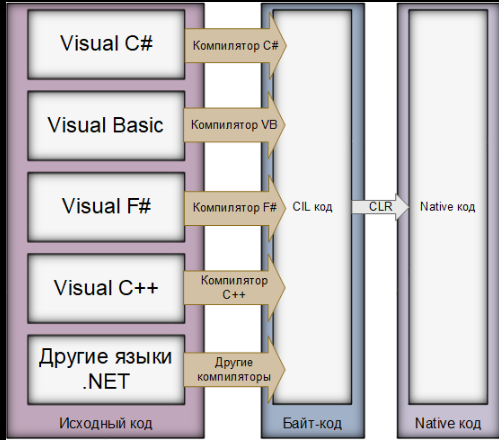
# 1. Состав. NET Framework. Структура среды выполнения CLR.

NET Framework состоит из **общеязыковой среды выполнения (среды CLR) и библиотеки классов .** **NET Framework**. Основой платформы . NET Framework является среда CLR.



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

**Portable Executable** (**PE**, «переносимый исполняемый») — [формат](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D1%82_%D1%84%D0%B0%D0%B9%D0%BB%D0%B0) [исполняемых файлов](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB), [объектного кода](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C) и [динамических библиотек](https://ru.wikipedia.org/wiki/DLL) (DLL), используемый в 32- и 64-разрядных версиях [операционной системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) [Microsoft Windows](https://ru.wikipedia.org/wiki/Microsoft_Windows).

Формат PE представляет собой структуру данных, содержащую всю информацию, необходимую [PE-загрузчику](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D1%87%D0%B8%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC) для отображения файла в память.

Сборка

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

**Common Intermediate Language** (сокращённо **CIL**) — «высокоуровневый ассемблер» виртуальной машины [.NET](https://ru.wikipedia.org/wiki/Microsoft_.NET).

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

**Common Type System** (сокр. CTS, [рус.](https://ru.wikipedia.org/wiki/%D0%A0%D1%83%D1%81%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Общая система типов*) — часть [.NET Framework](https://ru.wikipedia.org/wiki/.NET_Framework), формальная спецификация, определяющая, как какой-либо тип (класс, интерфейс, структура, встроенный тип данных) должен быть определён для его правильного выполнения средой .NET

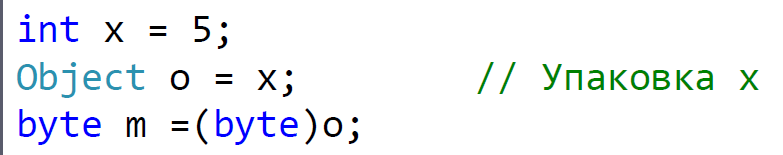
В C# существуют две разновидности типов: ссылочные типы и типы значений. **В переменных ссылочных типов хранятся ссылки на их данные (объекты), а переменные типа значений содержат свои данные непосредственно**.

# 4. Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, null-объединение

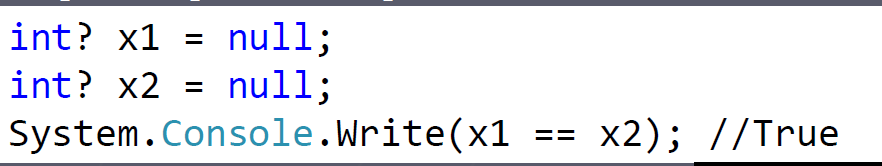
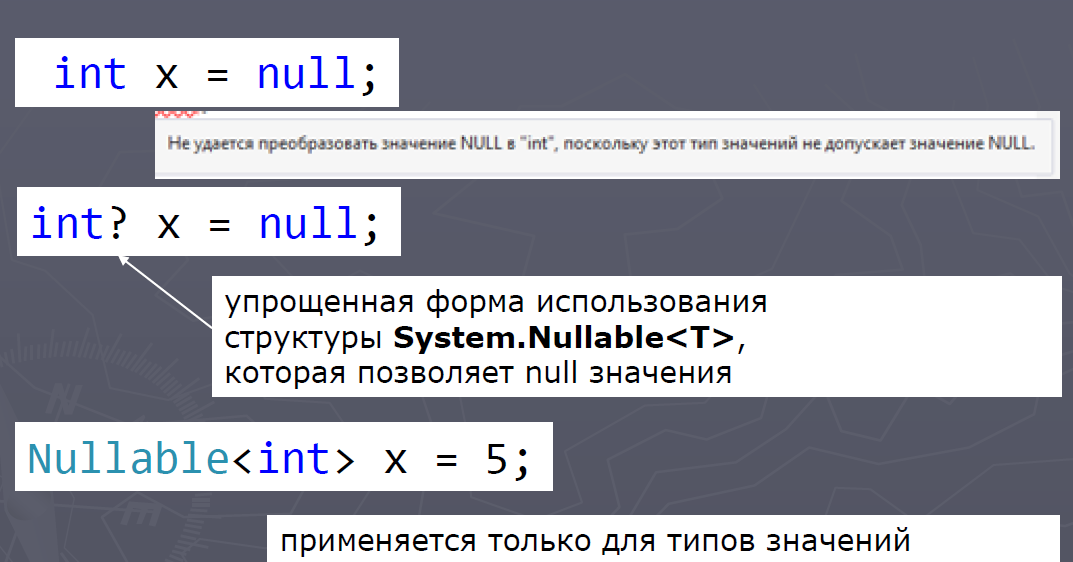
**Упаковкой (boxing)** называется процесс преобразования типа значения в тип System.Object

