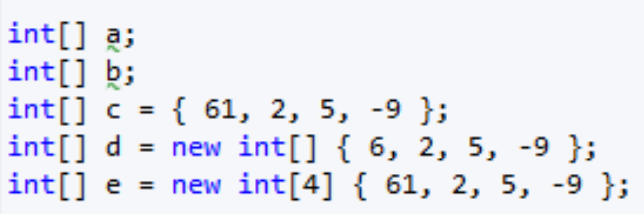
тип[ ] имя;

тип[ ] имя = new тип [ размерность ];

тип[ ] имя = { список инициализаторов };

тип[ ] имя = new тип [] { список инициализаторов };

тип[ ] имя = new тип [ размерность ] { список инициализаторов };



Цикл foreach предназначен для перебора элементов в контейнерах, в том числе в массивах. Формальное объявление цикла foreach:

int[] numbers = new int[] { 1, 2, 3, 4, 5 };

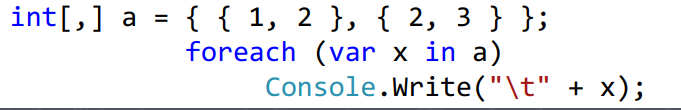
foreach (int i in numbers)

{

    Console.WriteLine(i);

}

**ПРЯМОУГОЛЬНЫЕ**



**СТУПЕНЧАТЫЕ**

# **Понятие кортежей. Свойства, создание**

**Кортежи** предоставляют удобный способ для работы с набором значений.

Кортеж представляет набор значений, заключенных в круглые скобки:

var tuple = (5, 10);

В данном случае определен кортеж tuple, который имеет два значения: 5 и 10. В дальнейшем мы можем обращаться к каждому из этих значений через поля с названиями Item[порядковый\_номер\_поля\_в\_кортеже]. Например:

static void Main(string[] args)

{    var tuple = (5, 10);     Console.WriteLine(tuple.Item1); // 5

    Console.WriteLine(tuple.Item2); // 10     tuple.Item1 += 26;

    Console.WriteLine(tuple.Item1); // 31     Console.Read(); }

**Использование кортежей** - Кортежи могут передаваться в качестве параметров в метод, могут быть возвращаемым результатом функции, либо использоваться иным образом.

Например, одной из распространенных ситуаций является возвращение из функции двух и более значений, в то время как функция можно возвращать только одно значение. И кортежи представляют оптимальный способ для решения этой задачи:

static void Main(string[] args)

{    var tuple = GetValues();

    Console.WriteLine(tuple.Item1); // 1

    Console.WriteLine(tuple.Item2); // 3

    Console.Read();}

private static (int, int) GetValues()

{    var result = (1, 3);

    return result;0..}

static Tuple<int, float, string, char> Corteg(int a, string name)

{ int sqr = a \* a;

float sqrt = (float)(Math.Sqrt(a));

string s = "Привет, " + name;

char ch = (char) (name[0]);

return Tuple.Create<int, float, string, char>(sqr, sqrt, s, ch); }

static void Main(string[] args)

{var myTuple = Corteg(25,"Alexandr");

Console.WriteLine("{0}\n25 в квадрате: {1}\nКвадратный корень из 25: "+ "{2}\nПервый символ в имени: {3}\n", myTuple.Item3, myTuple.Item1, myTuple.Item2, myTuple.Item4);

// Создаем кортеж произвольной размерности

var myTuple2 = Tuple.Create<int, char, string, decimal, float, byte, short, Tuple<int, float, string, char>>(12, 'C', "Name", 12.3892m, 0.5f, 120, 4501, myTuple);

Console. ReadLine();}

# **Принципы объектно-ориентированного программирования.**

**Объектно-ориентированное программирование** – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования

**Инкапсуляция (Encapsulation)**

Механизм, связывающий вместе данные и код, обрабатывающий эти данные, и сохраняющий их от внешнего воздействия и ошибочного использования

*Свойства инкапсуляции*

\*Совместное хранение данных и функций

\*Сокрытие внутренней информации отпользователя

\*Изоляция пользователя от особенностейреализации

**Наследование (Inheritance)**

процесс, благодаря которому один объект может наследовать (приобретать) свойства от другого объекта.

**Полиморфизм (Polymorphism)**

-способность вызывать метод потомка через экземпляр предка

-способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта

Поддержка полиморфизма осуществляется через виртуальные функции, механизм перегрузки функций и операторов, а также обобщения

**Абстракция данных(Abstraction)**

Абстракция подразумевает разделение и независимое рассмотрение интерфейса и реализации; абстракция – уровень описания/представления модели чего либо

# **Класс. Элементы класса. Свойства и индексаторы.**

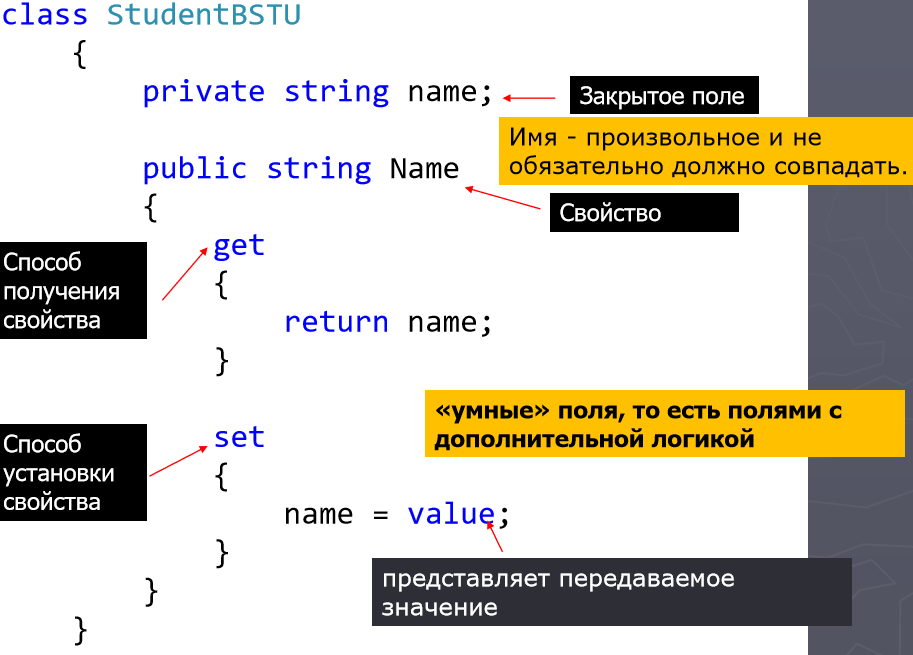
**Класс** – это шаблон, по которому определяется форма объекта.Объект – это физическая реализация класса(шаблона).

**Элементами класса** являются **Данные члены и**  **Функции-члены.**

***Данные члены*:** Поля (любые переменные, ассоциированные с классом ,Константы, События (определяют уведомления, которые может генерировать класс).

***Функции-члены*:** Методы (реализуют вычисления или другие действия, выполняемые классом или экземпляром),Свойства (определяют методы записи и чтения),Конструкторы,Финализаторы,Операции (задают действия с объектами с помощью знаков операций),Индексаторы (обеспечивают возможность доступа к элементам класса по их порядковому номеру).

**Свойства**–служат для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки (предоставляет инкапсуляцию).



*Ограничения свойств :* 1) не может быть передано методу в качестве параметра ref или out. 2) не подлежит перегрузке 3) не должно изменять состояние базовой переменной при вызове аксессора get 4) могут быть статическими, экземплярными, абстрактными и виртуальными 5) могут иметь модификатор доступа 6) могут определяться в интерфейсах

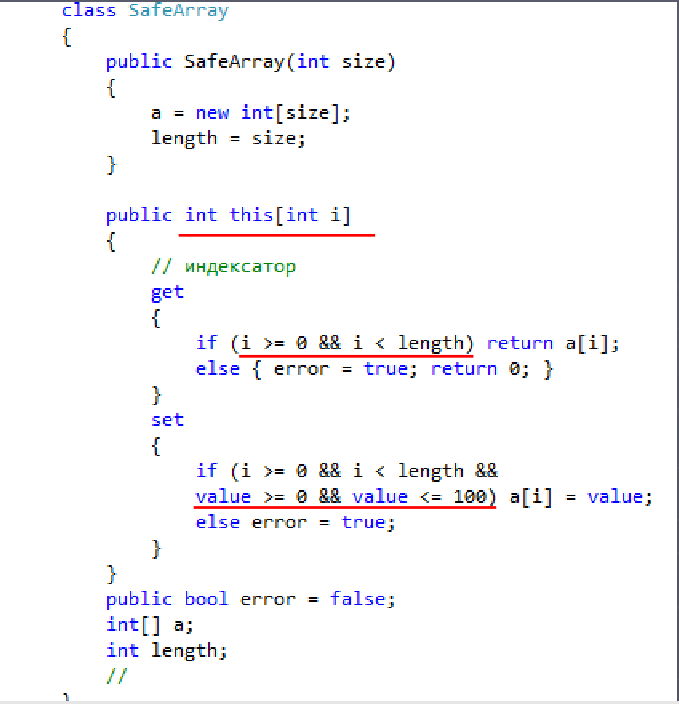
*Автоматические свойства:* тип имя { get; set; }. Компилятор автоматически реализует методы для правильного возвращения значения из поля и назначения значения полю

*Минусы использ*: неявная инициализация ,проблемы при сериализации и десериализации , во время отладки нельзя установить точку останова.

**Индексаторы**-свойства с параметрами. Позволяют индексировать объекты таким же способом, как массив или коллекцию. Позволяет разработчику перегружать оператор []

*Атрибуты спецификаторы* тип this [ список\_параметров ] get код доступа set код доступа.

*Ограничения на индексаторы*: 1) значение, выдаваемое индексатором, нельзя передавать методу в качестве параметра ref или out 2) индексатор не может быть объявлен как static. Можно перегружать.

←Пример

Многомерный индексатор



# **Класс. Константы. Поля только для чтения. Инициализаторы класса.**

**Класс** – это шаблон, по которому определяется форма объекта. Объект – это физическая реализация класса(шаблона).

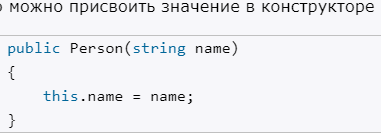
**Константы.**Значение константы обязательно устанавливается при ее объявлении в программе. В отличие от переменной значение константы не может изменяться программно. Если константа уже объявлена, то при попытке присвоить константе какое-нибудь новое значение компилятор выдаст сообщение об ошибке.



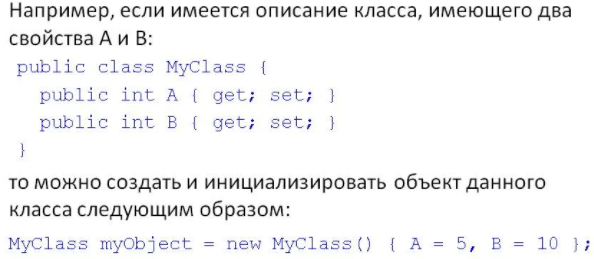
Нельзя получать адрес константы и передавать ее по ссылке. К моменту компиляции они должны быть определены.

**Поля для чтения – readonly**. Их значениянельзя изменить. Таким полям можно присвоить значение либо при при их объявлении, либо в конструкторе. В других местах программы присваивать значение таким полям нельзя, можно только считывать их значение.



**-**можно присв. значение в конструкторе

**Инициализаторы.** С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.



# **Спецификаторы доступа C#. Видимость типов. Доступ к членам типов.**

**Спецификаторы доступа:**

**public**: общедоступный класс, доступен из любого места в коде, а также из других программ и сборок.

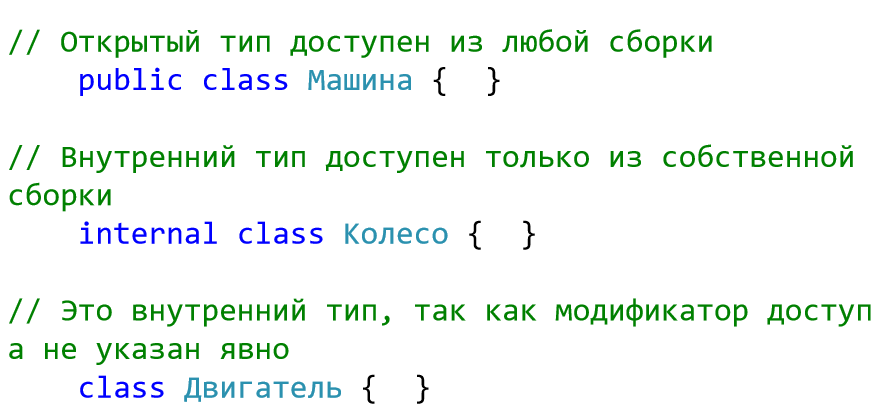
**private**: закрытый класс, доступен только из кода в том же классе или контексте.

**protected -** (используется для вложенных классов) Доступен только методам в определяющем типе (и вложенных в него типах) или в одном из его производных типов независимо от сборки

**internal -** доступ только из данной сборки

**protected internal**: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.

**Видимость типа:**может быть открытым (public) или внутренним (internal).



**Доступ к членам типов.**

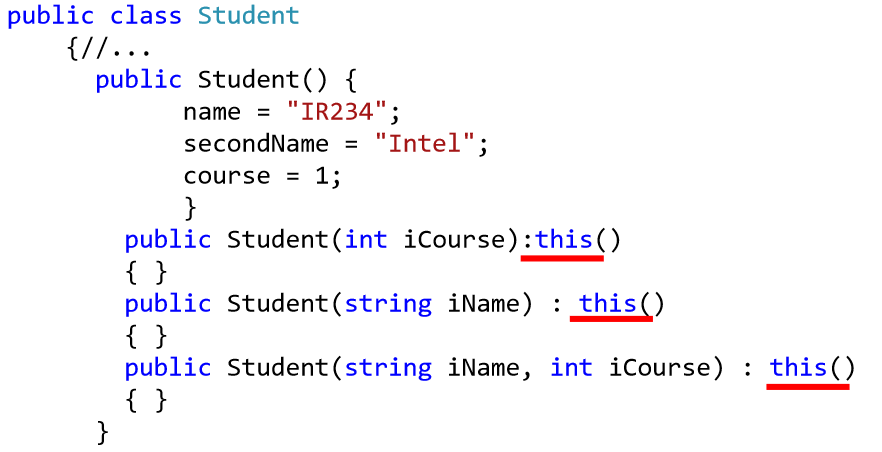
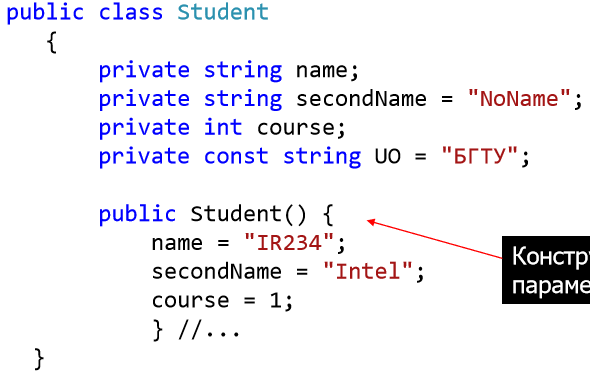
По умолчанию (если явно не указан модификатор доступа) в C# членам классов и [структур](https://programming-lessons.xyz/c-sharp/struktury-c-sharp/)назначается модификатор доступа **private**, а членам интерфейсов и [перечислений](https://programming-lessons.xyz/c-sharp/perechisleniya/)– **public**

# **Класс. Конструкторы и их свойства. Деструкторы**

**Класс** – это шаблон, по которому определяется форма объекта. Объект – это физическая реализация класса(шаблона).

**Конструкторы —** это специальные методы, позволяющие корректно инициализировать новый экземпляр типа.

**Свойства конструкторов :** 1) не имеет возвращаемого значения 2) имя такое же как и имя типа (класса)3) не наследуются 4) нельзя применять модификаторы virtual, new, override, sealed и abstract 5) для класса без явно заданных конструкторов компилятор создает конструктор по умолчанию (без параметров)6) для статических классов (запечатанных и абстрактных) компилятор не создает конструктор по умолчанию 7) может определяться несколько конструкторов, сигнатуры и уровни доступа к конструкторам обязательно должны отличаться8) можно явно заставлять один конструктор вызывать другой конструктор посредством зарезервированного слова this: обеспечивает доступ к текущему экземпляру класса .



**Деструкторы-**вызываться непосредственно перед окончательным уничтожением объекта системой "сборки мусора", чтобы гарантировать четкое окончание срока действия объекта.

~имя\_класса () {//код деструктора}

нельзя узнать, когда именно вызовется деструктор Если программа завершиться до того, как произойдет "сборка мусора", деструктор может быть вообще не вызван

**Свойства деструктора:**1.Класс может иметь только один деструктор. 2.не могут быть унаследованы или перегружены. 3.невозможно вызвать. Они запускаются автоматически. 4.Деструктор не принимает модификаторы и не имеет параметров.

# **Класс и методы System.Object.**

**Класс** – это шаблон, по которому определяется форма объекта. Объект – это физическая реализация класса(шаблона).

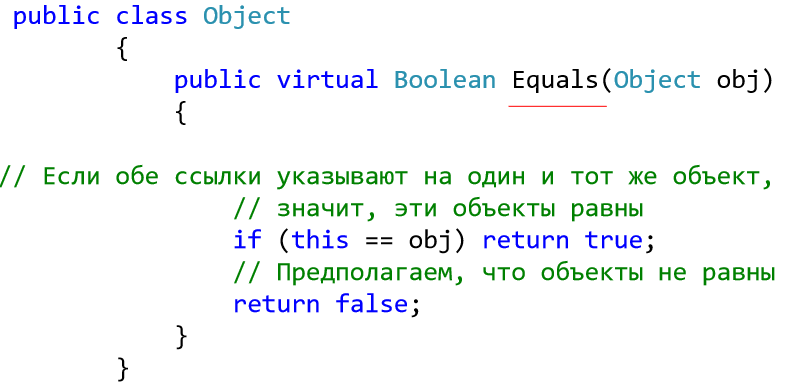
**В CLR** каждый объект (и типы значений) прямо или косвенно является производным от **System.Object** переменная ссылочного типа object может ссылаться на объект любого другого типа

**Методы System.Object:**

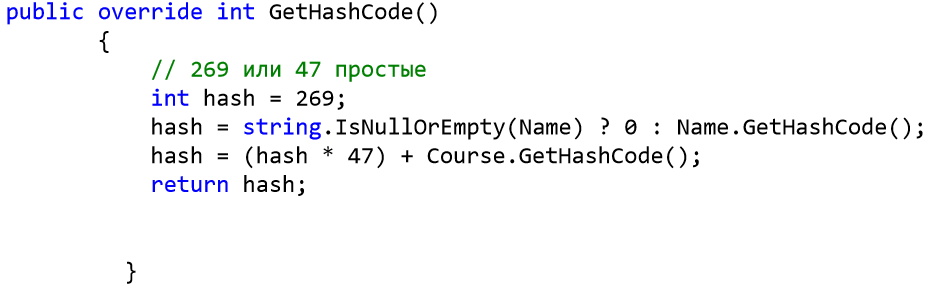
**ToString -** для получения строкового представления объекта

int year = 2017;Console.WriteLine(year.ToString());

**Equals** Равенство и тождество объектов. стандартная реализация метода Equals типа Object реализует проверку на тождество

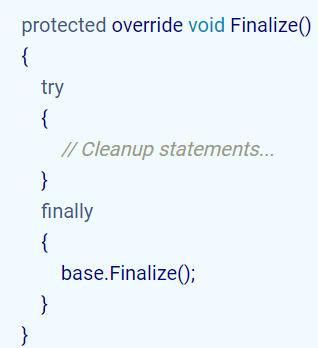


**GetHashCode** позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код.



**GetType**-позволяет получить тип данного объекта  
Console.WriteLine(ivan.GetType().Name);Возвращает объект типа Type

[**Finalize**](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.finalize?view=net-5.0)**-**Метод используется для выполнения операций очистки неуправляемых ресурсов, хранящихся в текущем объекте, до уничтожения объекта.



**Clone()** -создает копию объекта и возвращает ссылку на эту копию (неглубокое).неглубокое копирование - копируются все типы значений в классе, копируются только ссылки, а не объекты, на которые они указывают

# **Статические методы и статические конструкторы класса.**

**Статический** **метод** – это **метод**, который не имеет доступа к полям объекта, и для вызова такого **метода** не нужно создавать экземпляр (объект) класса, в котором он объявлен.

**Свойства: 1.**отсутствует ссылка this, поскольку такой метод не выполняется относительно какого-либо объекта **2.**в методе static допускается непосредственный вызов только других методов типа static **3**.для метода static непосредственно доступными оказываются только другие данные типа static, определенные в его классе

**Статический конструктор** нельзя вызывать напрямую. Если обычный конструктор вызывается в момент создания экземпляра класса, то статический – вызывается автоматически перед созданием первого экземпляра класса или перед использованием ссылки на какие-либо статические члены.

**Свойства**: закрытые автоматически ,не имеет параметров ,нельзя вызвать явным образом (вызываются до создания первого экземпляра объекта или до вызова любого статического метода).

class D

{private D() { } // закрытый конструктор

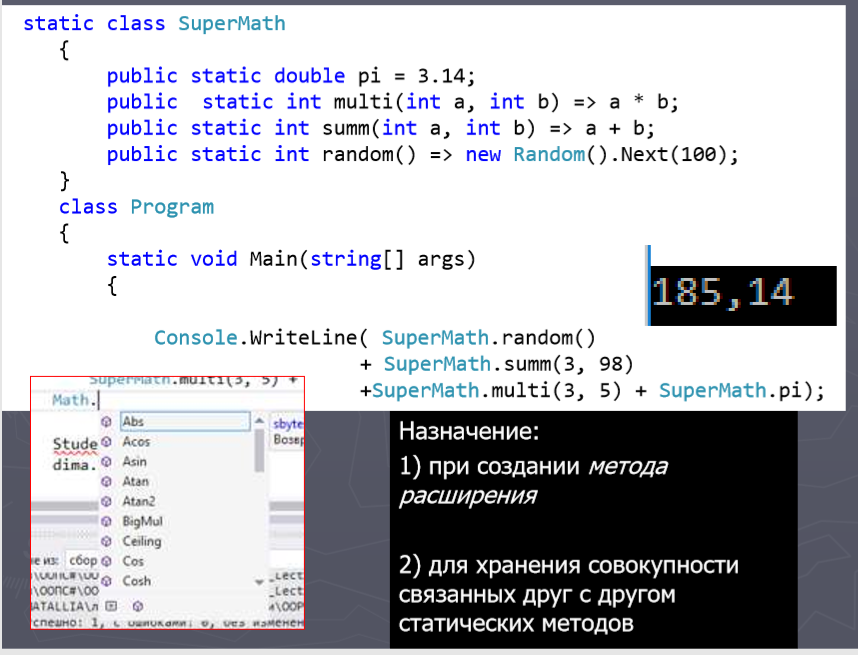
static D(){ \_a = 200; }// статический конструктор

static int \_a;}

# **Статические классы. Методы расширения и правила их определения.**

**Статический** **класс** – это **класс**, который объявляется с ключевым словом **static**. Общая форма объявления **статического** **класса**: **static** **class** ClassName {//. поля и методы **класса** // ...}

Свойства:1.прямой потомок System.Object 2.экземпляры такого класса создавать запрещено 3.не должен реализовывать никаких интерфейсов 4.нельзя использовать в качестве поля, параметра метода или локальной переменной 5.от него запрещено наследовать 6.все элементы такого класса должны явным образом объявляться с модификатором static 7.может иметь статический конструктор 8.Компилятор не создает автоматически конструктор по умолчанию

  
**Методы расширения** (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса.

**public static class NewFromAlex {**

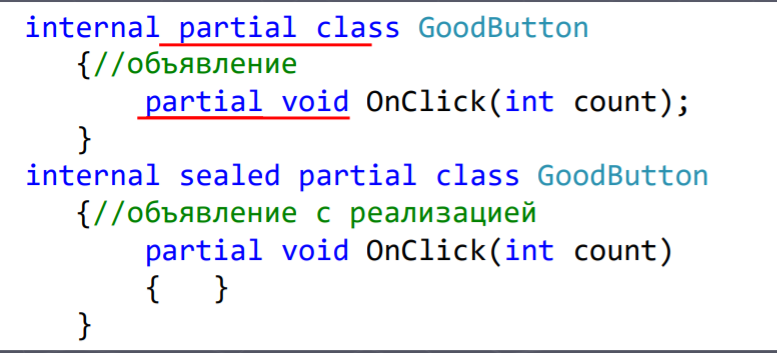
**public static bool isLetter(this String st, char a) {**

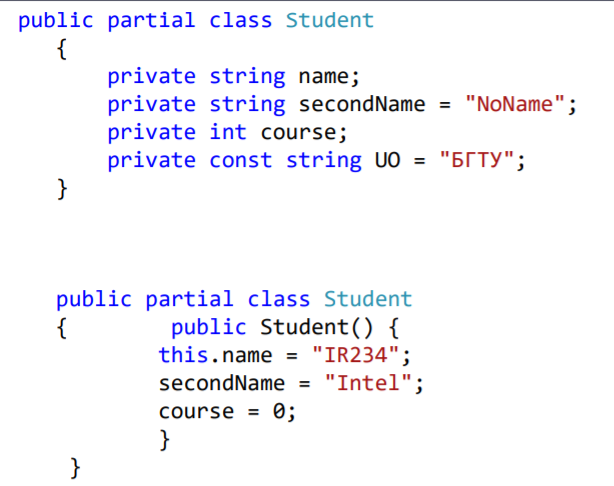
**for (Int32 index = 0; index < st.Length; index++)**

**if (st[index] == a) return true;**

**return false; } }**

1) Проверяется класс и его базовые 2) Ищется любой статический класс с методом ####, у которого первый параметр соответствует типу выражения ( this)  
**Правила для методов расширений** 1) Методы расширения должны быть объявлены в статическом необобщенном классе (первого уровня) 2) this перед первым аргументом и только один 3) использовать аккуратно





# **Анонимные типы.**

позволяют создать объект с некоторым набором свойств без определения класса (тип в одном контексте или один раз).

**var someType = new {Name = "Anna"};**

здесь var - механизм неявной типизации

не был определен тип имени, автоматически создает имя типа

компилятор определяет тип каждого выражения

создает закрытые поля

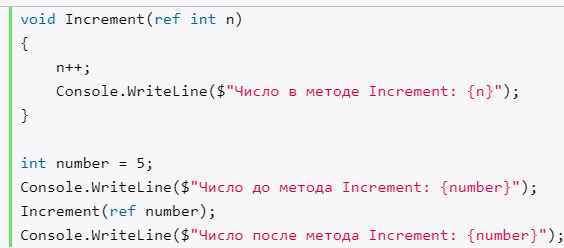
создает открытые свойства только для чтения

создает конструктор и инициализирует закрытые поля

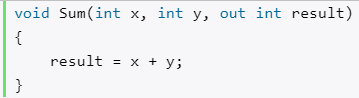
переопределяет методы Equals, GetHashCode и ToString

# **Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы.**

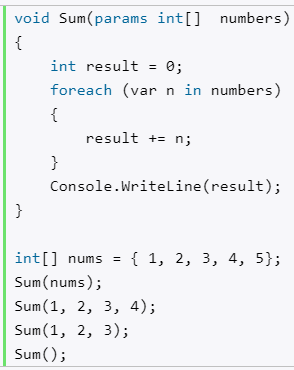
**Ref-**для передачи параметров по ссылке. Аргументу, передаваемому методу "в сопровождении" модификатора ref, должно быть присвоено значение до вызова метода.



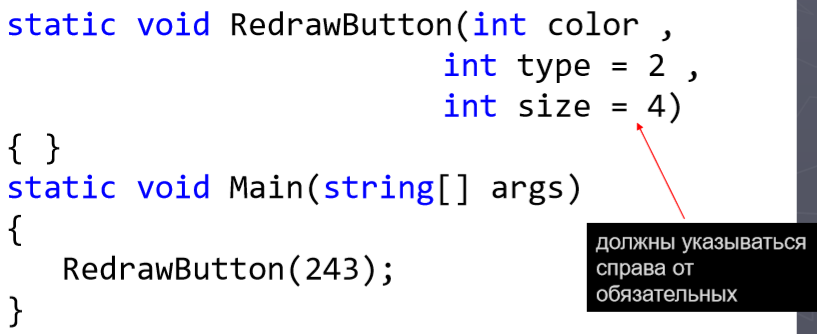
**Out-**чтобы сделать параметр выходным,перед ним ставится этот модификатор. его можно использовать для передачи значения из метода**.**out-параметр "поступает" в метод без начального значения, но метод обязательно должен присвоить этому параметру значение



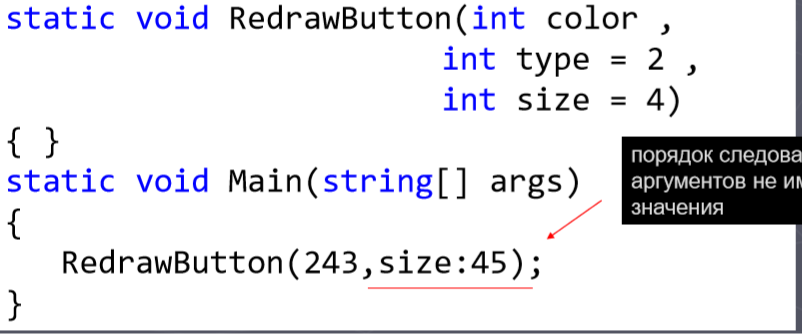
**params** позволяет передавать методу переменное количество аргументов одного типа , мы можем передавать неопределенное количество параметров:



**Необязательные аргументы -**позволяют определить используемое по умолчанию значение для параметра метода ,можно применять в конструкторах, индексаторах

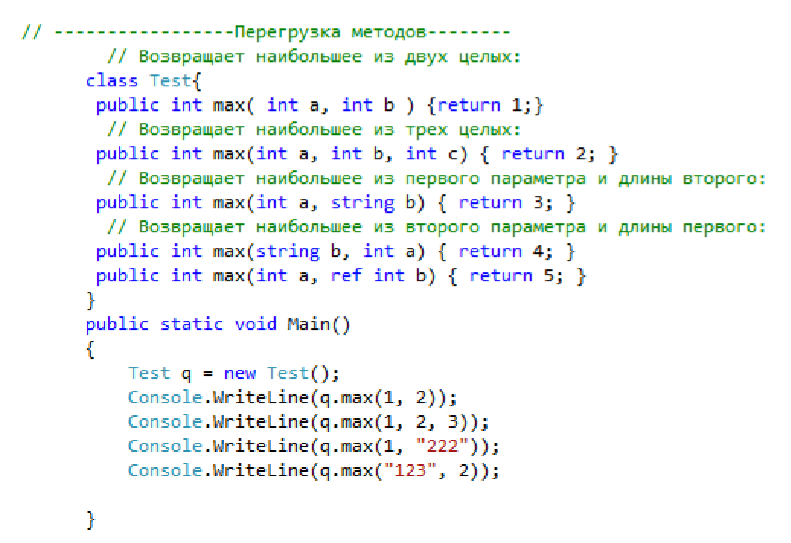


**Именованные аргументы** позволяет указать имя того параметра, которому присваивается его значение ( в конструкторах, индексаторах или делегатах.)



# **Перегрузка методов и операторов. Правила перегрузки операторов.**

**Перегрузка** **методов** – это объявление в классе **методов** с одинаковыми именами при этом с различными параметрами. 



**Перегрузка операций -**способ объявления новых операций для типа

**Спецификация CLR требует, чтобы перегруженные операторные методы были**

1) открытыми и статическими 2) тип одного из параметров или возвращаемого значения совпадал с типом, в котором определен операторный метод

Перегрузка операторов заключается в определении в классе, для объектов которого мы хотим определить оператор, специального метода:

public static возвращаемый\_тип operator оператор(параметры) {} **Операции подлежащие перегрузке:**

унарные операторы +, -, !, ~, ++, -- ;бинарные операторы +, -, \*, /, %; операции сравнения ==, !=, <, >, <=, >= ; операторы присваивания +=, -=, \*=, /=, %=; true, false (попарно)

**Операции не подлежащие перегрузке:**

[] (но есть индексатор); () (можно определить новые операторы преобразования); +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>= (но получаем автоматически в случае перегрузки бинарной операции); =, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof

**правила:** префиксные операции ++ и – – перегружаются парами;операции сравнения перегружаются парами: == и != ; < и >;<= и >=.

Перегруженные операции обязаны возвращать значения .Должны объявляться как public и static .префиксная и постфиксная формы операций ++ и --, в отличие от оригинальных операций, семантически НЕ различаются.

**если** перегружаются операторы == и !=, то для этого требуется переопределить методы Object.Equals() и Object.GetHashCode().

# **Операции преобразования типа. Явная и неявная форма. Ограничения.**

**Операции преобразования типа -**преобразует объект исходного класса в другой тип.

**Операция преобразования** типов предполагает указание в скобках того типа, к которому надо преобразовать значение:

|  |  |
| --- | --- |
|  | (тип\_данных\_в\_который\_надо\_преобразовать)значение\_для\_преобразования; |

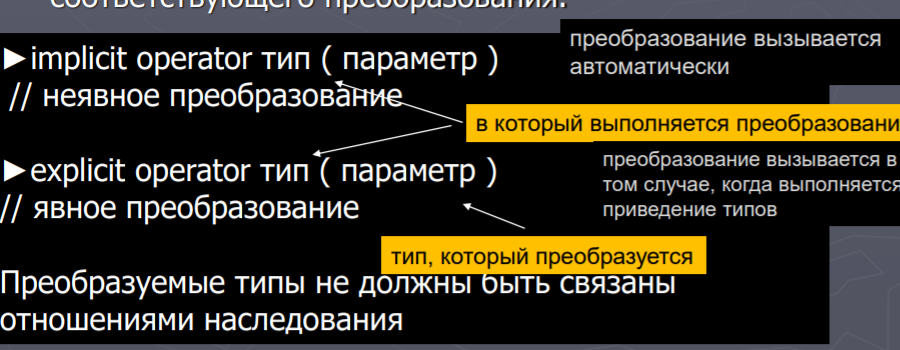
**Явная и неявна форма -** будет ли этот алгоритм выполняться неявно или необходимо будет явным образом указывать необходимость соответствующего преобразования.

**implicit operator тип ( параметр )** // неявное преобразование

**explicit operator тип ( параметр )** // явное преобразование

Преобразуемые типы не должны быть связаны отношениями наследования

Ключевые слова implicit и explicit в сигнатуру не включаются

  
**Ограничения на операторах преобразования:1.**Исходный или целевой тип преобразования должен относиться к классу, для которого объявлено данное преобразование. **2.**Нельзя указывать преобразование в класс object или же из этого класса **3.**Для одних типов данных нельзя указывать одновременно явное и неявное преобразование **4.**Нельзя указывать преобразование базового класса в производный класс**5.**Нельзя указывать преобразование в/из интерфейс

# **Вложенные типы. Вложенные объекты**

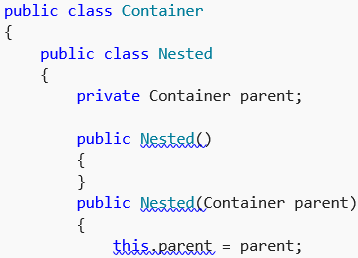
Тип, определенный внутри типа называется вложенным типом. По умолчанию являются private.

class Container

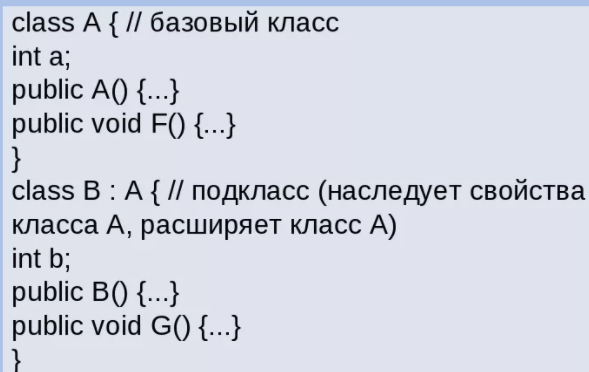
{ class Nested

{ Nested() { } }}

Вложенный тип может получить доступ к вмещающему типу, а внутренний тип — к внешнему. Чтобы получить доступ к вмещающему типу, передайте его в качестве аргумента в конструктор вложенного типа.



# **Правила наследования C#.**

**Наследование -** это механизм получения нового класса на основе уже существующего. **Роль наследования :**формирует иерархию ,поощряет повторное использование кода. **Правила наследования:**1) В C# наследование всегда подразумевается открытым2) Запрещено множественное наследование классов (но не интерфейсов) 3) наследуются все свойства, методы, поля и т.д., которые есть в базовом классе 4) Производному классу доступны public, internal, protected и protected internal члены базового класса (private – недоступны)5) не наследуются конструкторы базового класса 6) тип доступа к производному классу должен быть таким же, как и у базового класса или более строгим7) Ссылке на объект базового класса можно присвоить объект производного класса (но вызываются для него только методы и свойства, определенные в базовом классе.)

# **Сокрытие имен при наследовании. Обращение к срытым членам**

Сокрытие имен происходит, когда в базовом классе и в классе-наследнике объявлены методы с одинаковым именем. В такой ситуации метод базового класса скрывается и программа может работать не так как предусматривал программист. В таких случаях необходимо воспользоваться модификатором **new**, который скажет компилятору о вашем явном намерении скрыть метод базового класса и использовать метод, объявленный в классе наследнике.

class Square : Rectangle {public Square() {}

public Square(int size) : base(size, size) {}

public new int Width {get { return base.Width;}

set { base.Width = value; base.Height = value; }}

public new int Height {get { return base.Height;}

set { base.Width = value; base.Height = value; }}}

public static void Main(string[] args) {Square square = new Square(5);

Console.WriteLine(square.Height);//5

square.Width = 6;

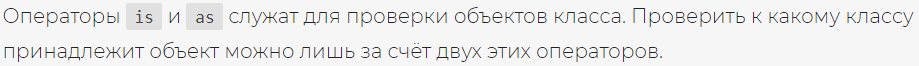
Console.WriteLine(square.Height);//6

Console.ReadKey();}

Обычное наследование всех членов базового класса в классе-наследнике.Переопределение членов базового класса в классе-наследнике (полиморфизм).Сокрытие членов базового класса в классе-наследнике

С помощью ключевого слова **base** мы можем обратиться к базовому классу. Для этого определяется метод с ключевым словом new, который скрывает реализацию данного метода из базового класса.

# **Использование операций is и as**



**is-**Возвращает булевское значение, говорящее о том, можете ли вы преобразовать данное выражение в указанный тип.Оператор is никогда не генерирует исключение.

int j = 123;

object boxed = j;

object obj = new Object();

Boolean chekJ = boxed is int; //true

Boolean checkObj = obj is int; //false

Console.WriteLine("boxed {0} System.ValueType", boxed is ValueType ? "is" : "is not");

Для null-ссылок оператор is всегда возвращает false, так как объекта, тип которого нужно проверить, не существует

**as-**позволяет преобразовывать тип в определенный ссылочный тип с применением следующего синтаксиса: операнд as <тип> **as никогда не генерирует исключения**

**Выполняется:**Если <операнд> имеет тип, заданный в <тип>. Если <операнд>, может быть неявно преобразован в <тип>. Если операнд <операнд>, может быть упакован в <тип>.

class A {}

class B : A {}

public static void Main() {

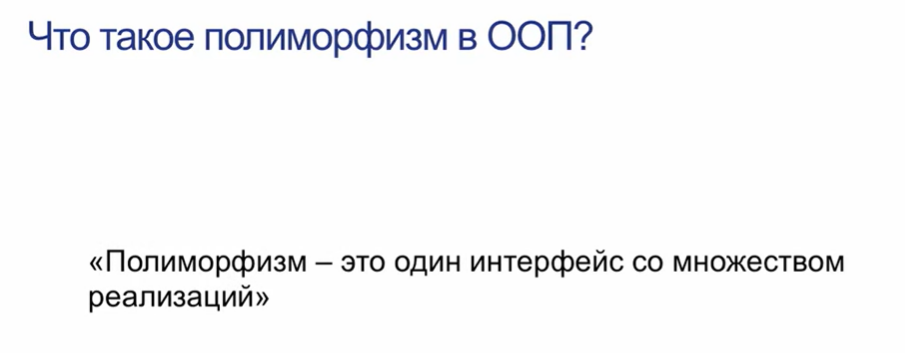
A obj1 = new A();

**B obj2 = obj1 as B; //null**

B obj3 = new B();

A obj4 = obj3 as A;

# **Полиморфизм. Виртуальные методы, свойства и индексаторы. Правила переопределения.**



**В C#** Полиморфизм достигается за счет использования абстрактных и виртуальных методов.

**Виртуальные методы, свойства, индексаторы**

полиморфный интерфейс в базовом классе - набор членов класса, которые могут быть переопределены в классе-наследнике

**virtual public void A\_method() { }**

переопределение виртуального метода в производном классе:

**override public void A\_method() { }**

**Правила переопределения** 1) Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса. 2) не может быть static или abstract 3) вызывается ближайший вариант, обнаруживаемый вверх по иерархии (многоуровневая)4)Если не virtual переопределять нельзя

# **Понятие раннего и позднего связывания.**

**связывание – это ассоциация синтаксического элемента, содержащего имя метода, с логической частью программы**.

**раннее связывание** – адрес функции назначается во время компиляции, и именно этот адрес используется при вызове функции**.   
позднее связывание (только для методов классов)** – во время выполнения приложения определяется действительный класс объекта, адрес которого находится в указателе, и вызывается метод нужного класса.  
**Виртуальные функции** предоставляют механизм позднего или динамического связывания.

|  |  |  |
| --- | --- | --- |
| Тип связывания | Достоинства | Недостатки |
| Ранее | высокое быстродействие получаемых выполнимых программ | снижение гибкости программ |
| Позднее | высокая гибкость выполняемой  программы, возможность реакции на событи | относительно низкое быстродействие программы |

# **Абстрактные классы и методы. Бесплодные классы.**

**Абстрактный класс**– это класс, содержащий методы, которые не имеют реализации.

Служит только для порождения потомков, предоставляют базовый функционал для классов-наследников. Может содержать и полностью определенные методы, переменные, конструкторы, свойства. Создавать объект абстрактного класса нельзя!!!!!!! (ссылку можно)

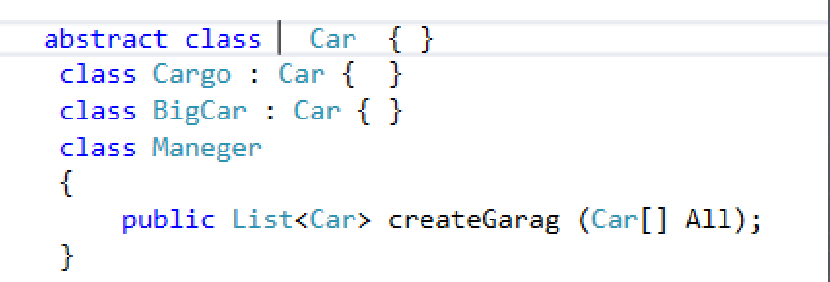
public abstract class Person { public abstract void work();}

public class Employee : Person {public override void work() { //... } }

Производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе.**Если класс** имеет хотя бы одно абстрактное свойство или метод, то он должен быть определен как абстрактный.

**Свойства abstract методов :** абстрактные методы автоматически виртуальные (virtual не ставится) ,не используются со static.

А.К. может быть параметром метода - полиморфные методы:



**Бесплодные (запечатанные) классы** -класс, от которого наследовать запрещается. Класс можно запечатать с помощью ключевого слова sealed .



Метод также может быть запечатан , и в этом случае метод не может быть переопределен. Однако метод может быть запечатан в классах, в которых они были унаследованы.

# **Структур в C#.**

Структуры – создания собственных типов данных в C#. Более того многие примитивные типы, например, int, double и т.д., по сути являются структурами. Могут содержать обычные переменные и методы. Структура относится к типу значения, т.е. размещается в стеке. Может содержать: структуру этого же типа, указатель на такую же структуру, указатель на функцию, прототип функции,перечисление, реализовывать интерфейсы, значение NULL, объявление конструкторов с параметрами.

Для определения структуры применяется ключевое слово **struct**:

struct имя\_структуры

{

    // элементы структуры - поля, методы и т.д.