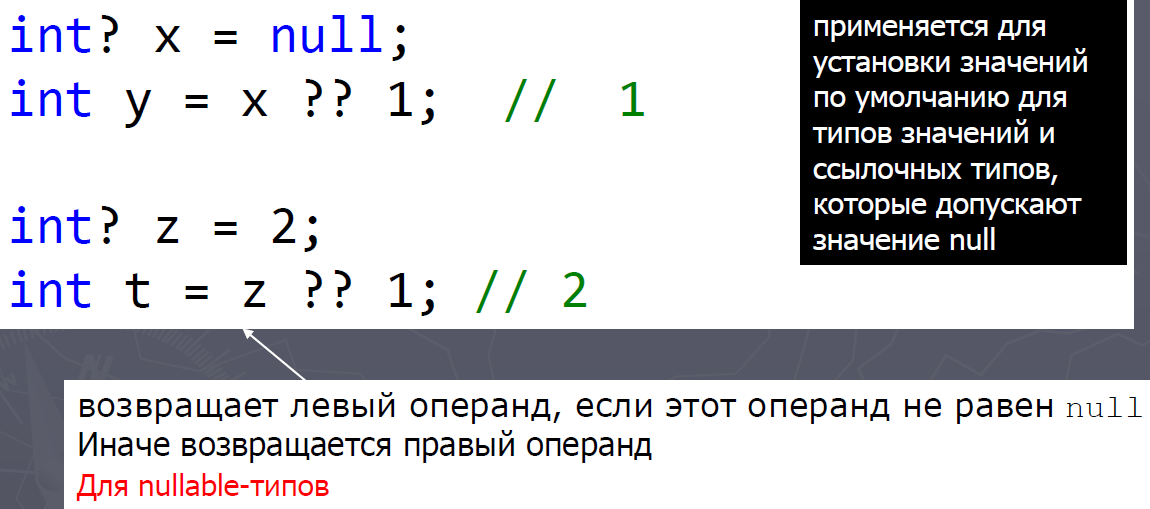
**Распаковка (unboxing) -** получение указателя на исходный значимый тип (поля данных), содержащийся в объекте

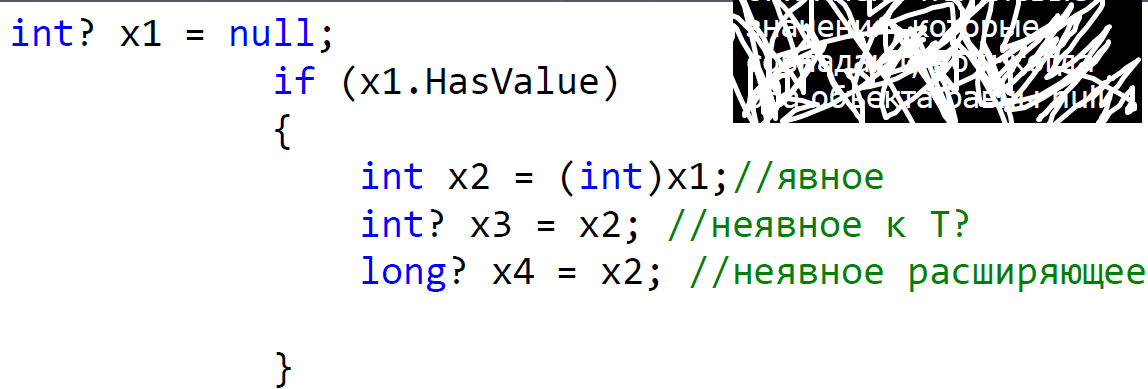
Объекты значимого типа существуют в двух формах: неупакованной (unboxed) и упакованной (boxed). Ссылочные типы бывают только в упакованной форме.