}

1) ключ слово struct

2) Может иметь конструктор c параметрами.

3) нельзя определить конструктор, используемый по умолчанию (конструктор без параметров). Он определяется для всех структур автоматически и не подлежит изменению

4) Объект структуры может быть создан с помощью оператора new (или нет)

5) размещение в стеке

6) Нельзя инициализировать поля структуры при объявлении

7) нет автоматической инициализации полей компилятором

8) структуры не поддерживают наследование

Назначение : повышении эффективности и производительности программ (тип значения)

**Создание объекта структуры:**

*1) инициализация с помощью конструктора:* Для инициализации создания объектов структуры применяется вызов конструктура с оператором **new**. Даже если в коде структуры нет ни одного конструктора, есть конструктор по умолчанию, который генерируется компилятором. Этот конструктор не принимает параметров и создает объект структуры со значениями по умолчанию.

**new** название\_структуры();

Например, создадим объект структуры Person с помощью конструктора по умолчанию: Person tom = new Person();  // вызов конструктора

или так // Person tom = new();

tom.name = "Tom";   // изменяем значение по умолчанию в поле name

tom.Print();    // Имя: Tom  Возраст: 0

struct Person

{   public string name;

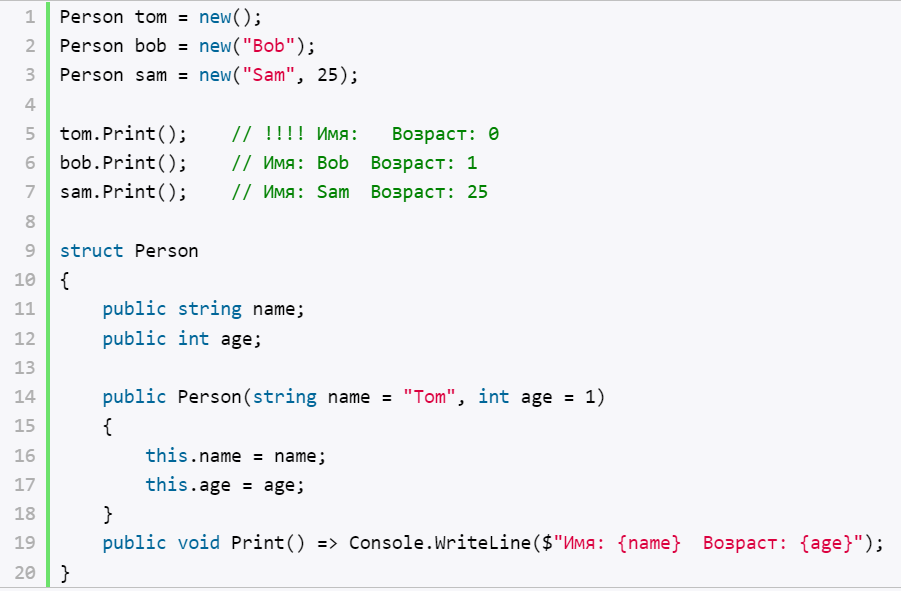
    public int age;

    public void Print()

    { Console.WriteLine($ "Имя: {name}  Возраст: {age}"); }}

*2) Непосредственная инициализация полей:* Если все поля структуры доступны (имеют модификатор **public**), то структуру можно инициализировать без вызова конструктора. В этом случае надо присвоить значения всем полям структуры перед получением значений полей и обращением к методам структуры. Например:

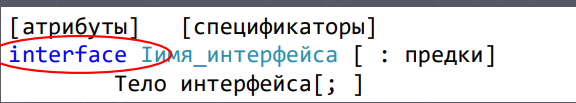
*3) Конструкторы структуры:* Как и класс, структура может определять конструкторы. Однако, если в структуре определяется конструктор, то в нем обязательно надо инициализировать все поля структуры. Например, добавим в структуру Person конструктор:



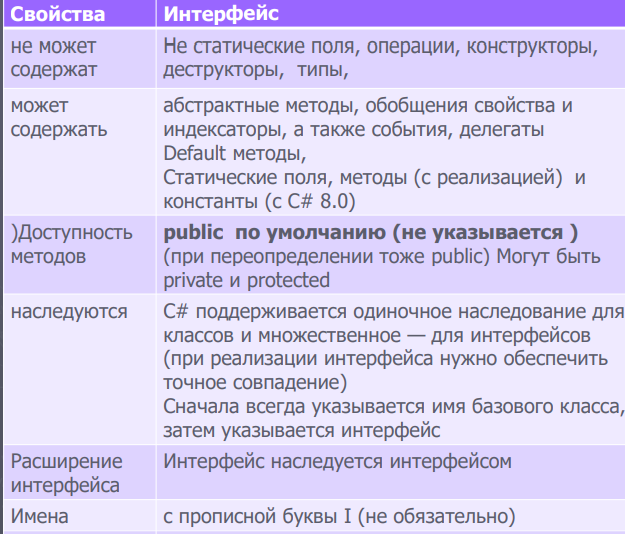
Чтобы можно было использовать переменные и методы структуры из любого места программы мы ставим перед переменными и методом модификатор доступа public. Структуру можно задать как внутри пространства имен, так и внутри класса, но не внутри метода.

# **Интерфейсы. Свойства интерфейсов. Реализация интерфейсов.**

Интерфейс представляет ссылочный тип, который может определять некоторый функционал - набор методов и свойств без реализации. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы. Для определения интерфейса используется ключевое слово interface. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, IComparable, IEnumerable. Интерфейсы могут определять следующие сущности: Методы, Свойства, Индексаторы,События, Статические поля и константы. Если его члены - методы и свойства не имеют модификаторов доступа, то фактически по умолчанию доступ public. Интерфейс или класс может наследовать свойства нескольких интерфейсов, в этом случае предки перечисляются через запятую. Назначение : задания общих характеристик объектов различных иерархий – навязывание контракта. Задается набор абстрактных методов, свойств, событий и индексаторов, которые должны быть реализованы в производных классах



***Свойства интерфейсов***



Для реализации интерфейса достаточно указать его имя после имени класса, аналогично базовому классу. Ниже приведена общая форма реализации интерфейса в классе:

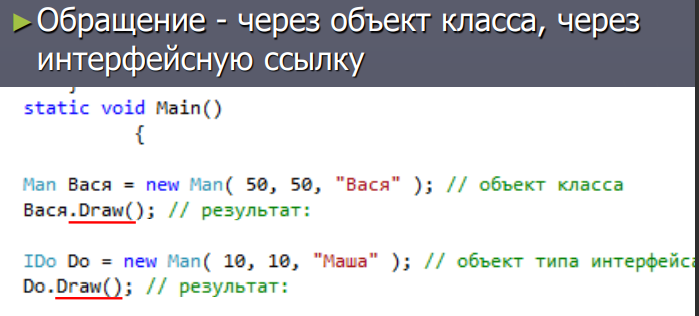
class имя\_класса : имя\_интерфейса

{    // тело класса}

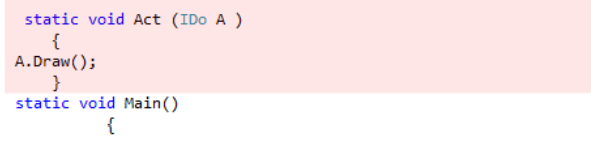
имя\_интерфейса — это конкретное имя реализуемого интерфейса.

***Реализация интерфейсов:***

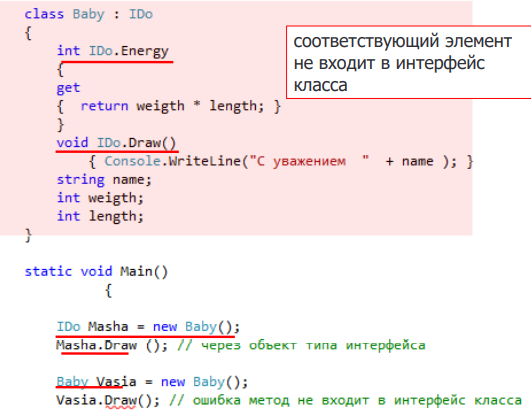
1) Обращение - через объект класса, через интерфейсную ссылку

****

2) Присваивании ссылке на интерфейс объектов различных типов (классов), поддерживающих этот интерфейс

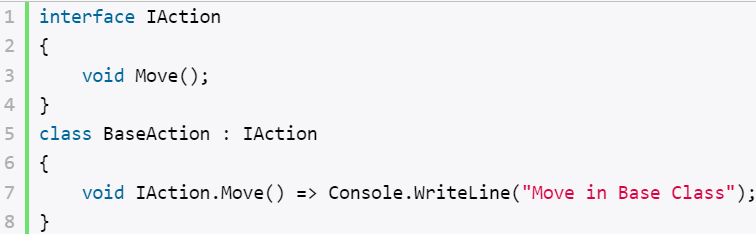
****

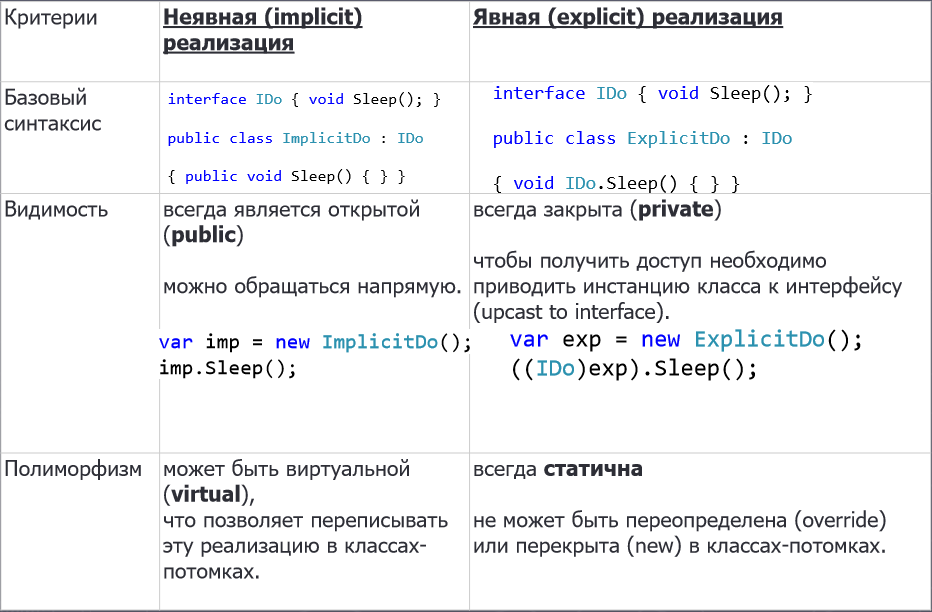
3) Явное указание имени интерфейса перед реализуемым элементом.

****

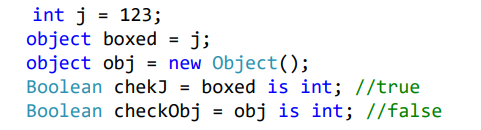
# **Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы.**

**Явная** **реализация** **интерфейса** необходима в том случае, когда несколько **интерфейсов**, реализуемых классом, имеют метод с одинаковой сигнатурой. При явной реализации указывается название метода или свойства вместе с названием интерфейса, при этом мы не можем использовать модификатор public, то есть методы являются закрытыми:

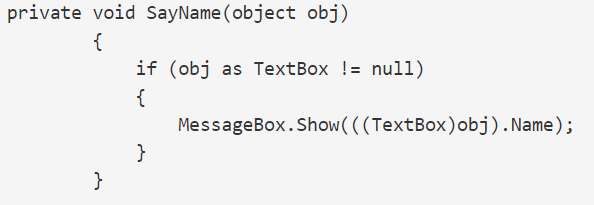
****



При работе с объектом через объект типа интерфейса бывает необходимо убедиться, что объект поддерживает данный интерфейс. Проверка выполняется с помощью бинарной операции is. Эта операция определяет, совместим ли текущий тип объекта, находящегося слева от ключевого слова is, с типом, заданным справа. Возвращает булевское значение, говорящее о том, можете ли вы преобразовать данное выражение в указанный тип

****

Недостатком использования операции is является то, что преобразование фактически выполняется дважды: при проверке и при собственно преобразовании. Более эффективной является другая операция — as. Она выполняет преобразование к заданному типу, а если это невозможно, формирует результат null, т.е. позволяет преобразовывать тип в определенный ссылочный тип с применением следующего синтаксиса: **операнд as <тип>.** as рассматривает только преобразования ссылок или преобразования с упаковкой.

****

# **Ковариантность интерфейсов. Контравариантность интерфейсов**

Понятия ковариантности и контравариантности связаны с возможностью использовать в приложении вместо некоторого типа другой тип.

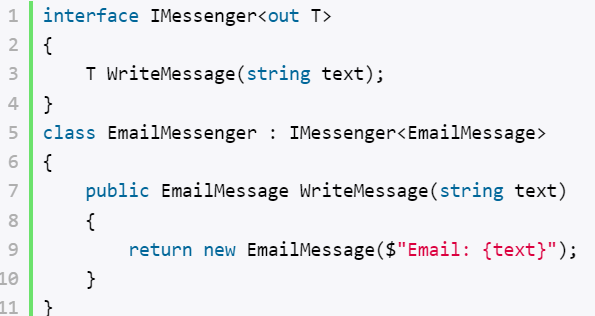
**Ковариантность**: позволяет использовать более конкретный тип, чем заданный изначально

***Ковариантность*** *интерфейсов (делегатов) 1) средство, разрешающее методу возвращать тип, производный от класса, указанного в параметре типа 2) для интерфейсов и делегатов 3) распространяться только на тип, возвращаемый методом 4) только для ссылочных типов 5) ковариантный тип нельзя использовать в качестве ограничения в интерфейсном методе.*

**Контравариантность**: позволяет использовать более универсальный тип, чем заданный изначально

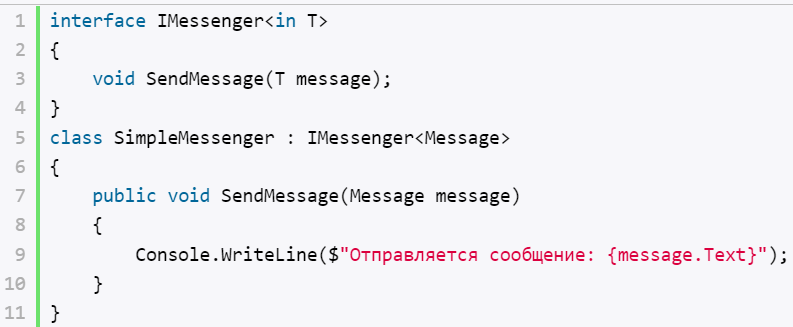
*Контравариантность интерфейсов (делегатов) 1) средство, разрешающим методу использовать аргумент, тип которого относится к базовому классу, указанному в соответствующем параметре типа 2) для ссылочных типов 3) параметр контравариантного типа можно применять только к аргументам методов 4)разрешает методу использовать аргумент, тип которого относится к базовому классу*

Обобщенные интерфейсы могут быть ***ковариантными***, если к универсальному параметру применяется ключевое слово **out**. Такой параметр должен представлять тип объекта, который возвращается из метода. Например:



Здесь обобщенный интерфейс IMessenger представляет интерфейс мессенджера и определяет метод WriteMessage() для создания сообщения. При этом на момент определения интерфейса мы не знаем, объект какого типа будет возвращаться в этом методе. Ключевое слово out в определении интерфейса указывает, что данный интерфейс будет ковариантным.

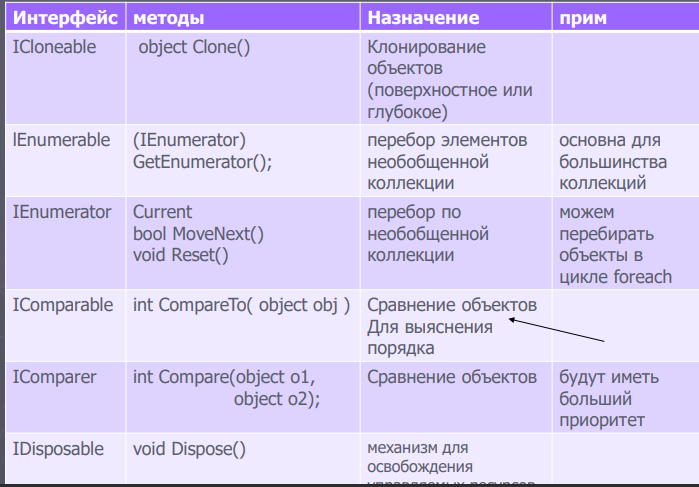
Для создания ***контравариантного*** интерфейса надо использовать ключевое слово **in**

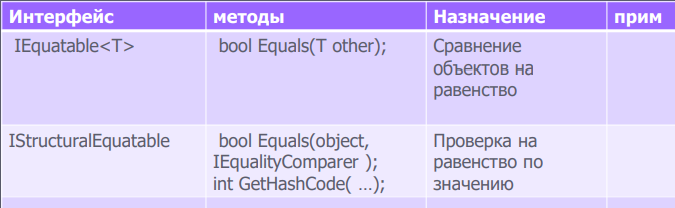


Здесь интерфейс IMessenger представляет интерфейс мессенджера и определяет метод SendMessage() для отправки условного сообщения.Класс SimpleMessenger представляет условную программу отправки сообщений и реализует этот интерфейс. В качестве типа этот класс использует тип Message. То есть SimpleMessenger фактически представляет тип IMessenger<Message>.

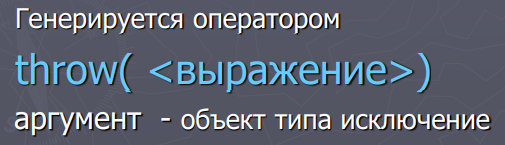
# **Стандартные интерфейсы .NET. Назначение и применение.**

В библиотеке классов .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов.





# **Исключительные ситуации. Генерация и повторная генерация исключений.**

***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.). Чтобы вручную сгенерировать исключение необходимо использовать оператор **throw**.

Под генерацией исключений подразумевается **создание объектов, описывающих исключительные ситуации, и информирование (с их помощью) системы о том, что произошла нештатная ситуация во время выполнения программы.**

Оператор **throw** информирует систему, прерывает выполнение программы, и «выбрасывает» объект специфического класса (класса «*Exception*» из пространства имен «*System»*, либо его наследника) с информацией о причине сбоя.

Для обработки исключений предназначена конструкция**try{}catch{}finally{}.**

Изображение выглядит как текст

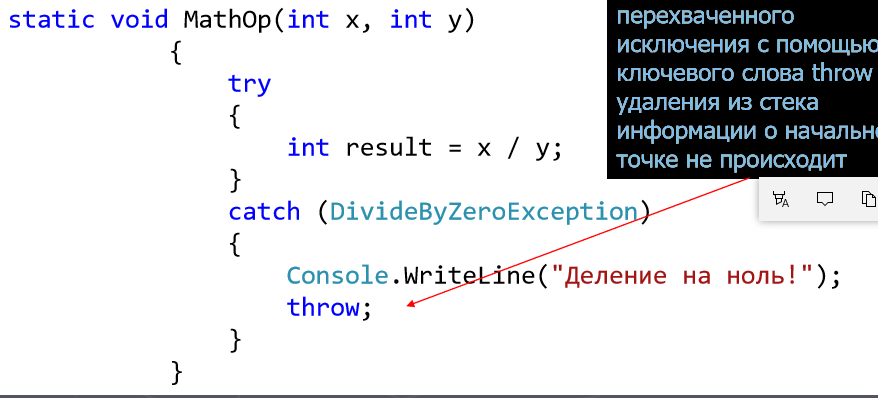
Автоматически созданное описаниеВначале выполняются все инструкции в блоке **try.** Если в этом блоке не возникло исключений, то после его выполнения начинает выполняться блок **finally.** И затем конструкция завершает свою работу.

Если же в блоке **try** возникает исключение, то среда CLR начинает искать блок **catch,** который может обработать это исключение. Если такой блок найден, то после его завершения выполняется блок **finally.** Если же нужный блок **catch** не найден, то программа аварийно завершает свое выполнение.

- try – контролируемый блок  
- throw - генерация искл. ситуации внутри try   
- catch – обработчики исключений, идут за try (несколько)   
- finally - код, очищающий ресурсы и др. действия (выполняется всегда) (один на один try)

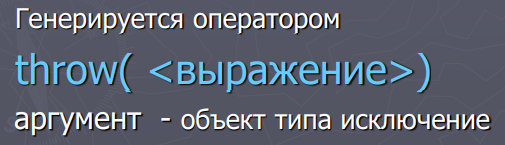
***Повторная генерация исключения*** создание нового объекта посредством повторного использования старого с помощью оператора **throw** **без параметров.**

Для этого нужно просто использовать ключевое слово throw внутри блока catch без указания какого-либо идентификатора:



# **Исключительные ситуации. Варианты обработки исключений. Фильтры исключений**

***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.). Чтобы вручную сгенерировать исключение необходимо использовать оператор **throw**.

****

Под генерацией исключений подразумевается **создание объектов, описывающих исключительные ситуации, и информирование (с их помощью) системы о том, что произошла нештатная ситуация во время выполнения программы.**

Оператор **throw** информирует систему, прерывает выполнение программы, и «выбрасывает» объект специфического класса (класса «*Exception*» из пространства имен «*System»*, либо его наследника) с информацией о причине сбоя.

Для обработки исключений предназначена конструкция**try{}catch{}finally{}.**

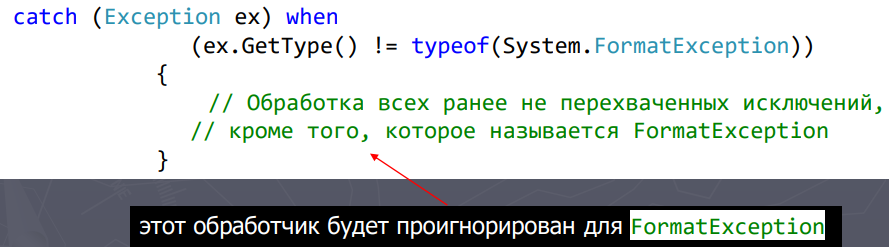
Изображение выглядит как текст

Автоматически созданное описаниеВначале выполняются все инструкции в блоке **try.** Если в этом блоке не возникло исключений, то после его выполнения начинает выполняться блок **finally.** И затем конструкция завершает свою работу.

Если же в блоке **try** возникает исключение, то среда CLR начинает искать блок **catch,** который может обработать это исключение. Если такой блок найден, то после его завершения выполняется блок **finally.** Если же нужный блок **catch** не найден, то программа аварийно завершает свое выполнение.

- try – контролируемый блок  
- throw - генерация искл. ситуации внутри try   
- catch – обработчики исключений, идут за try (несколько)   
- finally - код, очищающий ресурсы и др. действия (выполняется всегда) (один на один try)

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



Фильтры исключений позволяют обрабатывать исключения в зависимости от определенных условий. Для их применения после выражения catch идет выражение when, после которого в скобках указывается условие:

catch when(условие) {}

В этом случае обработка исключения в блоке catch производится только в том случае, если условие в выражении when истинно. Например:

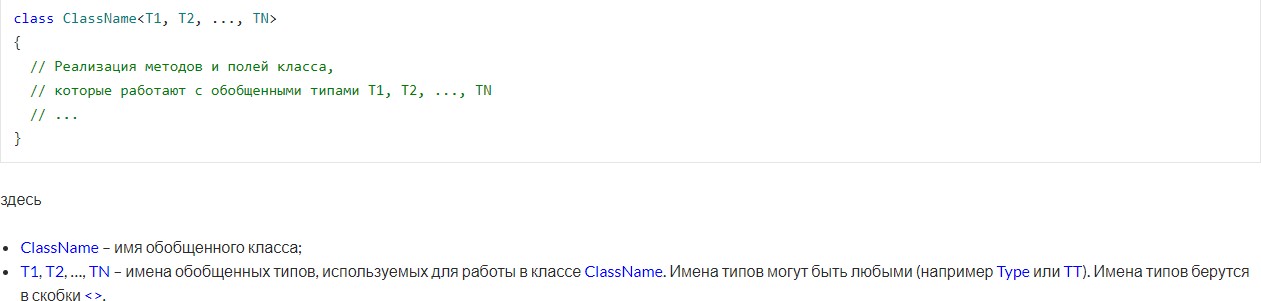
Изображение выглядит как текст

Автоматически созданное описание

В данном случае будет выброшено исключение, так как y=0. Поскольку для первого блока указано условие **y == 0**, то оно не будет обрабатывать исключение - условие, указанное после оператора when возвращает false. Поэтому CLR будет дальше искать соответствующие блоки catch и для обработки исключения выберет второй блок catch. Если убрать второй блок catch, то исключение вообще не будет обрабатываться.

# **Обобщения (generics). Свойства обобщений.**

***Обобщение***— средство языка C#, позволяющее создавать программный код, содержащий единственное (типизированное) решение задачи для различных типов, с его последующим применением для любого конкретного типа(int, float, char и т.д.). *Обобщение* – **параметризированный тип**. Роль параметризированных типов – **они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра.** С помощью обобщений можно, например, создать единый класс, который автоматически становится пригодным для обработки разнотипных данных.

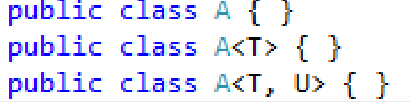
**Общая форма объявления обобщенного класса следующая:**

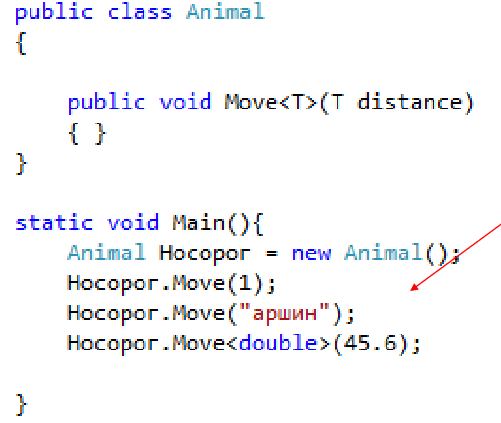
**ClassName –** имя обобщенного класса;

**T1,T2…TN –** имена обобщенных типов, используемых для работы в классе ClassName. Имена типов могут быть любыми(например Type или TT) и берутся в скобки.

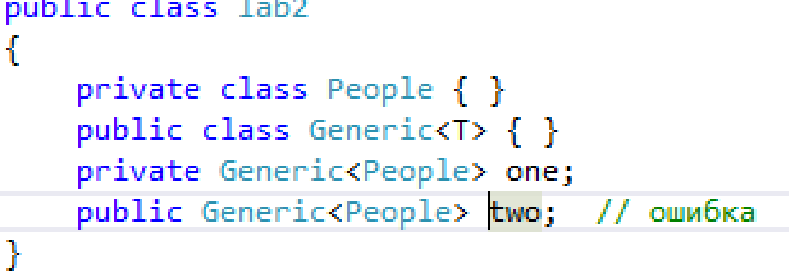
***Свойства*:**

*1)Обобщенный тип может содержать другой обобщенный тип*

*2) Обобщенные типы перегружаются на основе количества параметров*

*3) Обобщенными могут быть классы, структуры , интерфейсы, делегаты, методы public void Method <R> (A<R> iA, B<R,T> iB)  
логическое выведение типов (type inference) используется тип данных переменной, а не фактический тип объекта, на который ссылается*

*4) Могут содержать статические типы*

   
*5) Доступность конструируемых типов определяется на основе пересечения доступности обобщённого типа и типа в списке аргументов*

*5) Могут использовать несколько универсальных параметров одновременно  
6) поддерживает механизм ограничений*

# **Концепция ограничений обобщений. Статические члены обобщений.**

В С# предусмотрен ряд ограничений на типы данных:

***1) Ограничение на базовый класс***

Требует наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса.

**where T : имя\_базового\_класса**

***T*** обозначает имя параметра типа,

***имя\_\_базового\_класса*** — конкретное имя ограничиваемого базового класса. Одновременно в этой форме ограничения может быть указан только один базовый класс.

// Обобщенный класс использующий ограничение на базовый класс

class Info<T> where T : UserInfo

***2) Ограничение на интерфейс***

Требует реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.

class ClassName<T> where T : Interface1, Interface2, ... InterfaceN

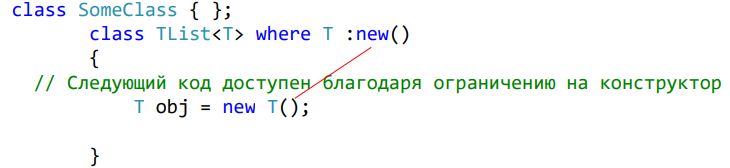
{...}

Interface1, Interface2, InterfaceN – перечень имен интерфейсов, которые должен реализовать тип T.

***3) Ограничение на конструктор***

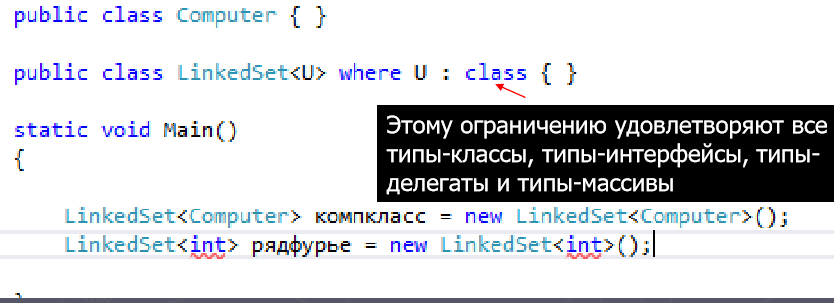
Требует предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора new() – гарантирует компилятору, что указанный аргумент-тип будет иметь неабстрактный тип, имеющий открытый конструктор без параметров.

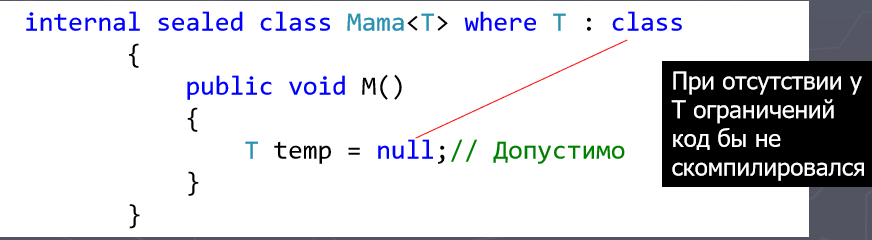
class GenClass<T> where T : new()



***4) Ограничение ссылочного типа***

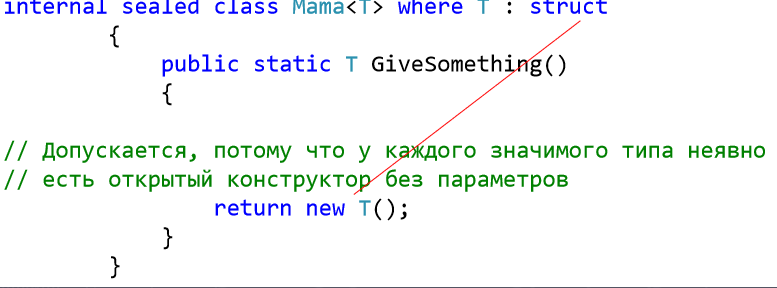
Требует указывать аргумент ссылочного типа с помощью оператора class.



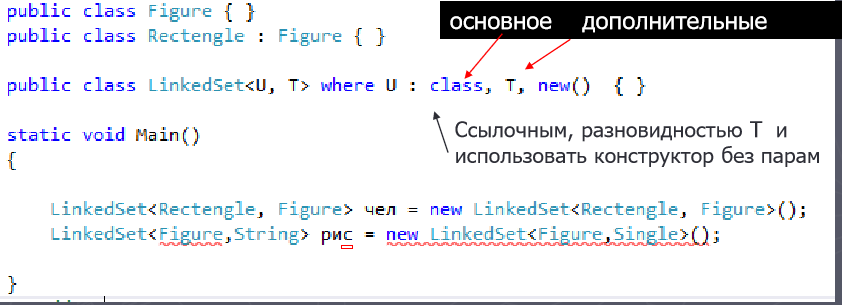


***5) Ограничение типа значения***

Требует указывать аргумент типа значения с помощью оператора struct.



***6) Ограничение на связь параметров***

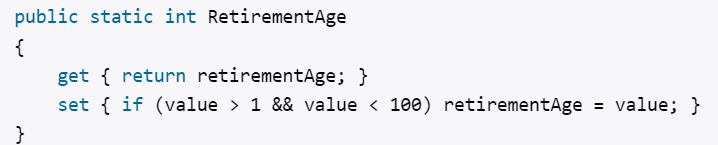


**Статические члены**

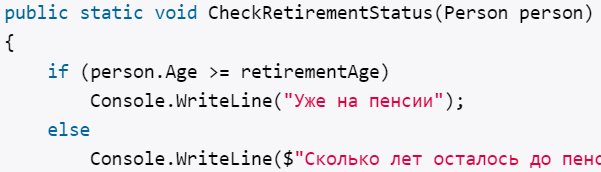
Статические поля, методы, свойства относятся ко всему классу/всей структуре и для обращения к подобным членам необязательно создавать экземпляр класса / структуры.

Статическое ***поле*** определяется как и обычное, только перед типом поля указывается ключевое слово static.  

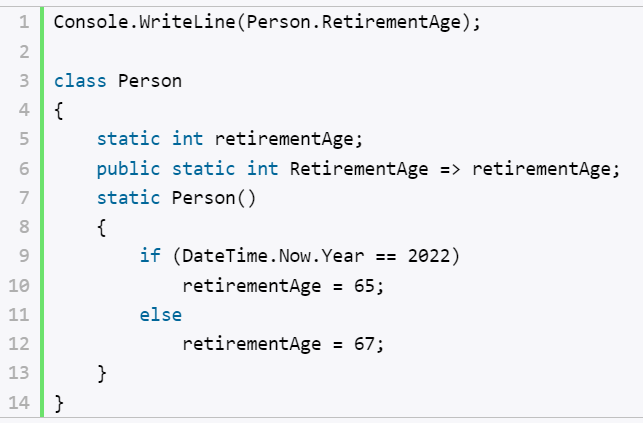

Статическое ***ствойство***



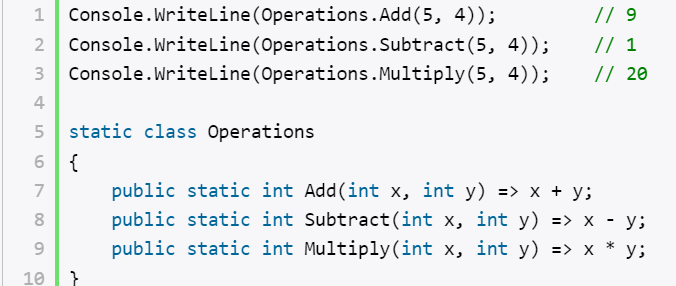
Статические ***методы*** определяют общее для всех объектов поведение, которое не зависит от конкретного объекта. Для обращения к статическим методам также применяется имя класса / структуры:



Кроме обычных конструкторов у класса также могут быть ***статические конструкторы***. Статические конструкторы имеют следующие отличительные черты: 1)Статические конструкторы не должны иметь модификатор доступа и не принимают параметров 2)Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса 3)Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются) 4)Статические конструкторы обычно используются для инициализации статических данных, либо же выполняют действия, которые требуется выполнить только один раз



Статические ***классы*** объявляются с модификатором static и могут содержать только статические поля, свойства и методы.

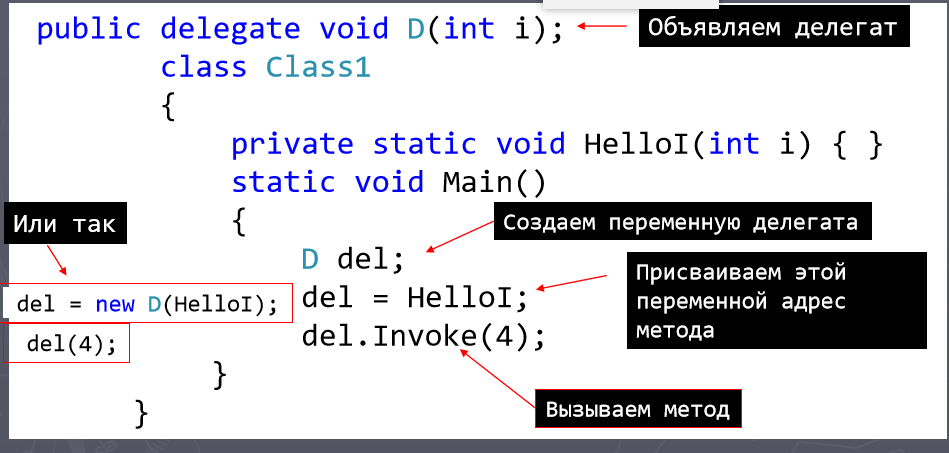


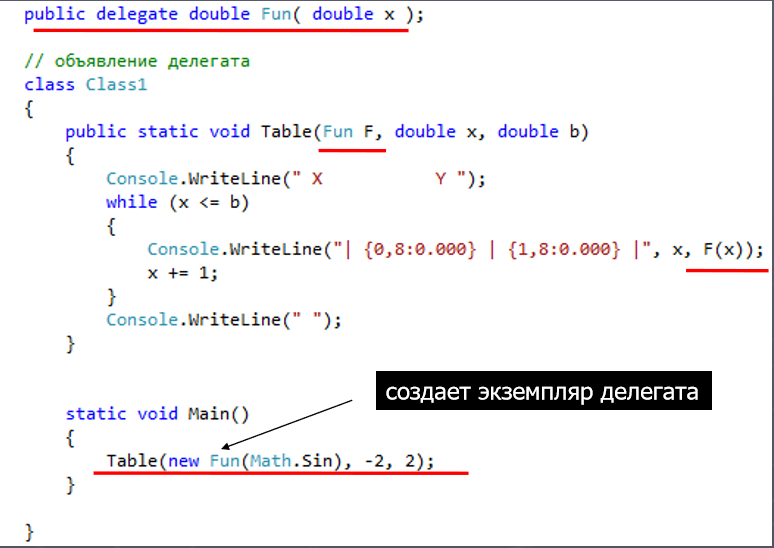
# **Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты.**

***Делегаты*** – объекты, предназначенные для хранения ссылок на методы. Это указатели на методы и с помощью делегатов мы можем вызывать методы. Для объявления делегата используется ключ слово **delegate**. **delegate, возвращаемый тип, название параметра**. (пример: delegate string strMod(string stx); Делегат может хранить ссылки на несколько методов и вызывать их поочередно - сигнатуры всех методов должны совпадать.

***Назначение***: Делегат ссылается на метод и после назначения метода ведет себя идентично ему. Делегат можно использовать как любую ф-ию с параметром и возвращаемым значением. 1)возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы; 2)обеспечения связи между объектами по типу «источник — наблюдатель»; 3)создания универсальных методов, в которые можно передавать другие методы; 4)поддержки механизма обратных вызовов(callback)

***Использование делегатов***: Чтобы использовать делегат, нам надо создать его объект с помощью конструктора, в который мы передаем адрес метода, вызываемого делегатом



**Передача делегатов в методы**

Операции над делегатами: 1)можно сравнивать на равенство и неравенство (не содержат ссылок или если ссылки на одни и те же методы в одном и том же порядке) 2)выполнять операции простого и составного присваивания (один тип д.и.) 3)является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.

Групповая адресация – создание списка, или цепочки вызовов, для методов, которые вызываются автоматически при обращении к делегату. Для организации групповой адресации методов с помощью делегата используются 4 операции:

операции ‘+’ и ‘+=’ – добавляют метод к списку методов;

операции ‘−’ и ‘−=’ – удаляют метод из списка методов.

Если в делегате сформирован список методов возвращающих некоторый тип, отличный от типа void, тогда результатом вызова будет результат возвращаемый **последним методом** в списке.

Если делегат возвращает тип void, то при вызове делегата будут **вызываться все методы**, которые сформированы в список (цепочку) с помощью групповой адресации.

# **Анонимные функции. Лямбда-выражения.**

***Анонимные функции -*** представляет собой безымянный кодовый блок, передаваемый конструктору делегата 1) Анонимные методы 2) Лямбда – выражения, Параметры должны соответствовать параметрам делегата, может не содержать никаких параметров, метод имеет доступ ко всем переменным, определенным во внешнем коде.

***Лямбда-выражение*** – упрощенная запись анонимных методов. Позволяет создать емкие методы, к-е могут возвращать некоторое значение и к-е можно передать в качестве параметров в др методы.

**параметр => выражение**

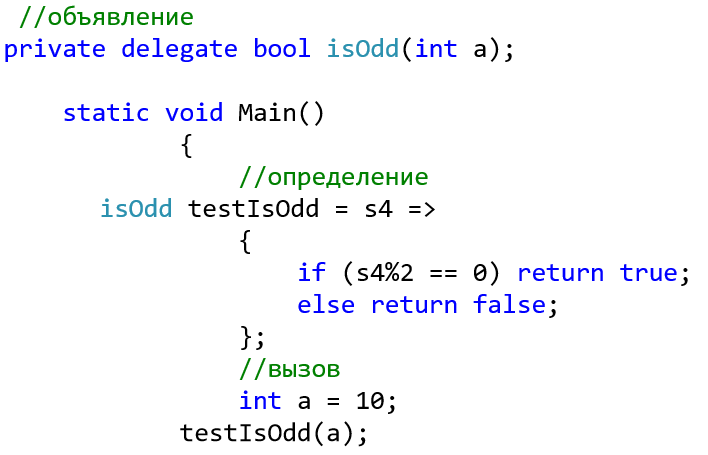
**i => i \* i;**

**(список\_параметров) => выражение**

**(x, y) => x + y;**

не надо указывать тип параметров, не надо использовать оператор return

**Блочные лямбда-выражения**



►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

# **Обобщённые делегаты .NET. Action, Func, Predicate**

В .NET есть несколько встроенных делегатов, которые используются в различных ситуациях. И наиболее используемыми являются ***Action, Predicate и Func.***

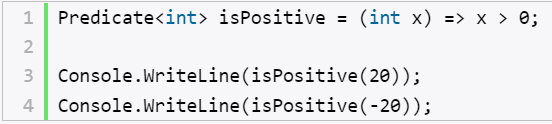
Делегат **Action** представляет некоторое действие, которое ничего не возвращает, то есть в качестве возвращаемого типа имеет тип void:



Делегат **Predicate<T>** принимает один параметр и возвращает значение типа bool:

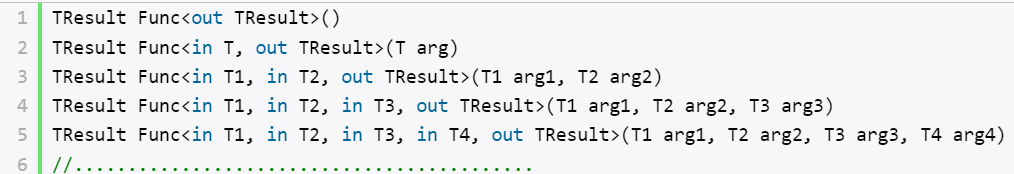


Как правило, используется для сравнения, сопоставления некоторого объекта T определенному условию. В качестве выходного результата возвращается значение true, если условие соблюдено, и false, если не соблюдено:

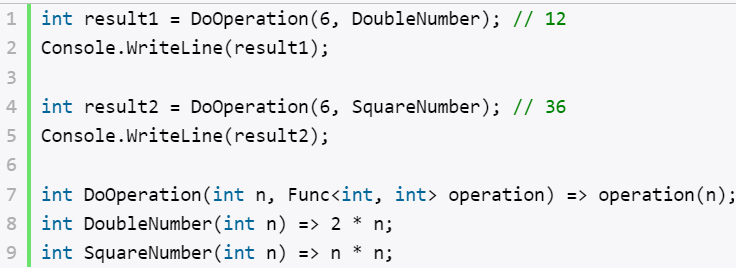


В данном случае возвращается true или false в зависимости от того, больше нуля число или нет.

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.