**Nullable:**

  
**Оператор ?? (null-объединение)**



**Преобразование****

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

**Тип string** предназначен для работы со строками символов в кодировке Unicode

Ему соответствует базовый класс System.String библиотеки .NET

**Операции для строк**

► присваивание (=);

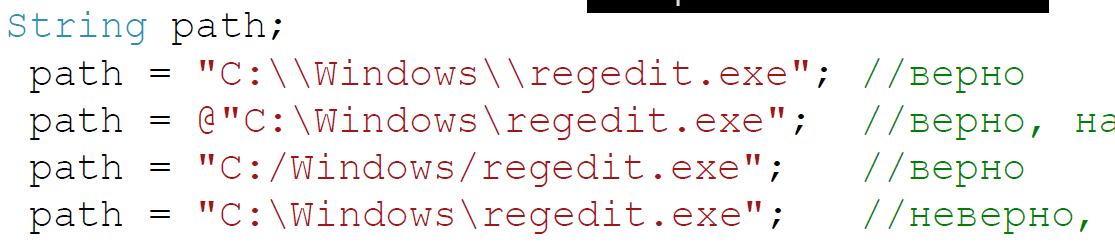
► проверка на равенство содержимого (==);

► проверка на неравенство (!=);

► обращение по индексу ([]);

► сцепление (конкатенация) строк (+)

► <,>, >=,<= - сравнивают ссылки!!!!!!!!

**Строковые литералы**

**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

**Форматированный вывод. Метод String.Format**

# 6. Неявная типизация – назначение и использование.

**Неявная типизация –** это возможность явным образом не указывать тип данных при инициализации переменной. В С# есть такая возможность: при помощи типа данных dynamic и var.

**Основное** **различие** **между** **var** **и** динамическими ключевыми словами **состоит** в том, что время привязки отличается: **var** - раннее связывание, динамическое связывание будет выполняться во время выполнения.

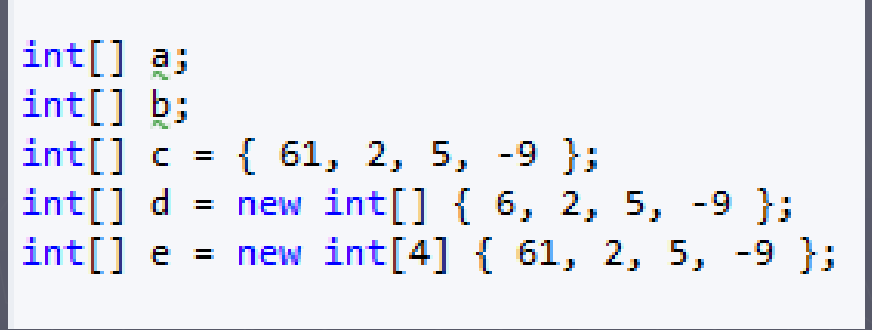
**Dynamic выполняется во время выполнения, т.е. это будет правильно:**



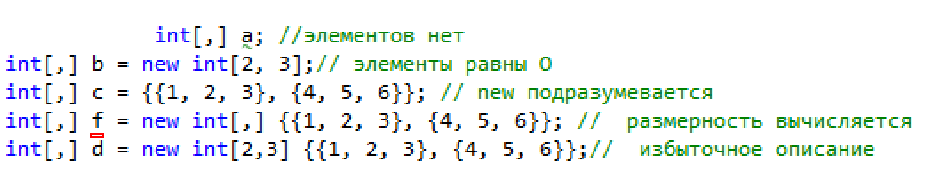
# 7. Массивы C# одномерные, прямоугольные и ступенчатые.