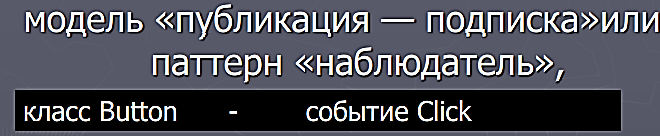
Данный делегат также часто используется в качестве параметра в методах:



Метод DoOperation() в качестве параметра принимает делегат Func<int, int>, то есть ссылку на метод, который принимает число int и возвращает также значение int. При первом вызове метода DoOperation() ему передается ссылка на метод DoubleNumber, который увеличивает число в два раза. Во втором случае передается метод SquareNumber - опять же метод, который принимает параметр типа int и возвращает результат в виде значения int

# **События и делегаты.**

**События -** это элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния.

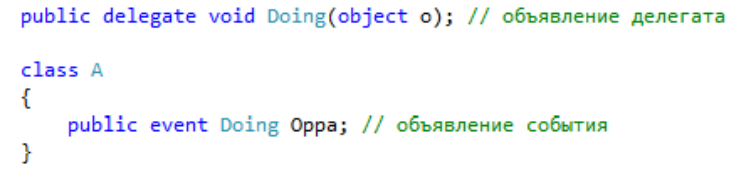
 класс, являющийся *отправителем* (sender) сообщения, публикует события, которые он может инициировать, а другие классы, являющиеся *получателями* (receivers) сообщения, подписываются на получение этих событий.

*События построены* на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому создание события в классе состоит из следующих частей:

1)описание делегата, задающего сигнатуру обработчиков событий;

2)описание события;

3)описание метода (методов), инициирующих событие.

****

Свойства: 1) Обработка событий выполняется в классах-получателях 2) сигнатура методов-обработчиков событий == типу делегата 3) Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод: += (-=) 4) Поддерживается групповая адресация

***Делегаты*** – объекты, предназначенные для хранения ссылок на методы. Это указатели на методы и с помощью делегатов мы можем вызывать методы. Для объявления делегата используется ключ слово **delegate**. **delegate, возвращаемый тип, название параметра**. (пример: delegate string strMod(string stx); Делегат может хранить ссылки на несколько методов и вызывать их поочередно - сигнатуры всех методов должны совпадать.

***Назначение***: Делегат ссылается на метод и после назначения метода ведет себя идентично ему. Делегат можно использовать как любую ф-ию с параметром и возвращаемым значением.

**Стандартные .NET делегаты**

*правила*: 1)имя делегата заканчивается суффиксом EventHandler; 2)делегат получает два параметра: первый параметр задает источник события и имеет тип object; второй параметр задает аргументы события и имеет тип EventArgs или производный от него. 3)Имя обработчика события принято составлять из префикса On и имени события;4)Классы содержащие информацию о событиях должны наследовать от типа System.EventArgs, а имя типа должно заканчиваться словом EventArgs

+делегаты  
 1)Делегат можно вызвать асинхронно (в отдельном потоке), при этом в исходном потоке можно продолжать действия. 2)Анонимный делегат (без создания классанаблюдателя): s.Go += new SomeDelegat(()=> { Console.WriteLine("Я слышу что идете!"); });3)Делегаты и события обеспечивают взаимодействие взаимосвязанных объектов. 4)События включены во многие стандартные классы .NET, используемые для разработки Windows-приложений.

# **Стандартные коллекции .NET. Типы коллекций.**

**Типы коллекций:**

1. ***необобщенные***System.Collections*:* функциональные возможности описываются в интерфейсах(IEnumerable, IList, ICollection, к-е реализуются сл классами ArrayList, Stack, Queue) 1)наличие разнотипных данных 2)ссылки на данные типа object (не обеспечивают типовую безопасность)

2. ***обобщенные*** System.Collections.Generic: обеспечивают типовую безопасность и повышают производительность. 1)обеспечивают типовую безопасность 2)System.Collections.Generic

3. ***специальные*** System.Collections.Specialized: для работы с данными конкретного типа или для обработки их особым образом.

4. ***с поразрядной организацией*** BitArray: подразрядные операции(и, или, исключающее или)

5. ***параллельные***System.Collections.Generic: поддерживают многопоточный доступ к коллекции – одновременный доступ к коллекции со стороны 2х или больше || исполняемых потоков. 1)многопоточный доступ к коллекции

Каждый класс коллекции оптимизирован под конкретную форму хранения данных и доступа к ним, и каждый из них предоставляет специализированные методы.

*Интерфейсы, используемые в коллекциях*: 1)IEnumerable<T>(перечислитель, с помощью которого становится возможен последовательный перебор коллекции) 2)IEnumerator<> (позволяет перебирать элементы коллекции) 3)ICollection<T> 4)IList<T> 5)IComparer<T>(сравнение двух объектов) и др.

**Обобщенные коллекции**

►Dictionary <Tkey, TValue>

►LinkedList<T>

►List<T>

►Queue<T>

►SortedDictionary<Tkey, Tvalue>

►SortedList<T> (использовании памяти и в скорости вставки и удаления)

►HashSet<T> и SortedSet<T>

►Stack<T>

преимущества: повышение производительности (не надо тратить время на упаковку и распаковку объекта) и повышенная типобезопасность

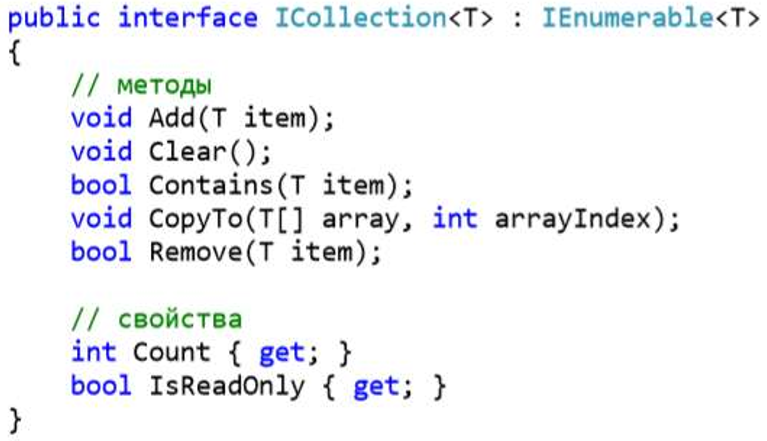
# **Стандартные интерфесы коллекций.**

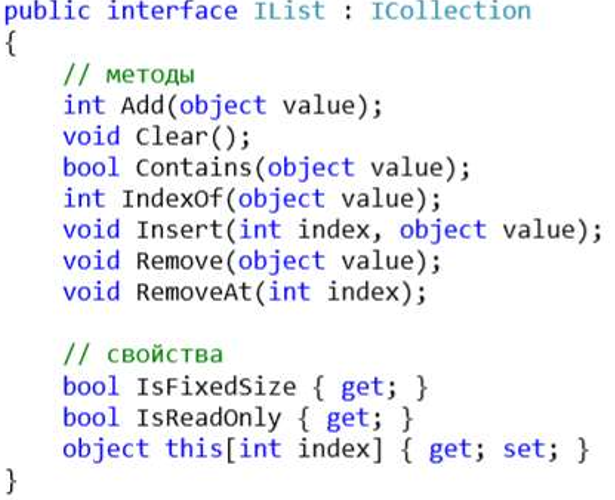
**Интерфейсы, используемые в коллекциях:**

**1) IEnumerable<T> (**перечислитель, с помощью которого становится возможен последовательный перебор коллекции**)** для foreach, GetEnumerator()

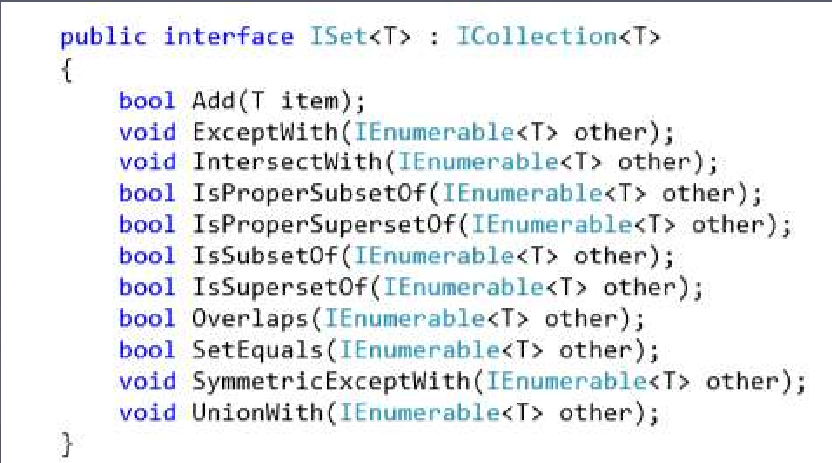
**2) IEnumerator<> (**позволяет перебирать элементы коллекции**)**

**3) ICollection<T> :** Count, СоруТо(),Add(), Remove(), Clear()

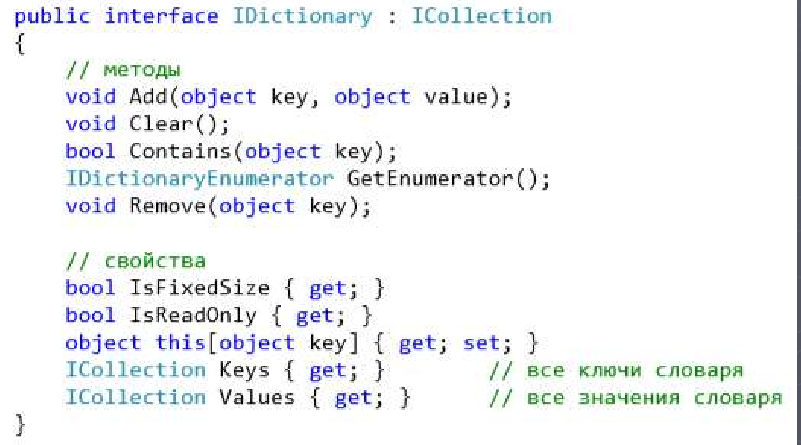
  
**4) IList (**описывает набор данных, которые проецируются на массив**)**

  
**5) IList<T> (**позволяет получать элементы коллекции по порядку**):** Индексатор, Insert(), Remove()

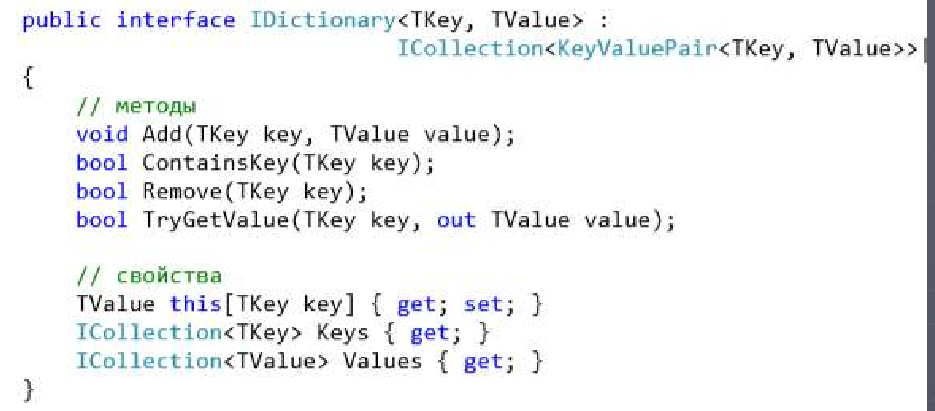
**6) ISet<T>**



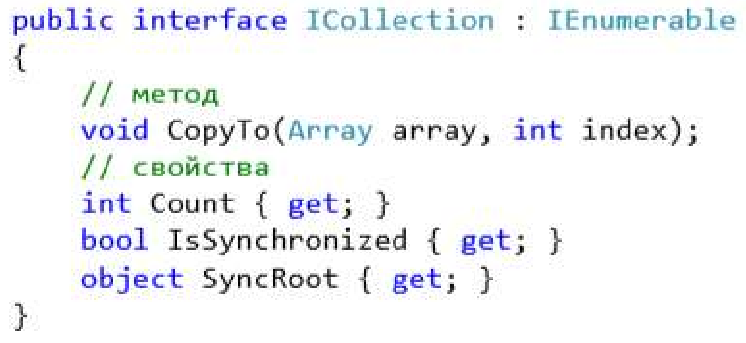
**7) IDictionary (**протокол взаимодействия для коллекций-словарей (KeyValuePair<TKey, TValue> – это вспомогательная структура, у которой определены свойства Key и Value)**)**



**8) IDictionary<TKey, TValue>**

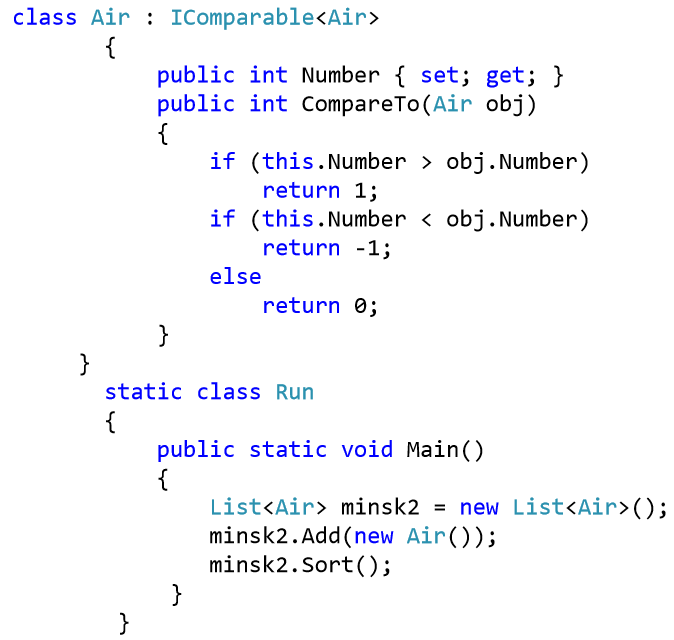


**9)ICollection:** определяет элементы

 **10) IComparer<T> (**сравнения двух объектов**)**

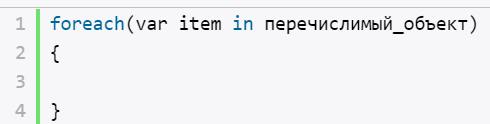
**11) IComparer:**  содержит и требует его реализации int Compare(object x, object y)

**12)** **IComparable:** Для сортировки и сравнения объектов (SortedList), Требует реализации **int CompareTo(object obj)**



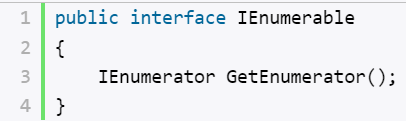
# **IEnumerable и IEnumerator**

необобщенный интерфейс IEnumerator или обобщенный интерфейс IEnumerator (Перечислители) 1)Реализация object Current { get; } 2)bool MoveNext() 3)void Reset() 4)Доступ только для чтения. Основной для большинства коллекций является реализация интерфейсов IEnumerable и IEnumerator. Благодаря такой реализации мы можем перебирать объекты в цикле foreach:

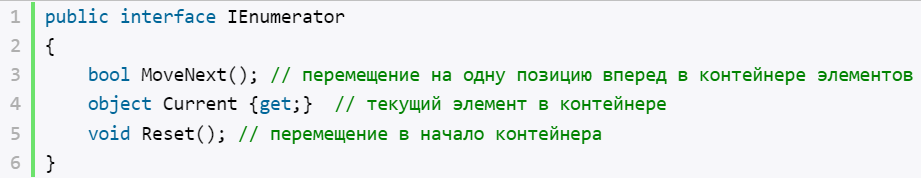


Перебираемая коллекция должна реализовать интерфейс IEnumerable.

Интерфейс IEnumerable имеет метод, возвращающий ссылку на другой интерфейс - перечислитель:



А интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере:



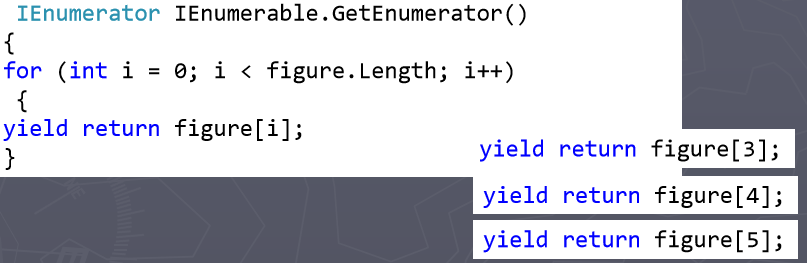
Метод **MoveNext()** перемещает указатель на текущий элемент на следующую позицию в последовательности. Если последовательность еще не закончилась, то возвращает true. Если же последовательность закончилась, то возвращается false.

Свойство **Curr**ent возвращает объект в последовательности, на который указывает указатель.

Метод **Reset()** сбрасывает указатель позиции в начальное положение.

**Итераторы** (метод, оператор или аксессор, возвращающий по очереди члены совокупности объектов и имеет оператор yield.)

При обращении к оператору yield return будет сохраняться текущее местоположение и при переходе к следующей итерации для получения нового объекта, итератор начнет выполнения с этого местоположения.



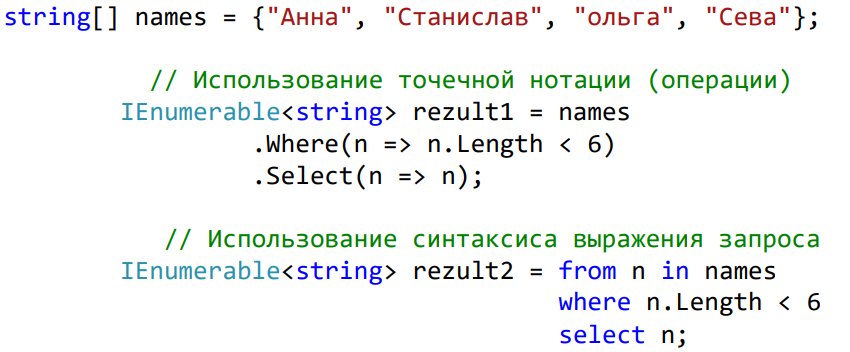
**Именованный итератор**

public IEnumerable имя\_итератора(список\_параметров) { // ... yield return obj; }

# **LINQ to Objects. Синтаксис. Форма. Возврат результата. Грамматика выражений запросов. Отложенные и неотлаженные операции.**

**LINQ** (Language-Integrated Query) - простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс IEnumerable (например, стандартные коллекции, массивы), набор данных DataSet и т.д. Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

***Синтаксис*** выражений запросов поддерживается : Where, Select, SelectMany, Join, GroupJoin, GroupBy, OrderBy, ThenBy, OrderByDescending и ThenByDescending



Стандартные операторы запросов — это методы, формирующие шаблон LINQ. Большинство из этих методов работают с последовательностями, где последовательность — это объект, тип которого реализует интерфейс [IEnumerable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1) или интерфейс [IQueryable<T>.](https://docs.microsoft.com/en-us/dotnet/api/system.linq.iqueryable-1) Стандартные операторы запросов предоставляют возможности запросов, включая фильтрацию, проецирование, агрегирование, сортировку и многое другое.

Существует два набора стандартных операторов запросов LINQ: один, который работает с объектами типа [IEnumerable<T>,](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1)другой, который работает с объектами типа [IQueryable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.linq.iqueryable-1).

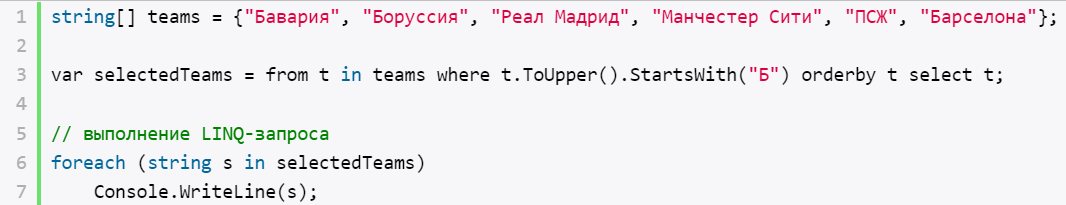
***Операторы***: Агрегация (Count, Min, Max) Преобразование (Cast, ofType, ToArray, ToList, ToDictionary) Конкатенация (Concat) Элемент (Last, First, Single , ElemetAt+ Default) Множество (Except, Distinct, Union) Генерация (Empty, Range, Repeat) Соединение (Join, GroupJoin) Упорядочивание (OrderBy, ThenBy, Reverse,….) Проекция (Select, SelectMany) Разбиение (Skip, Take , +While) Ограничение (Where) Квантификатор (Any, All, Contains) Эквивалентность(SequeceEqual)

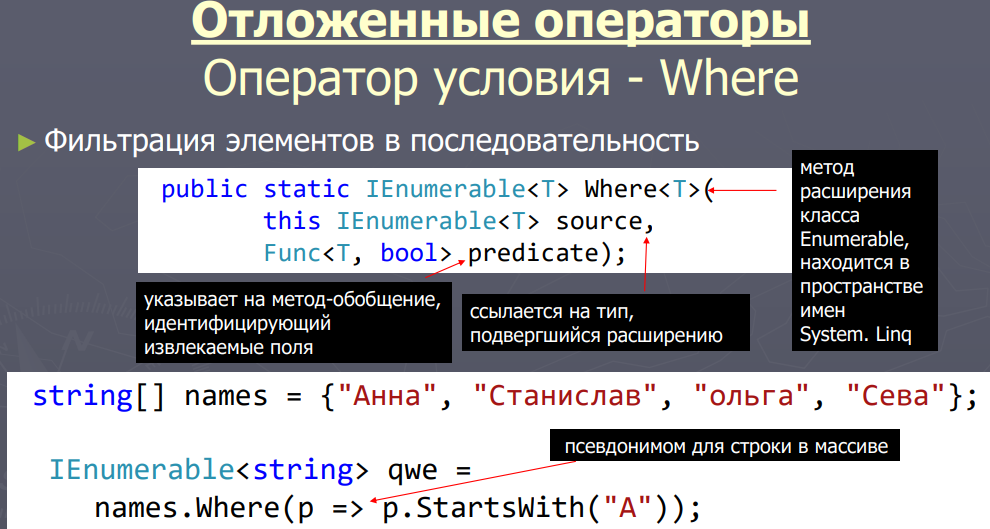
Грамматика выражений запросов 1) Начало - from 2) [ from, let или where] 3) [orderby, ascending или descending ] 4) [select или group] 5) [конструкции into, join, или повторение с п.2. ] Выражение в методы расширения

Есть два способа выполнения запроса LINQ: отложенное и немедленное выполнение

При отложенном выполнении LINQ-выражение не выполняется, пока не будет произведена итерация или перебор по выборке.

***Отложенное выполнение***





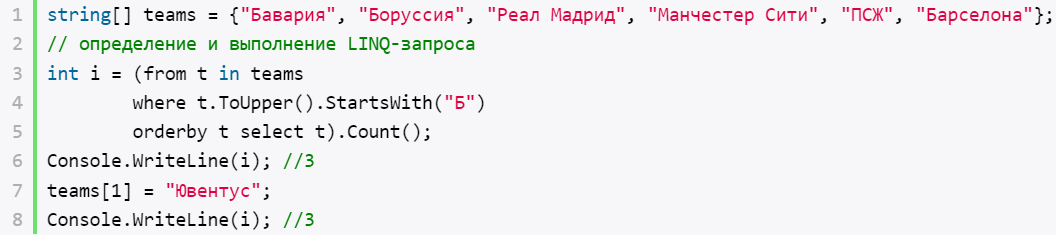
Фактически LINQ-запрос разбивается на три этапа:

Получение источника данных

Создание запроса

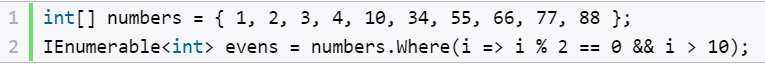
Выполнение запроса и получение его результатов

С помощью ряда методов мы можем применить немедленное выполнение запроса. Это методы, которые возвращают одно атомарное значение или один элемент. Например, Count(), Average(), First() / FirstOrDefault(), Min(), Max() и т.д. Например, метод Count() возвращает числовое значение, которое представляет количество элементов в полученной последовательности. А метод First() возвращает первый элемент последовательности. Но чтобы выполнить эти методы, вначале надо получить саму последовательность, то есть результат запроса, и пройтись по ней циклом foreach, который вызывается неявно внутри структуры запроса.



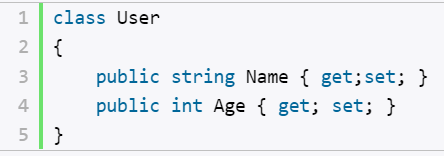
# **LINQ to Objects. Операции Where, Select, Take, OrderB, Join, GroupBy**

Операция **Where** используется для фильтрации элементов в последовательность. Например, выберем все четные элементы, которые больше 10.

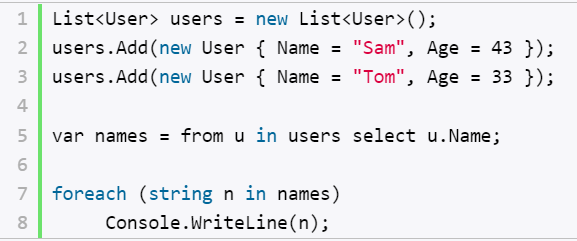


Этот прототип **Where** принимает входную последовательность и делегат метода-предиката, а возвращает объект, который при перечислении проходит по входной последовательности, выдавая элементы, для которых делегат метода-предиката возвращает true.

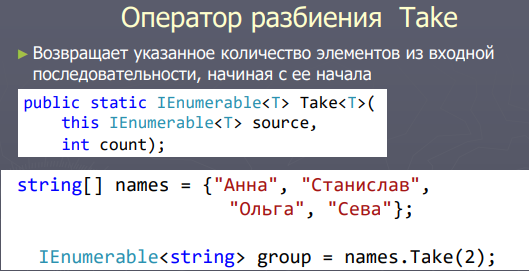
Проекция позволяет спроектировать из текущего типа выборки какой-то другой тип. Для проекции используется оператор **select**. Допустим, у нас есть набор объектов следующего класса, представляющего пользователя:



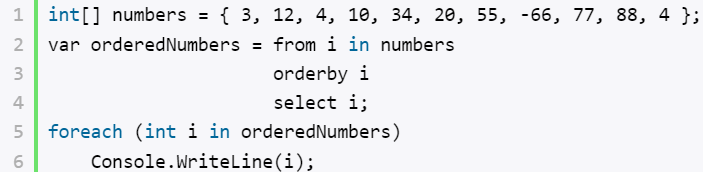
Но нам нужен не весь объект, а только его свойство Name:

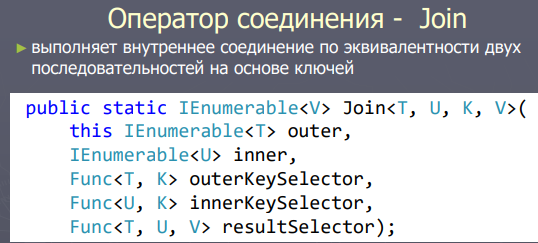


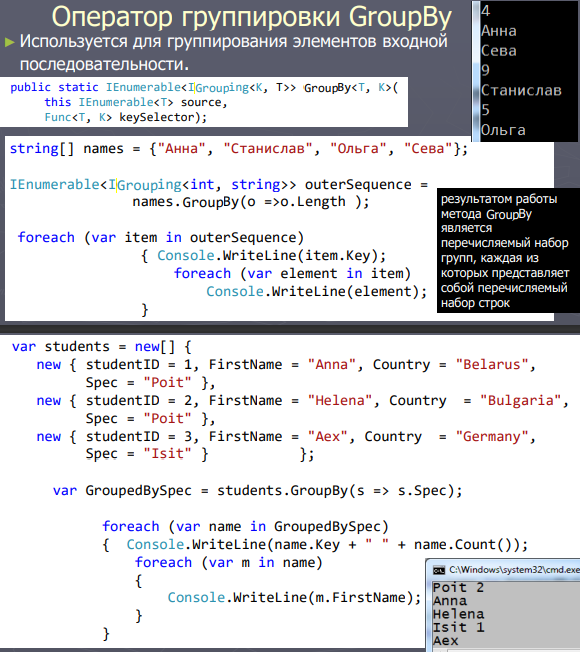
Результат выражения LINQ будет представлять набор строк, поскольку выражение select u.Name выбирают в результирующую выборку только значения свойства Name.



Для сортировки набора данных по возрастанию используется оператор **orderby**:







# **Рефлексия. System Type.**

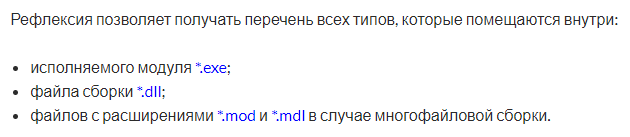
**Рефлексия представляет собой процесс выявления типов во время выполнения приложения.** Каждое приложение содержит набор используемых классов, интерфейсов, а также их методов, свойств и прочих кирпичиков, из которых складывается приложение. **И рефлексия как раз и позволяет определить все эти составные элементы приложения.** Основной функционал рефлексии сосредоточен в пространстве имен **System.Reflection.**

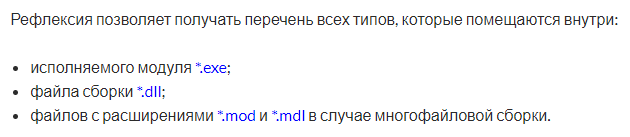
Основной функционал рефлексии сосредоточен в пространстве имен **System.Reflection**. В нем мы можем выделить следующие основные классы:

* **Assembly**: класс, представляющий **сборку** и позволяющий манипулировать этой сборкой (например, чтобы загрузить сборку, есть метод LoadFrom и Load, первый принимает путь к сборке)
* **AssemblyName**: класс, хранящий **информацию о сборке**
* **MemberInfo**: базовый абстрактный класс, определяющий общий функционал для классов EventInfo, FieldInfo, MethodInfo и PropertyInfo
* **EventInfo**: класс, хранящий **информацию о событии**
* **FieldInfo**: хранит информацию **об определенном поле типа**
* **MethodInfo**: хранит **информацию об определенном методе**
* **PropertyInfo**: хранит **информацию о свойстве**
* **ConstructorInfo**: класс, представляющий конструктор
* **Module**: класс, позволяющий получить доступ к определенному модулю внутри сборки
* **ParameterInfo**: класс, хранящий информацию о параметре метода

Эти классы **представляют составные блоки типа и приложения: методы, свойства и т.д.** Но чтобы получить **информацию о членах типа**, нам надо воспользоваться классом **System.Type.**

**Рефлексия (reflection) в .NET – это процесс выявления метаданных (типов) во время выполнения программы в виде объектной модели**. **Другими словами, рефлексия – это средство для получения сведений о типе данных.**





Рефлексия (отражение) — это процесс, во время которого программа может отслеживать и модифицировать собственную структуру и поведение во время выполнения, **это своего рода процесс обнаружения типов во время выполнения.**

**Преимущества рефлексии:**

1. Можно создавать типы и вызывать их методы без предыдущих знаний об именах, которые помещаются в этих типах. Это динамическая идентификация типов
2. Не нужно на этапе компиляции знать информацию о типе, из которого будут вытягиваться метаданные. Достаточно только название типа
3. Представление типов, которые вытягиваются из сборки, в виде удобной объектной модели

**Класс System.Type представляет изучаемый тип,** инкапсулируя всю информацию о нем. **С помощью его свойств и методов можно получить эту информацию.** Некоторые из его свойств и методов:

* Метод FindMembers() возвращает массив объектов MemberInfo данного типа
* Метод **GetConstructors()** возвращает все конструкторы данного типа в виде набора объектов ConstructorInfo
* Метод **GetEvents()**возвращает все события данного типа в виде массива объектов EventInfo
* Метод **GetFields()** возвращает все поля данного типа в виде массива объектов FieldInfo
* Метод **GetInterfaces()** получает все реализуемые данным типом интерфейсы в виде массива объектов Type
* Метод **GetMembers()** возвращает все члены типа в виде массива объектов MemberInfo
* Метод **GetMethods()** получает все методы типа в виде массива объектов MethodInfo
* Метод **GetProperties()** получает все свойства в виде массива объектов PropertyInfo
* Свойство **Name** возвращает имя типа
* Свойство **Assembly** возвращает название сборки, где определен тип
* Свойство **Namespace** возвращает название пространства имен, где определен тип
* Свойство **IsArray** возвращает true, если тип является массивом
* Свойство **IsClass** возвращает true, если тип представляет класс
* Свойство **IsEnum** возвращает true, если тип является перечислением
* Свойство **IsInterface** возвращает true, если тип представляет интерфейс

**Чтобы управлять типом и получать всю информацию о нем, нам надо сперва получить данный тип**. Это можно сделать тремя способами: с помощью ключевого слова **typeof**, с помощью метода **GetType()** класса **Object** и применяя **статический метод Type.GetType().**

Например, определен класс User с некоторым набором свойств и полей. И чтобы получить его тип, используется выражение Type myType = typeof(User);

Получение типа с помощью метода GetType класса Object:

|  |  |
| --- | --- |
|  | User user = new User("Tom", 30);  Type myType = user.GetType(); |

В отличие от предыдущего примера здесь, чтобы получить тип, надо создавать объект класса. И третий способ получения типа - статический метод Type.GetType():

|  |  |
| --- | --- |
|  | Type myType = Type.GetType("TestConsole.User", false, true); |

Первый параметр указывает на полное имя класса с пространством имен. Второй параметр указывает, будет ли генерироваться исключение, если класс не удастся найти. И третий параметр указывает, надо ли учитывать регистр символов в первом параметре.

1. **Поздним связыванием (late binding)** называется **технология, которая позволяет создавать экземпляр определенного типа и вызывать его члены во время выполнения** без кодирования факта его существования жестким образом на этапе компиляции.
2. **При раннем связывании компилятор заранее знает адрес с типом, с которым будет оперировать**. При позднем связывании компилятор узнаёт этот адрес в процессе выполнения программы.

**Перечисление BindingFlags может принимать различные значения:**

1. **DeclaredOnly**: **получает только методы непосредственно данного класса**, унаследованные методы не извлекаются
2. **Instance**: получает только методы экземпляра
3. **NonPublic**: извлекает не публичные методы (указывает, что приватные члены нужно включить в поиск)
4. **Public**: **получает только публичные методы**
5. **Static**: **получает только статические методы**

# **Классы для работы с файловой системой.**

**Пространство имен System.IO в .NET — это область библиотек базовых классов, посвященная службам файлового ввода-вывода, а также ввода-вывода из памяти.** Фреймворк .NET предоставляет большие возможности по управлению и манипуляции файлами и каталогами, которые по большей части сосредоточены в пространстве имен System.IO. **Классы, расположенные в этом пространстве имен (такие как Stream, StreamWriter, FileStream и др.), позволяют управлять файловым вводом-выводом.**

Многие типы из пространства имен System.IO сосредоточены на программных манипуляциях физическими каталогами и файлами. Дополнительные типы предоставляют поддержку чтения и записи данных в строковые буферы, а также области памяти.

**BinaryReader, BinaryWriter** - **Эти классы позволяют сохранять и извлекать элементарные типы данных** (целочисленные, булевские, строковые и т.п.) **в двоичном виде**

**BufferedStream** - Этот класс предоставляет временное хранилище для потока байтов, которые могут затем быть перенесены в постоянные хранилища

**Directory, DirectoryInfo - Эти классы используются для манипуляций структурой каталогов машины**. Тип Directory представляет функциональность, используя статические члены. Тип DirectoryInfo обеспечивает аналогичную функциональность через действительную объектную ссылку

**DriveInfo** - **Этот класс предоставляет детальную информацию относительно дисковых устройств**, используемых данной машиной

**File, FileInfo** - **Эти классы служат для манипуляций множеством файлов данной машины**. Тип File представляет функциональность через статические члены. Тип FileInfo обеспечивает аналогичную функциональность через действительную объектную ссылку

**FileStream** - Этот класс обеспечивает произвольный доступ к файлу (т.е. возможности поиска) с данными, представленными в виде потока байт

**FileSystemWatcher** - Этот класс позволяет отслеживать модификации внешних файлов в определенном каталоге

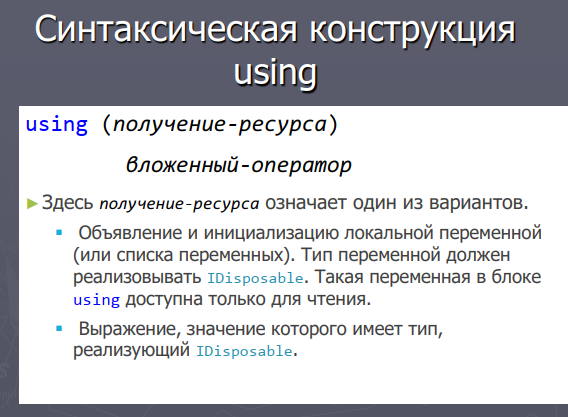
**MemoryStream** - Этот класс обеспечивает произвольный доступ к данным, хранящимся в памяти, а не в физическом файле

**Path** - Этот класс выполняет операции над типами System.String, содержащими информацию о пути к файлу или каталогу в независимой от платформы манере

**StreamWriter, StreamReader** - Эти классы используются для хранения (и извлечения) текстовой информации из файла. Эти классы не поддерживают произвольного доступа к файлу

# **Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы.**

Синтаксическая конструкция using



Синтаксис C# предлагает синонимичную конструкцию для автоматического вызова метод Dispose - конструкцию using:

Using (Person p= new Person()) {}

Конструкция using оформляет блок кода и создает объект некоторого класса, который реализует интерфейс IDisposable, в частности, его метод Dispose. При завершении блока кода у объекта вызывается метод Dispose. **Важно, что данная конструкция применяется только для классов, которые реализуют интерфейс IDisposable.**

Для работы с бинарными файлами предназначена пара классов **BinaryWriter и BinaryReader**. Эти классы позволяют читать и записывать данные в двоичном формате. Класс FileStream не очень удобно применять для работы с текстовыми файлами. К тому же для этого в пространстве System.IO определены специальные классы: **StreamReader и StreamWriter.**

**Для записи в текстовый файл используется класс StreamWriter.** **StreamWriter(string path):** через параметр path передается путь к файлу, который будет связан с потоком

**StreamWriter(string path, bool append): параметр append указывает, надо ли добавлять в конец файла данные или же перезаписывать файл**. Если равно true, то новые данные добавляются в конец файла. Если равно false, то файл перезаписываетсяя заново

**int Close(): закрывает записываемый файл** и освобождает все ресурсы

void Flush(): записывает в файл оставшиеся в буфере данные и очищает буфер.

**void Write(string value): записывает в файл данные простейших типов, как int, double, char, string и т.д.**

**void WriteLine(string value): также записывает данные, только после записи добавляет в файл символ окончания строки**

**Класс StreamReader позволяет нам легко считывать весь текст или отдельные строки из текстового файла.**

**StreamReader(string path): через параметр path передается путь к считываемому файлу**

**StreamReader(string path, System.Text.Encoding encoding):** параметр encoding задает кодировку для чтения файла

**void Close(): закрывает считываемый файл** и освобождает все ресурсы

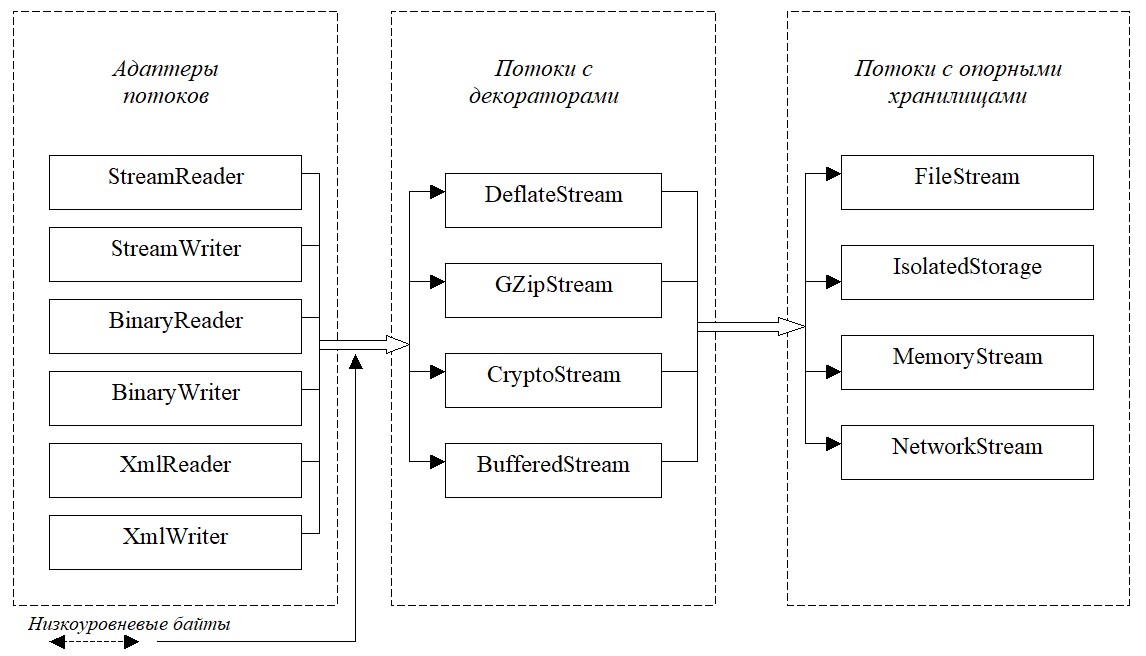
**int Peek(): возвращает следующий доступный символ**, если символов больше нет, то возвращает -1

**int Read(): считывает и возвращает следующий символ в численном представлении**.

**string ReadLine(): считывает одну строку в файле**

# **Классы адаптеры потоков.**

Поток представляет собой абстракцию, которая обеспечивает ввод/вывод информации в программе. Система ввода/вывода связывает поток с физическим устройством. Работа потока на ввод или на вывод содержит одинаковый набор команд независимо от физического устройства.



Адаптеры потоков относятся к более высокому уровню взаимодействия с программой. Они позволяют конвертировать байтовые потоки (потоки с декораторами, потоки с опорными хранилищами) в конкретный формат.

Адаптеры потоков работают по единому принципу: они помещают байтовый поток в оболочку адаптерного класса с соответствующими методами.

* **TextReader/ TextWriter** — абстрактный класс, который может читать/ записывать последовательности символов;
* **StreamReader** — класс, реализующий TextReader;
* **StreamWriter** — класс, который реализует абстрактный класс TextWriter. Класс содержит средства для записи символов в поток в заданной кодировке;
* **BinaryReader** — класс, содержащий методы чтения примитивных типов данных (int, float, double и т.п.) в указанной системе кодировки;
* **BinaryWriter** — класс, реализующий методы записи примитивных типов данных (int, float, byte и т.д.) и строк в указанной системе кодировки;
* **XmlReader** — класс, содержащий средства некешированного считывания XML-данных;
* **XmlWriter** — класс, содержащий средства записи потоков или файлов с XML-данными.

# **Сериализация. Форматы сериализации.**

**Термин сериализация описывает процесс сохранения (и, возможно, передачи) состояния объекта в потоке (т.е. файловом потоке и потоке в памяти).** **Последовательность сохраняемых данных содержит всю информацию, необходимую для реконструкции (или десериализации) состояния объекта с целью последующего использования**. Применяя эту технологию, очень просто сохранять большие объемы данных (в различных форматах) с минимальными усилиями. **Сериализация (serialization) - это операция преобразования объекта в формат, пригодный для записи на диск или отправки в сеть. Десериализация (deserialization) - это операция восстановления объекта из сериализованного представления. CЕРИАЛИЗАЦИЯ – ПРОЦЕСС ПРЕОБРАЗОВАНИЯ ОБЪЕКТОВ В ПОТОК БАЙТ. ДЕСЕРИАЛИЗАЦИЯ – ПОЛУЧЕНИЕ ИЗ ПОТОКА БАЙТ СОХРАНЕННОГО ОБЪЕКТА.**

**Форматы:**

1. Бинарный
2. SOAP
3. Xml
4. JSON

Пространство имен: System.Runtime.Serialization

**Сериализуемый тип – тип, помеченный атрибутом [Serializable], у которого все поля имеют сериализуемый тип.** Базовые типы платформы .NET являются сериализуемыми.

**БИНАРНАЯ СЕРИАЛИЗАЦИЯ.**

Класс BinaryFormatter определен в пространстве имен System.Runtime.Serialization.Formatters.Binary. Ключевыми методами типа BinaryFormatter являются **Serialize() и Deserialize():**

**Serialize() сохраняет граф объектов в указанный поток в виде последовательности байтов;** ПЕРВЫЙ ПАРАМЕТР – ПОТОК, ВТОРОЙ – ОБЪЕКТ. СЕРИАЛИЗУЕТСЯ В ЗАКРЫТЫЙ ДВОИЧНЫЙ ФОРМАТ.