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

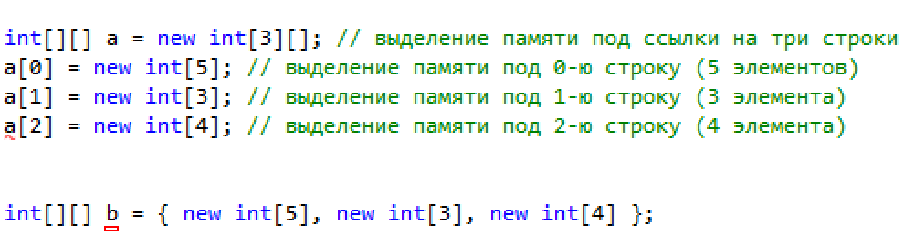
**Одномерные массивы (single-dimensional):**



**Прямоугольные (многомерные) массивы (матрица):**



**Ступенчатые, зубчатые массивы (**это массив массивов, в котором длина каждого массива разная**):**

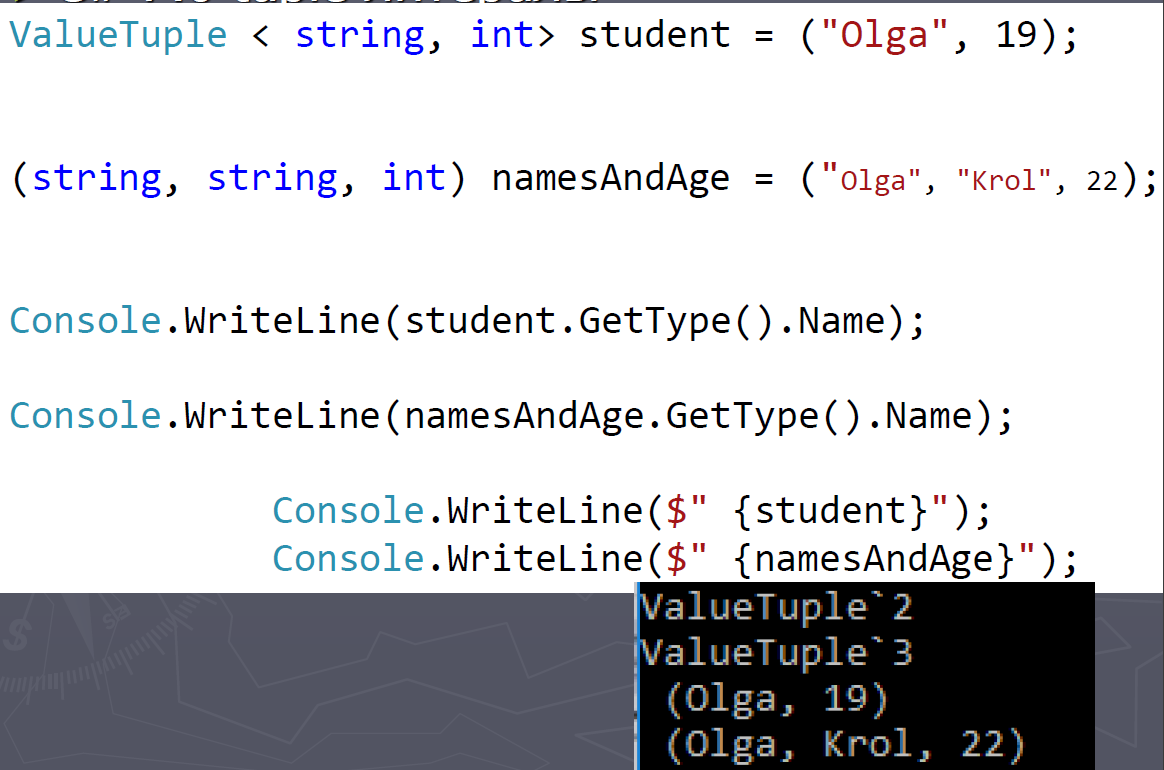


# 8. Понятие кортежей. Свойства, создание

Кортежи - набором значений

Неявное определение var tuple = (5, 10);

Явное определение (int, int) tuple = (5, 10);



**Свойства:**►создается один раз и остается неименным (все свойства доступны только для чтения)   
►позволяют использовать методы CompareTo, Equals, GetHashCode и ToString, свойство Size   
►реализуют интерфейсы IStructuralEquatable, IStructuralComparable и IComparable (можно сравнивать)

# 9. Принципы объектно-ориентированного программирования.

//https://habr.com/ru/company/otus/blog/525336/

**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.

Принципы ООП:

1. Абстрация
2. Инкапсуляция
3. Наследование
4. Полиморфизм

**Инкапсуляция** – это принцип, согласно которому любой класс и в более широком смысле – любая часть системы должны рассматриваться как «черный ящик»: пользователь класса или подсистемы должен видеть только интерфейс и не вникать во внутреннюю реализацию.

**Наследование** — это возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка (суперкласса), добавляя при необходимости новые свойства и  
методы.

**Полиморфизм** — это возможность использовать классы – потомки в контексте, который был предназначен для класса – предка.

**Абстракция данных.** Подразумевает разделение и независимое рассмотрение интерфейса и реализации. Абстракция - уровень описания/представления модели чего - либо.

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

**Класс** – это некоторое абстрактное понятие - шаблон, по которому определяется форма объекта

**Объект** – это физическая реализация класса(шаблона).

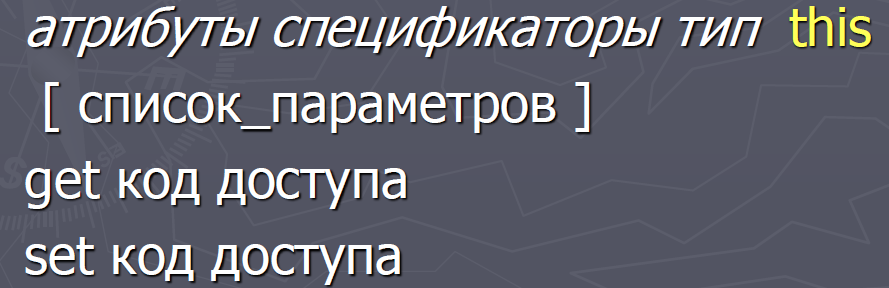
**элементам класса:** полям, методам, другим элементам класса

**Свойства** – специальные методы класса, служат для организации доступа к полям класса

**Индексаторы (свойства с параметрами)**

►Позволяют индексировать объекты таким же способом, как массив или коллекцию ►«умный» индекс для объектов

►средство, позволяющее разработчику перегружать оператор []





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

**Константы**

1) компилятор сохраняет значение константы в метаданных модуля

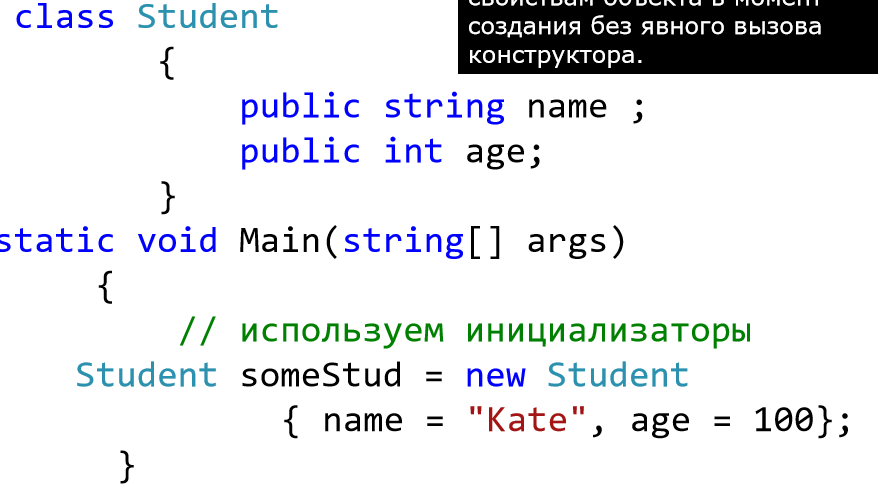
2) константы считаются неявно статическими, всегда связаны с типом, а не с экземпляром тип

3) нельзя получать адрес константы и передавать ее по ссылке

4) определять можем один раз

5) к моменту компиляции они должны быть определены.  
**Поля для чтения readonly**

1) Запись в поле разрешается при объявлении или в коде конструктора

2) Инициализировать или изменять их значение в других местах нельзя, можно только считывать их значение.  
**Инициализаторы**С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.  


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

**private:** закрытый или приватный компонент класса или структуры. Приватный компонент доступен только в рамках своего класса или структуры.