**Deserialize() преобразует сохраненную последовательность байт в граф объектов**. ПАРАМЕТР - ПОТОК

**СЕРИАЛИЗАЦИЯ В JSON.**

JSON — это формат, который хранит структурированную информацию и в основном используется для передачи данных между сервером и клиентом.

Основная функциональность по работе с JSON сосредоточена в пространстве имен **System.Text.Json. Ключевым типом является класс JsonSerializer**, который и позволяет сериализовать объект в json и, наоборот, десериализовать код json в объект C#.

Serialize (ОБЪЕКТ obj, ТИП, ОПЦИИ);

Deserialize (СТРОКА json, ТИП, ОПЦИИ); - требует указания типа

**ВАЖНО: Объект, который подвергается десериализации, должен иметь конструктор без параметров. Сериализации подлежат только публичные свойства объекта (с модификатором public).** Атрибут [JsonIgnore] позволяет исключить из сериализации определенное свойство.

**XML-СЕРИАЛИЗАЦИЯ.**

**Класс должен иметь конструктор без параметров. Сериализации подлежат только открытые члены. Приватные данные игнорируются. Требует указания типа.**

Атрибуты, которые влияют на кодирование данных XML в потоке:

[XmlAttribute] Поле или свойство будет сериализовано как атрибут XML (а не как подэлемент)

[XmlElement] Поле или свойство будет сериализовано как элемент XML с указанным именем

[XmlEnum] Этот атрибут предоставляет имя элемента, являющееся членом перечисления

[XmlRoot] Этот атрибут управляет тем, как будет сконструирован корневой элемент (пространство имен и название элемента)

[XmlText] Свойство или поле должно быть сериализовано как текст XML (т.е. содержимое, находящееся между начальным и конечным дескрипторами корневого элемента)

[XmlType] Этот атрибут предоставляет имя и пространство имен типа XML

**SOAP-СЕРИАЛИЗАЦИЯ.**

**Протокол SOAP (Simple Object Access Protocol) представляет простой протокол для обмена данными между различными платформами**. При такой сериализации данные упакуются в конверт SOAP, данные в котором имеют вид xml-подобного документа. **Надо добавить в проект сборку System.Runtime.Serialization.Formatters.Soap.dll.**

Принцип использования SoapFormatter похож на рассмотренную бинарную сериализацию. **Здесь также создается поток, записывающий данные в файл people.soap. Для сериализации используется метод formatter.Serialize(fs, people), использующий поток и объект для сериализации.** При десериализации считываем ранее сохраненные объекты и преобразуем их к **нужному нам объекту в методе Deserialize.**

**В отличие от BinaryFormatter, платформа и операционная система не влияют на успешное восстановление данных, сериализированных с помощью SoapFormatter**. **Как и в случае с BinaryFormatter (де)сериализация происходит с помощью ключевых методов Serialize() и Deserialize() – они вместе реализуют интерфейс IFormatter. Сериализуемый класс должен содержать конструктор без параметров, который вызывается при создании нового объекта в ходе десериализации.**

# **Сериализация контрактов данных. интерфейс ISerializable.**

**Контракт данных – это тип (класс или структура), объект которого описывает информационный фрагмент (открытые поля и свойства) – один из механизмов сериализации.** Основным форматом хранения контрактов данных является XML. **Контракт данных - формальное соглашение между службой и клиентом, абстрактно описывающее данные, обмен которыми происходит.** Это значит, что для взаимодействия клиент и служба не обязаны совместно использовать одни и те же типы, достаточно совместно использовать одни и те же контракты данных.

Данные контракта могут быть явными или неявными. Простой тип, такой как int, string и т. Д., ммеет неявный контракт данных. Пользовательский объект является явным или сложным типом, для которого вы должны определить контракт данных, используя атрибуты [DataContract] и [DataMember].

**[DataContract] Контракт данных: он указывает, что ваш класс сущностей готов к процессу сериализации. [DataMember] Элементы данных. Указывает, что конкретное поле является частью контракта данных и может быть сериализовано.**

Использование сериализатора контрактов данных предполагает следующие три шага

* Выбрать класс: DataContractSerializerили NetDataContractSerializer
* Добавить сериализуемым АТРИБУТЫ [DataContract]и [DataMember] (соответственно)
* СОЗДАТЬ ЭКЗЕМПЛЯР СЕРИАЛИЗАТОРА
* ВЫЗВАТЬ МЕТОДЫ WriteObject()/ReadObject()

**DataContractSerializer — обеспечивает слабую привязку типов .NET к типам контрактов данных**. Может генерировать совместимый со стандартами XML код. Требует предварительной явной регистрации сериализуемых производных типов, чтобы иметь возможность сопоставлять имена контрактов данных с именами типов .NET

**NetDataContractSerializer — характеризуется тесной привязкой типов .NET к типам контрактов данных, не требует явной регистрации сериализуемых производных типов,** т.к. самостоятельно записывает полные имена типов и сборок сериализуемых типов

Чтобы сериализация работала корректно:

1) У класса должен быть дефолтный конструктор без параметров.

2) Этот класс должен быть помечен атрибутом "Serializable".

**3) Если мы хотим сами решать, что сериализовать, а что нет — реализовываем интерфейс "ISerializable". В этом интерфейсе определен метод "GetObjectData()".  Это метод, который вызывается при сериализации. Внутри этого метода и определяется какие поля и куда добавлять. Этот интерфейс позволяет объекту управлять собственной сериализацией и десериализацией.**

# **Атрибуты. Создание собственного атрибута.**

**[OnSerializing] – атрибут указывает, какой метод вызывается перед процессом сериализации**

**[OnSerialized] – атрибут указывает метод, который будет вызван немедленно после того**, как объект будет сериализован

**[OnDeserializing] – атрибут указывает метод, который будет вызван перед процессом десериализации**

**[OnDeserialized] – атрибут указывает метод, который будет вызван немедленно после десереиализации объекта**

**Использование этих атрибутов дает менее громоздкий код, чем реализация интерфейса ISerializable. Вместо этого можно напрямую модифицировать данные состояния, когда форматер работает с вашим типом**. **Эти атрибуты определены в System.Runtime.Serialization.**

**Атрибуты** в .NET представляют специальные инструменты, которые позволяют **встраивать в сборку дополнительные метаданные**. Атрибуты могут применяться как ко всему типу (классу, интерфейсу и т.д.), так и к отдельным его частям (методу, свойству и т.д.). **Основу атрибутов составляет класс System.Attribute**.

Первым шагом в процессе создания специального атрибута является создание **нового класса, унаследованного от System.Attribute**. **В классе атрибута определяются члены, поддерживающие атрибут. Классы атрибутов зачастую оказываются довольно простыми и содержат небольшое количество полей или свойств**.

// Создаем атрибут

public sealed class UInfoAttribute : System.Attribute

{

public string Desc;

public UInfoAttribute() { }

public UInfoAttribute(string str)

{

Desc = str;

}

}

**Как только класс атрибута будет определен, атрибут можно присоединить к элементу. Атрибут указывается перед тем элементом, к которому он присоединяется, и для этого его конструктор заключается в квадратные скобки. По умолчанию действие специальных атрибутов может распространяться на практически любой аспект кода (методы, классы, свойства и т.д.). Чтобы ограничить область действия специального атрибута, необходимо применить к его определению**атрибут **[AttributeUsage].**

# **Процесс. Домен приложений. Поток выполнения.**

**При запуске приложения операционная система создает для него отдельный процесс, которому выделяется определённое адресное пространство в памяти и который изолирован от других процессов**. **Процесс может иметь несколько потоков**. **Как минимум, процесс содержит один - главный поток. В приложении на C# точкой входа в программу является метод Main.** Вызов этого метода автоматически создает главный поток. А из главного потока могут запускаться вторичные потоки. Формально поток, который во входной точке процесса создается первым, называется главным потоком (primary thread).

**Для каждого загружаемого в память файла \*.exe в ОС создается отдельный изолированный процесс, который используется на протяжении всего времени его существования.** Выход из строя одного процесса не сказывается на работе других процессов. **Каждый процесс Windows получает PID, уникальный идентификатор процесса.**

**Поток – это некая независимая последовательность инструкций для выполнения того или иного действия в программе**. В одном конкретном потоке выполняется одна конкретная последовательность действий.  
**Совокупность таких потоков, выполняемых в программе параллельно называется многопоточностью программы.**

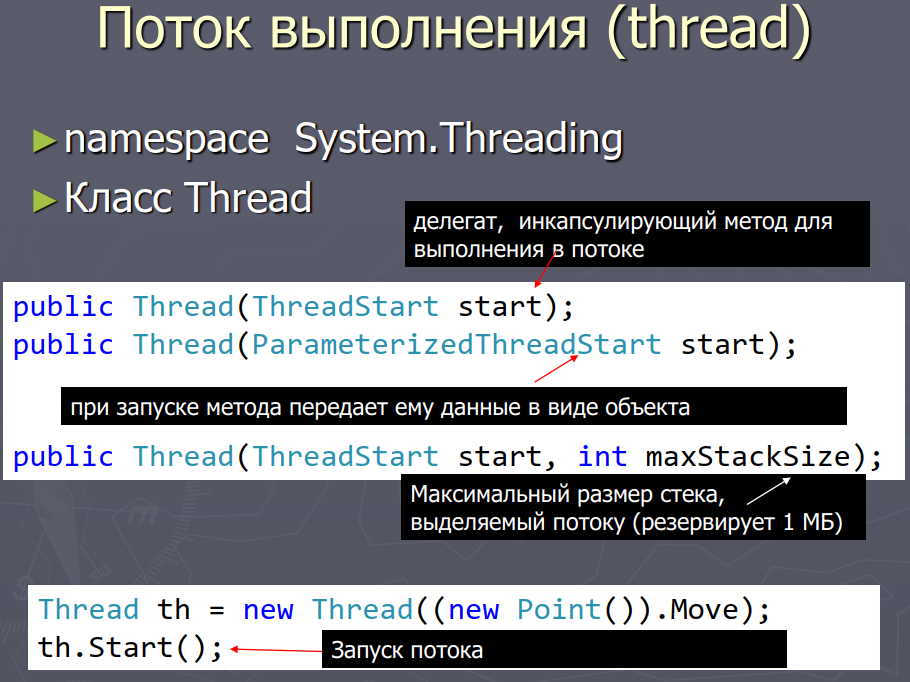
**При запуске приложения, написанного на C#, операционная система создает процесс**, а **среда CLR создает внутри этого процесса логический контейнер, который называется доменом приложения и внутри которого работает запущенное приложение (класс AppDomain). Исполняемые файлы обслуживаются внутри ДОМЕНА ПРИЛОЖЕНИЯ, в отдельном логическом разделе внутри процесса. В одном процессе может содержаться несколько таких доменов (класс System.AppDomain)**

О домене приложения:

1. Существует внутри процессов
2. Содержат загруженные сборки
3. Процесс запускает при старте домен по умолчанию AppDomain.CurrentDomain
4. Домены могут создаваться и уничтожаться в ходе работы в рамках процесса, создаем экземпляр AppDomain и используем его метод Load(). Выгружать сборку из домена отдельно нельзя, можно выгружать сразу весь домен
5. Обеспечивают изоляцию кода

**Поток – используемый внутри процесса путь выполнения. CLR поддерживает многопоточность, опираясь на многопоточность ОС.** В каждом процессе содержится первоначальный поток – входная точка приложения, метод Main(). **Главный поток создается автоматически, а процессы, в которых содержится единственный главный поток, называются безопасными. Поток представлен классом Thread в пространстве имен System.Threading**

# **Создание потоков , классы приоритетов. Состояния потоков**



Статусы(состояния) 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

**CLR делит потоки: фоновые и остальные**. Процесс не может завершиться, пока не завершены его основные потоки.

**Чтобы поток не забывал, на чем он работал перед тем, как  
его выполнение было приостановлено, каждому потоку  
предоставляется возможность записывать данные в локальное хранилище потоков (Thread Local Storage — TLS) и выделяется отдельный стек вызовов.**

**Приоритеты потоков, определены в свойстве Priority**

* **Lowest**
* **BelowNormal**
* **Normal (по умолчанию)**
* **AboveNormal**
* **Highest**

**Методы класса Thread:**

**GetDomain** - статический, возвращает ссылку домен приложения

**GetDomainId** - статический, возвращает id домена приложения, в котором выполняется текущий поток

**Sleep** – статический, останавливает поток на определенное количество миллисекунд

**Abort** - уведомляет среду CLR о том, что надо прекратить поток (происходит не сразу)

**Interrupt** - прерывает поток на некоторое время

**Join** - блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод

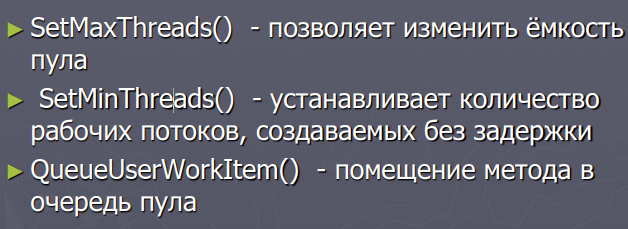
**Resume** - возобновляет работу приостановленного потока

**Start** - запускает поток

**Suspend** - приостанавливает поток

**Yield** - передаёт управление следующему ожидающему потоку системы

**Пул потоков – для уменьшения издержек, связанных с созданием потоков, платформа .NET поддерживает специальный механизм, называемый пул потоков. Пул состоит из очереди методов и рабочих потоков. Емкость пула потоков – максимальное число рабочих потоков.**

Создавать подобный список потоков самостоятельно не понадобится. Для управления таким списком предусмотрен класс **ThreadPool**, который по мере необходимости уменьшает и увеличивает количество потоков в пуле до максимально допустимого. Значение максимально допустимого количества потоков в пуле может изменяться.

# **Синхронизация потоков. Lock. Monitor. Мutex. Semaphore**

Мьютекс. System.Threading.Mutex

Mutex mutex = new Mutex(); //создание объекта мьютекса

Позволяет организовывать критическую секцию для нескольких процессов. Критическая секция – участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным, устройству), который не должен быть использован одновременно более чем одним потоком исполнения.

Методы Mutex – это WaitOne() (вход в критическую секцию, приостанавливает выполнение потока до тех пор, пока не будет получен мьютекс) и ReleaseMutex() – для выхода из нее, выход может быть произведен только в том же потоке выполнения, что и вход.

Семафор. System.Threading.Semaphore(между процессами) и SemaphoreSlim(один процесс)

Объект синхронизации, позволяющий войти в заданный участок кода не более чем N потокам (N – емкость семафора). Получение и снятие блокировки в случае семафора может выполняться в разных потоках.

Методы: Wait() – получение блокировки, Release() – снятие блокировки

Для создания семафора используется класс Semaphore: static Semaphore sem = new Semaphore(3, 3);. 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.

Lock.

**Средство блокировки встроено в язык C#. Благодаря этому все объекты могут быть синхронизированы. Синхронизация организуется с помощью ключевого слова lock.** Она была предусмотрена в C# с самого начала.

**lock(lockObj) { // синхронизируемые операторы }**

**где lockObj обозначает ссылку на синхронизируемый объект.** **Если же требуется синхронизировать только один оператор, то фигурные скобки не нужны.** **Оператор lock гарантирует, что фрагмент кода, защищенный блокировкой для данного объекта, будет использоваться только в потоке, получающем эту блокировку**. **А все остальные потоки блокируются до тех пор, пока блокировка не будет снята**. **Блокировка снимается по завершении защищаемого ею фрагмента кода.**

**Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока.**

Монитор.

Наряду с оператором lock для синхронизации потоков мы можем использовать мониторы, представленные классом System.Threading.Monitor. Метод 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 метода Monitor.Exit происходит освобождение объекта locker, если блокировка осуществлена успешно, и он становится доступным для других потоков.

Кроме блокировки и разблокировки объекта класс Monitor имеет ряд методов, которые позволяют управлять синхронизацией потоков. Так, метод Monitor.Wait освобождает блокировку объекта и переводит поток в очередь ожидания объекта. А все потоки, которые вызвали метод Wait, остаются в очереди ожидания, пока не получат сигнала от метода Monitor.Pulse или Monitor.PulseAll.

# **Библиотека параллельных задач TPL. Класс Task. Состояние задачи.**

**В эпоху многоядерных машин, которые позволяют параллельно выполнять сразу несколько процессов, стандартных средств работы с потоками в .NET уже оказалось недостаточно**. Поэтому во фреймворк .NET была добавлена **библиотека параллельных задач TPL (Task Parallel Library),** В данной библиотеке находятся классы, которые позволяют запускать задачи параллельно. Ее **основной функционал - в пространстве имен System.Threading.Tasks.** **Данная библиотека позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах, если на целевом компьютере имеется несколько ядер**. Кроме того, упрощается сама работа по созданию новых потоков.

**Parallel Framework – набор типов и технологий, являющийся частью платформы .NET, предназначен для повышения производительности за счет средств, упрощающих добавление параллелизма в приложения: параллелизм на уровне задач** **(библиотека TPL, System.Threading.Tasks и System.Threading),** параллелизм при императивной обработке данных и параллелизм при декларативной обработке данных.

Вероятно, самым главным среди новых средств, внедренных в версию 4.0 среды .NET Framework, является библиотека распараллеливания задач (TPL). **Эта библиотека усовершенствует многопоточное программирование двумя основными способами**.

**Во-первых, она упрощает создание и применение многих потоков. И во-вторых, она позволяет автоматически использовать несколько процессоров.**

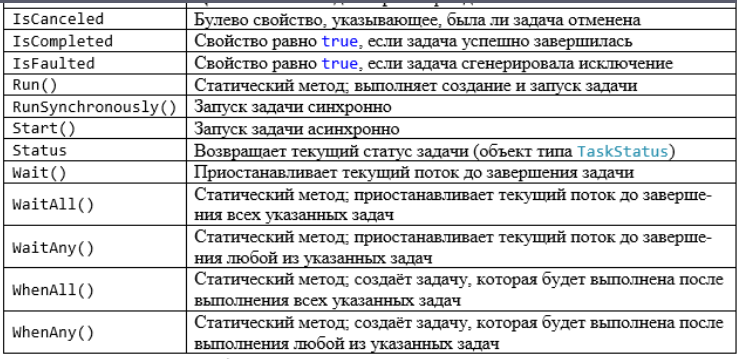
**В основе библиотеки TPL лежит КОНЕЦПИЯ ЗАДАЧ, каждая из которых описывает отдельную продолжительную операцию**. В библиотеке классов .NET задача представлена специальным **классом - классом Task, который находится в пространстве имен System.Threading.Tasks. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. ЗАДАЧА – АБСТРАКЦИЯ БОЛЕЕ ВЫСОКОГО УРОВНЯ, ЧЕМ ПОТОК.**

**Класс Task описывает отдельную продолжительную операцию, которая запускается АСИНХРОННО в одном из потоков пула потоков – подобна потокам, но абстракция более высокого уровня. TPL — это абстракция, в которой мы оперируем понятием задач, а не потоков (примечание: работа с потоками — это низкоуровневые операции).**

Класс Task имеет ряд свойств, с помощью которых мы можем получить информацию об объекте. Некоторые из них:

* **AsyncState**: возвращает объект состояния задачи
* **CurrentId**: возвращает идентификатор текущей задачи
* **Exception**: возвращает объект исключения, возникшего при выполнении задачи
* **Status**: возвращает статус задачи

**!!!! Состояния задачи: Running, Faulted, RanToCompleted, Canceled, WaitingToRun, Created.**



Эффективность параллельного алгоритма существенно зависит от элементов массива, числа потоков, сложности математической функции и т.д. **Следует учитывать, что при малом объеме элементов массива, накладные****расходы, связанные с организацией многопоточной обработки, превышают выигрыш от параллельности обработки.** При последовательном выполнении примитивной циклической обработки быстродействие достигается за счет оптимального использования кэш-памяти.

**Идея параллелизма на основе задач состоит в том, чтобы разбить работу на небольшие задачи, которые могут дать результаты в будущем, а затем эффективно распределить ровно столько потоков, сколько есть процессоров, чтобы вы не тратили время на переключение контекста**.

* **Task по умолчанию использует пул потоков ,** в то время как прямое использование Thread потребует создания нового потока.
* **Task будет обрабатывать исключения и результаты**, поэтому его использование будет проще.
* **Task может поддерживать отмену при использовании потоков**, вы должны реализовать ее самостоятельно.
* **Большая часть работы выполняется фреймворком.**
* **ЗАДАЧА ИМЕЕТ БОЛЕЕ ВЫСОКИЙ УРОВЕНЬ АБСТРАКЦИИ!**
* Более эффективное и масштабируемое использование системных ресурсов.
* В фоновом режиме задачи помещаются в очередь [ThreadPool](https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.threadpool), усовершенствованную с помощью алгоритмов, которые определяют и настраивают количество потоков и обеспечивают балансировку нагрузки. **Это делает задачи относительно простыми и позволяет создавать множество задач для использования точного параллелизма.**
* Больший программный контроль по сравнению с потоком или рабочим элементом. Задачи и построение платформы на их основе предоставляют богатый набор интерфейсов API, которые поддерживают ожидание, отмену, продолжения, надежную обработку исключений, подробные состояния.

**Задача - это то, что вы хотите сделать (например, выполнить метод). Она может выполнятся как синхронно, так и асинхронно, но для её выполнения необязательно создавать отдельный поток, может подойти и уже существующий. Потоки же - это те, кто выполняет поставленные им задания, они так сказать «работники», выполняющие те же задачи.**

# **Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения.**

**1 СПОСОБ.** Для определения и запуска задачи можно использовать различные способы. **Первый способ создание объекта Task и вызов у него метода Start:**

|  |
| --- |
| Task task = new Task(() => Console.WriteLine("Hello Task!"));  task.Start(); |

В качестве параметра объект Task принимает делегат Action, то есть мы можем передать любое действие, которое соответствует данному делегату, например, лямбда-выражение, как в данном случае, или ссылку на какой-либо метод. То есть в данном случае при выполнении задачи на консоль будет выводиться строка "Hello Task!".**А метод Start() собственно запускает задачу.**

**2 СПОСОБ.** Второй способ заключается в использовании статического **метода Task.Factory.StartNew(). Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться.** При этом этот метод сразу же запускает задачу:

|  |
| --- |
| **Task task = Task.Factory.StartNew**(() => Console.WriteLine("Hello Task!")); |

В качестве результата метод возвращает запущенную задачу.

**3 СПОСОБ.** Третий способ определения и запуска задач представляет использование **статического метода Task.Run():**

|  |  |
| --- | --- |
|  | Task task = Task.Run(() => Console.WriteLine("Hello Task!")); |

**Метод Task.Run() также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.**

Организовать ожидание завершения задач можно совершенным способом, **применяя МЕТОДЫ ОЖИДАНИЯ**, специально предоставляемые в классе Task. **Самым простым из них считается метод Wait(), приостанавливающий исполнение вызывающего потока до тех пор, пока не завершится вызываемая задача. WaitAll().** Этот метод организует ожидание завершения группы задач. Возврата из него не произойдет до тех пор, пока не завершатся все задачи.

**ВОЗВРАТ РЕЗУЛЬТАТА. Иногда требуется организовать ожидание до тех пор, пока не завершится любая из группы задач. Для этой цели служит метод WaitAny().**

public Task(Func<TResult> функция)

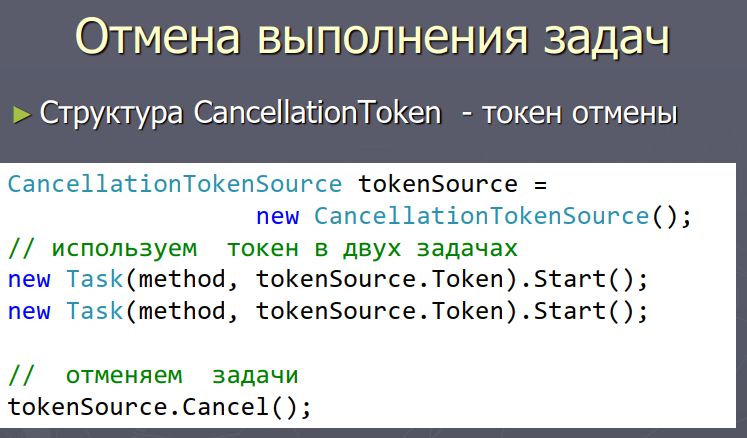
public Task(Func<Object, TResult> функция, Object состояние)

**где функция обозначает выполняемый делегат**

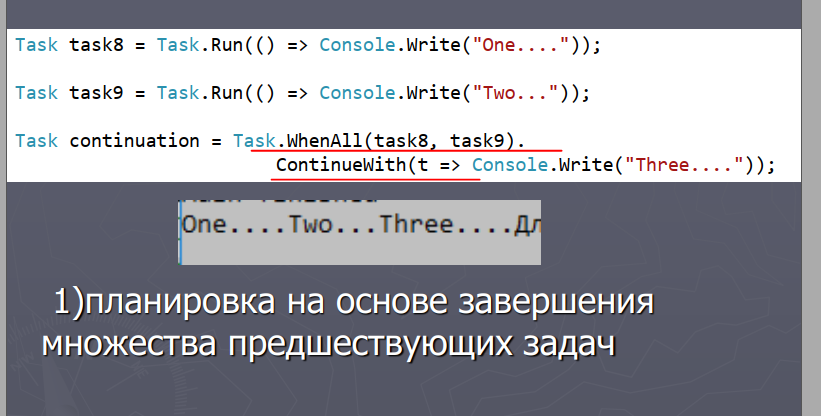
**Тип Func используется именно в тех случаях, когда задача возвращает результат**. В первом конструкторе создается задача без аргументов, а во втором конструкторе — задача, принимающая аргумент типаObject, передаваемый как состояние.

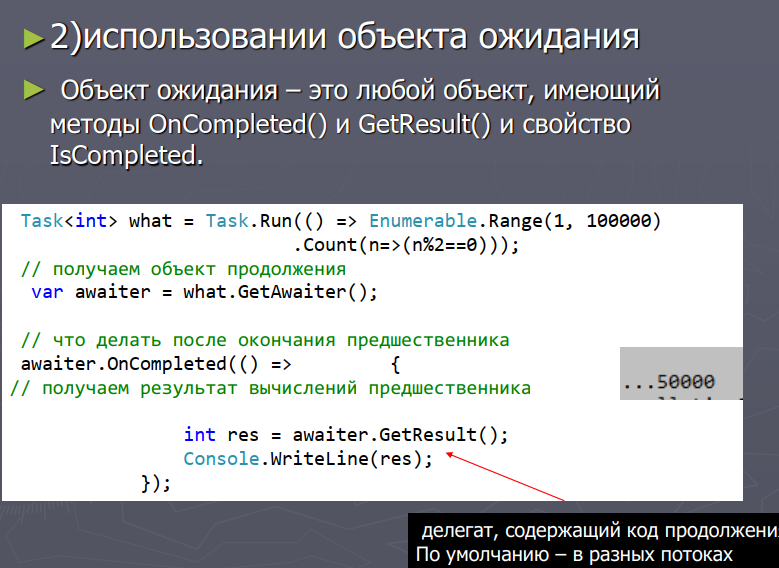
**ОБРАБОТКА ИСКЛЮЧЕНИЙ.** При генерации исключений все они агрегируются в одном исключении типа AggregateException.

**ОТМЕНА ВЫПОЛНЕНИЯ.**



**Задачи продолжения или continuation task позволяют определить задачи, которые выполняются после завершения других задач, сообщают задаче, что после ее завершения она должна продолжить делать что-то другое**. Благодаря этому мы можем вызвать после выполнения одной задачи несколько других, определить условия их вызова, передать из предыдущей задачи в следующую некоторые данные – **методы ContinueWith, WhanAll. ОСУЩЕСТВЛЯЕТСЯ ПЛАНИРОВКА на основе завершения множества предыдущих задач.**



**GetAwaiter() – взять объект ожидания!!!!**

# **Параллелизм при императивной обработке данных. Класс Parallel**

Одним из главных классов в TPL является System.Threading.Tasks.Parallel. Этот класс поддерживает набор методов, которые позволяют выполнять **ИТЕРАЦИИ ПО КОЛЛЕКЦИИ ДАННЫХ** (точнее, по объектам, реализующим [IEnumerable<T>](https://professorweb.ru/my/csharp/charp_theory/level12/12_19.php)) **В ПАРАЛЛЕЛЬНОМ РЕЖИМЕ**. Заглянув в документацию .NET Framework 4.0 SDK, вы увидите, что этот класс поддерживает два статических метода — **Parallel.For() и Parallel.ForEach()**

**Эти методы позволяют создавать тело операторов кода, которое может выполнятся в параллельном режиме**. Концептуально эти операторы **представляют собой логику того же рода, которая была бы написана в нормальной циклической конструкции (с использованием ключевых слов C# for и foreach).** Однако **их преимущество состоит в том, что класс Parallel САМОСТОЯТЕЛЬНО БЕРЕТ ПОТОКИ ИЗ ПУЛА ПОТОКОВ (и управляет конкуренцией).**

**Класс Parallel также является частью TPL и предназначен для упрощения параллельного выполнения кода**. Parallel имеет ряд методов, которые позволяют распараллелить задачу.

Одним из методов, позволяющих параллельное выполнение задач, является метод Invoke. **Метод Parallel.Invoke в качестве параметра принимает массив объектов Action, то есть мы можем передать в данный метод набор методов, которые будут вызываться при его выполнении**. Количество методов может быть различным

**Метод Parallel.For позволяет выполнять итерации цикла параллельно**. Он имеет следующее определение: For(int, int, Action<int>), где первый параметр задает начальный индекс элемента в цикле, а второй параметр - конечный индекс. Третий параметр - делегат Action - указывает на метод, который будет выполняться один раз за итерацию

**Метод Parallel.ForEach осуществляет итерацию по коллекции, реализующей интерфейс IEnumerable, подобно циклу foreach**, только осуществляет параллельное выполнение перебора. Он имеет следующее определение: ParallelLoopResult ForEach<TSource>(IEnumerable<TSource> source,Action<TSource> body),

где ***первый параметр представляет перебираемую коллекцию***, а ***второй параметр - делегат, выполняющийся один раз за итерацию*** для каждого перебираемого элемента коллекции.

**Методы Parallel.ForEach и Parallel.For возвращают объект ParallelLoopResult**, наиболее значимыми свойствами которого являются два следующих:

* **IsCompleted: определяет, завершилось ли полное выполнение параллельного цикла**
* **LowestBreakIteration: возвращает индекс, на котором произошло прерывание работы цикла**

# **Асинхронные методы. async и await**

**При асинхронном вызове поток выполнения разделяется на 2 части: в одной выполняется метод, в другой – процесс программы.** **Асинхронность позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы или блоки кода. Ключевыми для работы с асинхронными вызовами в C# являются два ключевых слова: async и await, цель которых - упростить написание асинхронного кода.** Они используются вместе для создания асинхронного метода.

Асинхронный метод обладает следующими признаками:

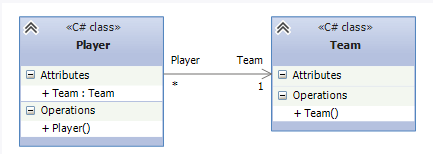
1. **В заголовке метода стоит модифактор async**
2. **Метод содержит одно/несколько выражений await**
3. **В качестве возвращаемого типа – void, Task, Task<T>, ValueTask<T>**
4. **Может быть любое количество параметров/может не быть**
5. **Не может быть ref и out параметров**
6. **Слово async не делает метод асинхронным – оно указывает, что данный метод содержит одно или несколько выражений await.**
7. **Слово await отмечает точку, в которой метод не может продолжаться до тех пор, пока ожидаемая асинхронная операция не будет завершена**

**Цель такого синтаксиса: сделать возможным код, который читается как последовательность операторов, но выполняется в гораздо более сложном порядке на основе выделения внешних ресурсов и при завершении задач**.

# **Проектирование отношений. Агрегация, композиция и ассоциация**

**Связь представляет собой семантическую взаимосвязь между классами. UML ­– унифицированный язык моделирования.**

**Наследование -** Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского). Нередко отношения наследования еще называют генерализацией или обобщением. Наследование определяет отношение IS A, то есть "является".

**Ассоциация** - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Например, объект одного типа содержит или использует объект другого типа.

class Team

{}

class Player

{ public Team Team { get; set; }}

**Композиция, «содержит как часть» -** Композиция определяет отношение **HAS A, то есть отношение "имеет".** **ЦЕЛОЕ КОНТРОЛИРУЕТ ВРЕМЯ ЖИЗНИ СВОЕЙ СОСТАВНОЙ ЧАСТИ (сильно связаны).** Это ассоциация между классом A (часть) и классом B(целое), которая дополнительно накладывает более сильные ограничения в сравнении с  
агрегацией: композиционно часть A может входить только в одно целое B, ***часть существует, только пока существует целое и прекращает свое существование вместе с целым***. Например, в класс автомобиля содержит объект класса электрического двигателя

public class ElectricEngine { }

public class Car

{     ElectricEngine engine;

    public Car()

    { engine = new ElectricEngine(); }}

**Агрегация -** От композиции следует отличать агрегацию. Она также предполагает отношение **HAS A**, но реализуется она иначе. **Включает в себя – CONTAINS, ОБЪЕКТЫ РАВНОПРАВНЫ. ВРЕМЯ ЖИЗНИ ЦЕЛОГО И ЧАСТИ НЕ СВЯЗАНЫ.** При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны. В конструктор Car передается ссылка на уже имеющийся объект Engine.

public abstract class Engine { }

 public class Car {Engine engine;

    public Car(Engine eng)

    {engine = eng;}}

# **Антипаттерны проектирования. Рефакторинг. Методы рефакторинга.**

**Антипаттерны проектирования — шаблоны ошибок, которые совершаются при решении различных задач.**

1. Спаггети-код — слабо структурированная и плохо спроектированная система, запутанная и очень сложная для понимания. Подобный код в будущем не сможет разобрать даже его автор.
2. Золотой молоток – одно решение. Уверенность в полной универсальности любого решения. На практике, это — применение одного решения (чаще всего какого-либо одного паттерна проектирования) для всех возможных и невозможных задач.
3. Программирование копи-пастом – Ctrl+C, Ctrl+V
4. Магические числа - константы, используемые в коде, но которые не несут никакого смысла без соответствующего комментария.
5. Hard-код - фиксация в коде различных данных об окружении

Жёсткое кодирование — внедрение различных данных об окружении в реализацию. Например — различные пути к файлам, имена процессов, устройств и так далее.

1. Soft-код – много абстракции
2. Слишком заумные решения - Academicals complexity
3. Лодочный якорь - сохранение неиспользуемых частей системы, которые остались после оптимизации или рефакторинга.
4. Изобретение колеса, например, своя сортировка
5. Smell-code, старый, недокументированный код
6. Слепая вера будущему пользователю, недостаточная проверка корректности входных данных, отсутствие тестирования при разработки кода и исправлении ошибок
7. Бездумное комментирование
8. God Object (божественный объект) (один класс)

**Рефакторинг (англ. refactoring) или реорганизация кода — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы**. ЭТО НЕ ОПТИМИЗАЦИЯ, НЕ ИНЖЕНЕРИНГ.

**Причины: добавить новую функцию, которая не добавляется в принятое архитектурное решение, есть ошибка и причины не ясны, сложная логика программы.**

Признаки:

* дублирование кода;
* длинный метод;
* большой класс;
* длинный список параметров;
* зависимые функции;
* избыточные временные переменные;
* не сгруппированные данные.

**Методы рефакторинга**

* Изменение сигнатуры метода (Суть изменения сигнатуры метода заключается в добавлении, изменении или удалении параметра метода)
* Инкапсуляция поля (Если у класса имеется открытое поле, необходимо сделать его закрытым и обеспечить методы доступа)
* Выделение класса
* Выделение интерфейса
* Выделение метода (Выделение метода заключается в выделении из длинного и/или требующего комментариев кода отдельных фрагментов и преобразовании их в отдельные методы, с подстановкой подходящих вызовов в местах использования)
* Встраивание (Inline)
* Введение фабрики
* Введение параметра
* Подъём метода
* Спуск метода
* Перемещение метода (Перемещение метода применяется по отношению к методу, который чаще обращается к другому классу, чем к тому, в котором сам располагается)
* Замена условного оператора полиморфизмом
* Замена наследования делегированием

# **Чистый код. Требования к именам, функциям, форматированию.**

«Чистый код»

1. логика прямолинейная
2. зависимости — минимальные
3. **стратегия ОБРАБОТКИ ОШИБОК**
4. производительность — близка к оптимальной
5. **ЧИТАБЕЛЬНЫЙ**
6. компактный
7. решает одну задачу
8. невозможно улучшить

Чтобы сказать, что код чистый и система спроектирована грамотно, легкого чтения кода недостаточно. **Он должен обладать и другими качествами**:  
**Код легко модифицировать**. При правильном проектировании и архитектуре расширение кода обходится без особых временных и технических затрат. Сущности кода не должны быть тесно связаны между собой, **код должен быть отчасти абстрактным и самодостаточным.** **Каждая сущность, которой мы оперируем при разработке, должна отвечать только за свою часть функциональности**.

**Код должен быть стабильным, предсказуемым, безопасным и надежным**. Каким бы простым код не был в чтении, **он должен быть покрыт тестами**. Хороший код и тесты всегда рядом. Причем важно не только количество тестов, но и их качество. С таким кодом не возникает проблем при запуске и отладке, он не вызывает изменений в окружающей среде.

**Защищенный код.** При написании любого кода нельзя забывать об общей безопасности продукта. **Хороший код — код, которого нет.** Это не значит, что весь код должен быть написан в одну строку, а вы — обязательно гордиться тонкими методами. **Это значит, что код не стоит дублировать, а часть общих вещей должна остаться на уровне абстракций. Теоретически, упрощение кода должно привести к уменьшению количества дефектов.** Само чтение кода тоже немаловажно. У каждого разработчика есть собственный стиль написания, а уровень чтения зависит от нашего опыта. **Все мы хотим писать простой, красивый и лаконичный код.**

**Требования к именам:**

1. Содержательные и передают намерения
2. Не длинные, не заумные
3. Не дезинформируют
4. Удобно произносятся. Различаются между собой
5. Устарели – венгерская запись, handle, m\_, CShape…, Ifactory
6. ИМЕНА КЛАССОВ – СУЩЕСТВИТЕЛЬНЫЕ
7. ИМЕНА МЕТОДОВ – ГЛАГОЛЫ
8. Содержательные имена. Но не избыточные
9. ИЗБЕГАТЬ ЦИФР, АББРЕВИАТУР
10. Одно слово для каждой концепции
11. Не использовать общие понятия. Не использовать длинные имена

**Требования к функциям:**

1. Блоки и отступы
2. Компактность. Удобство при чтении
3. ФУНКЦИЯ ДОЛЖНА ВЫПОЛНЯТЬ ТОЛЬКО ОДНУ ОПЕРАЦИЮ – Определяется уровнем абстракции
4. Чем меньше аргументов у функции, тем лучше
5. Возвращать значения через аргументы - плохо
6. Используйте исключения вместо возвращения кодов ошибок
7. Имя должно соответствовать действия и возвращаемому значению
8. НЕ ДУБЛИРОВАТЬ КОД!
9. Используйте конструкцию глагол-объект для именования методов

**Комментарии:**

1. Они часто устаревают. Вводят в заблуждение, путают.
2. Избегать избытка.
3. Использовать для предупреждения.
4. Закомментированный код удаляем
5. Не использовать /\* \*/
6. Не писать //for //if

**Требования к форматированию:**

1. Понятное название файла
2. Предпочтительно небольшой размер
3. Вертикальное сжатие
4. Переменные объявлять ближе к месту использования
5. Тесно связанные друг с другом концепции, должны находиться поблизости друг от друга по вертикали – вертикальное разделение концепций
6. Вызываемая функция должна располагаться ниже вызывающей функции (сверху-вниз)
7. Имена функции не отделяются от аргументов.
8. Знаки разделяются пробелами
9. Размер отступа соответствует позиции в иерархии
10. Длина строки <120(80) символов (экран)

# **Чистый код. Требования к классам и объектам. Закон Деметры. DTO. Избыточный код.**

«Чистый код» : логика прямолинейная ; зависимости — минимальные; **стратегия ОБРАБОТКИ ОШИБОК;** производительность — близка к оптимальной; **ЧИТАБЕЛЬНЫЙ;** компактный; решает одну задачу; невозможно улучшить

Чтобы сказать, что код чистый и система спроектирована грамотно, легкого чтения кода недостаточно. **Он должен обладать и другими качествами**:  
**Код легко модифицировать**. При правильном проектировании и архитектуре расширение кода обходится без особых временных и технических затрат. Сущности кода не должны быть тесно связаны между собой, **код должен быть отчасти абстрактным и самодостаточным.** **Каждая сущность, которой мы оперируем при разработке, должна отвечать только за свою часть функциональности**.

**Код должен быть стабильным, предсказуемым, безопасным и надежным**. Каким бы простым код не был в чтении, **он должен быть покрыт тестами**. Хороший код и тесты всегда рядом. Причем важно не только количество тестов, но и их качество. С таким кодом не возникает проблем при запуске и отладке, он не вызывает изменений в окружающей среде.

**Защищенный код.** При написании любого кода нельзя забывать об общей безопасности продукта.

**Хороший код — код, которого нет.** Это не значит, что весь код должен быть написан в одну строку, а вы — обязательно гордиться тонкими методами. **Это значит, что код не стоит дублировать, а часть общих вещей должна остаться на уровне абстракций. Теоретически, упрощение кода должно привести к уменьшению количества дефектов.**

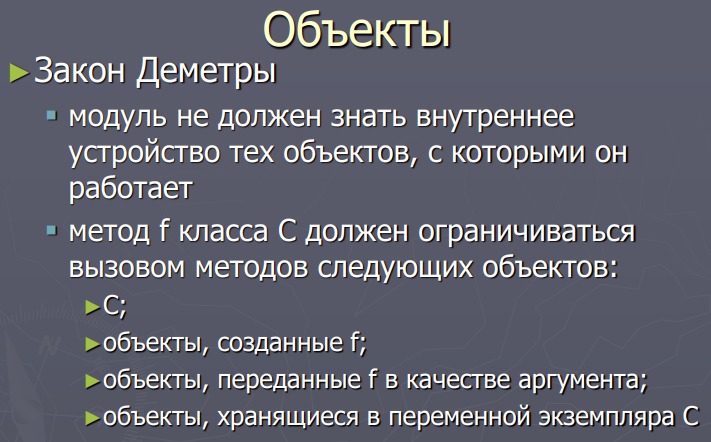
Само чтение кода тоже немаловажно. У каждого разработчика есть собственный стиль написания, а уровень чтения зависит от нашего опыта. **Все мы хотим писать простой, красивый и лаконичный код.**

**Требования к классам**

* Последовательность описания
* открытые статические константы
* приватные статические переменные
* приватные переменные
* защищённые
* открытых обычно нет
* открытые функции
* приватные функции (м.б. около открытых)
* КЛАССЫ ДОЛЖНЫ БЫТЬ КОМПАКТНЫМИ: система с множеством малых классов имеет не больше «подвижных частей», чем система с несколькими большими классами
* Имя файла должно соответствовать классу
* Имя – существительное. **НЕ АББРЕВИАТУРЫ!**
* **Названия ПОНЯТНЫЕ, БЕЗ ЦИФР.**
* Чтобы изолировать воздействие подробностей на класс, **в систему вводятся интерфейсы и абстрактные классы**

**Чем с большим количеством переменных работает метод, тем выше связность этого метода со своим классом.** **Разбиение большой функции на много мелких функций - открывает возможность для выделения нескольких меньших классов.**

**ВАЖНО: Структура системы должна быть такой, чтобы обновление системы (с добавлением новых или изменением существующих аспектов) создавало как можно меньше проблем**



Закон Деметры говорит нам о том же, о чем в детстве говорили родители: «Не разговаривай с незнакомцами». **А разговаривать можно вот с кем:  
— С методами самого объекта.**

**— С методами объектов, от которых объект зависит** напрямую.

**— С созданными объектами.**

**— С объектами, которые приходят в метод** в качестве параметра.

**— С глобальными переменными**

Благодаря закону мы получаем, что код меньше связан. **Достигается за счёт того, что классы общаются только со своими близкими родственниками** (с собой, аргументами метода и прямыми зависимостями).

**Избыточный код:**

1. **НЕ ПИСАТЬ В КОНСОЛЬ**
2. **НЕ СОЗДАВАТЬ ЛИШНИХ ПЕРЕМЕННЫХ**
3. **ИСПОЛЬЗОВАТЬ ИНВЕРСИЮ (! – знак отрицания, инверсии)**