**private protected:** компонент класса доступен из любого места в своем классе или в производных классах, которые определены в той же сборке.

file: добавлен в версии C# 11 и применяется к типам, например, классам и структурам. Класс или структура с такми модификатором доступны только из текущего файла кода.

**protected:** такой компонент класса доступен из любого места в своем классе или в производных классах. При этом производные классы могут располагаться в других сборках.

**internal:** компоненты класса или структуры доступен из любого места кода в той же сборке, однако он недоступен для других программ и сборок.

**protected internal**: совмещает функционал двух модификаторов protected и internal. Такой компонент класса доступен из любого места в текущей сборке и из производных классов, которые могут располагаться в других сборках.

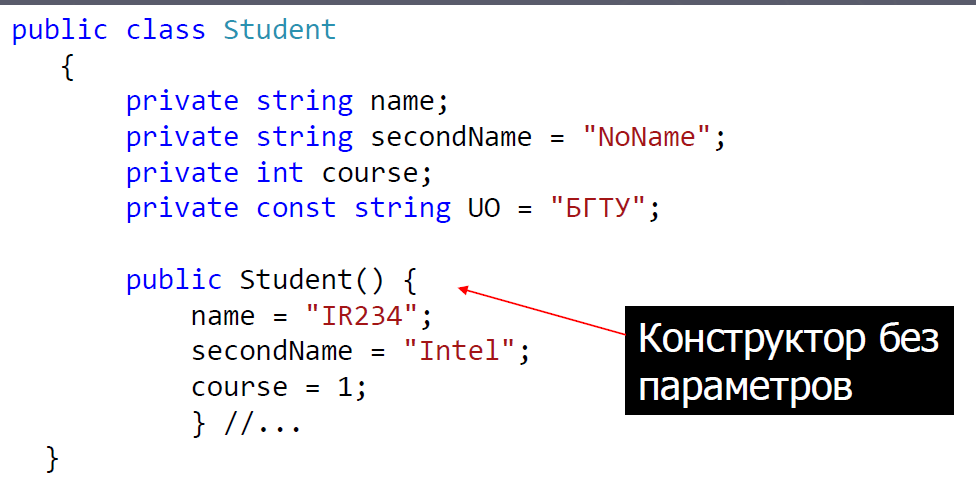
**public:** публичный, общедоступный компонент класса или структуры. Такой компонент доступен из любого места в коде, а также из других программ и сборок.



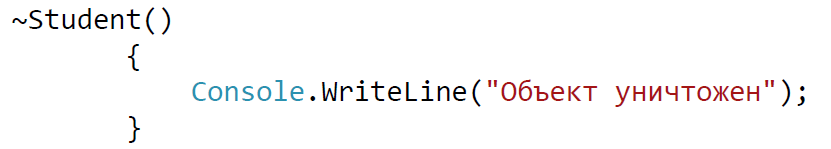
# 13. Класс. Конструкторы и их свойства. Деструкторы

**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 отличаться

**Деструкторы**  
► вызываться непосредственно перед окончательным уничтожением объекта системой "сборки мусора", чтобы гарантировать четкое окончание срока действия объекта.  
**~имя\_класса () { // код деструктора }**   


**Свойства деструктора**  
►Класс может иметь только один деструктор.

►Деструкторы не могут быть унаследованы или перегружены.

►Деструкторы невозможно вызвать. Они запускаются автоматически.

►Деструктор не принимает модификаторы и не имеет параметров.

# 14. Класс и методы System.Object.

Все классы в .NET, даже те, которые мы сами создаем, а также базовые типы, такие как System.Int32, являются неявно производными от класса Object.

**1.ToString**

Метод ToString служит для получения строкового представления данного объекта. Для базовых типов просто будет выводиться их строковое значение:

**2.** **GetHashCode**

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

**3.** **GetType**

Метод GetType позволяет получить тип данного объекта:

**4.** **Equals**

Метод Equals позволяет сравнить два объекта на равенство. В качестве параметра он принимает объект для сравнения в виде типа object и возврашает true, если оба объекта равны

# 15. Статические методы и статические конструкторы класса.

-переменные и свойства, которые хранят состояние, общее для всех объектов класса, следует определять как статические   
-методы, которые определяют общее для всех объектов поведение, также следует объявлять как статические

**Свойства статических методов:**

►отсутствует ссылка this, поскольку такой метод не выполняется относительно какого-либо объекта

►в методе static допускается непосредственный вызов только других методов типа static

►для метода static непосредственно доступными оказываются только другие данные типа static, определенные в его классе

**Статический конструктор** используется для инициализации любых статических данных или для выполнения определенного действия, которое требуется выполнить только один раз

**Свойства:**

►закрытые автоматически

►не имеет параметров

►нельзя вызвать явным образом

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

**Статические классы:**  
►прямой потомок System.Object

►экземпляры такого класса создавать запрещено

►не должен реализовывать никаких интерфейсов (не вызвать)

►нельзя использовать в качестве поля, параметра метода или локальной переменной

►от него запрещено наследовать

►все элементы такого класса должны явным образом объявляться с модификатором static

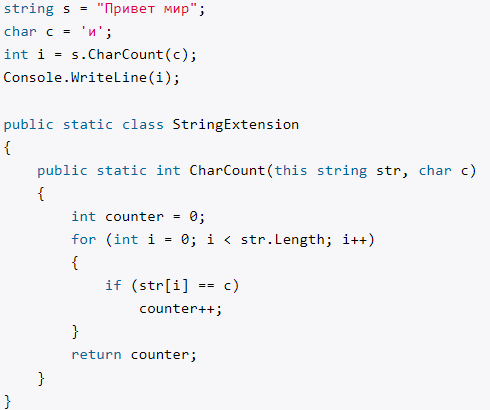
►может иметь статический конструктор

►Компилятор не создает автоматически конструктор по умолчанию  
**Методы расширения** (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса.

**Правила для методов расширений**

►1) Методы расширения должны быть объявлены в статическом необобщенном классе (первого уровня)

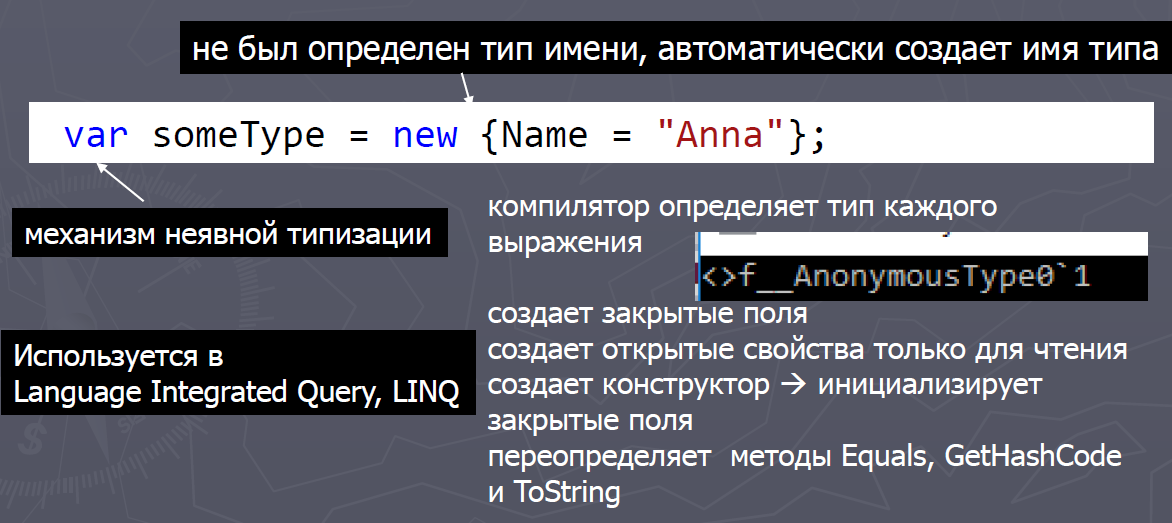
►2) this перед первым аргументом и только один



# 17. Анонимные типы.

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

свойства анонимных типов доступны только для чтения



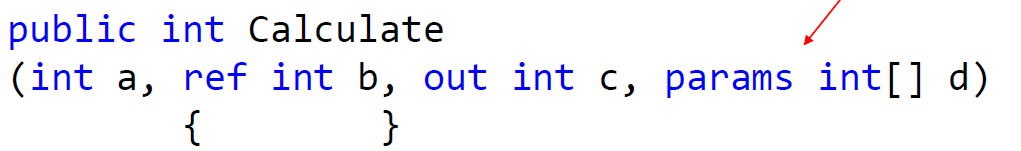
# 18. Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы.

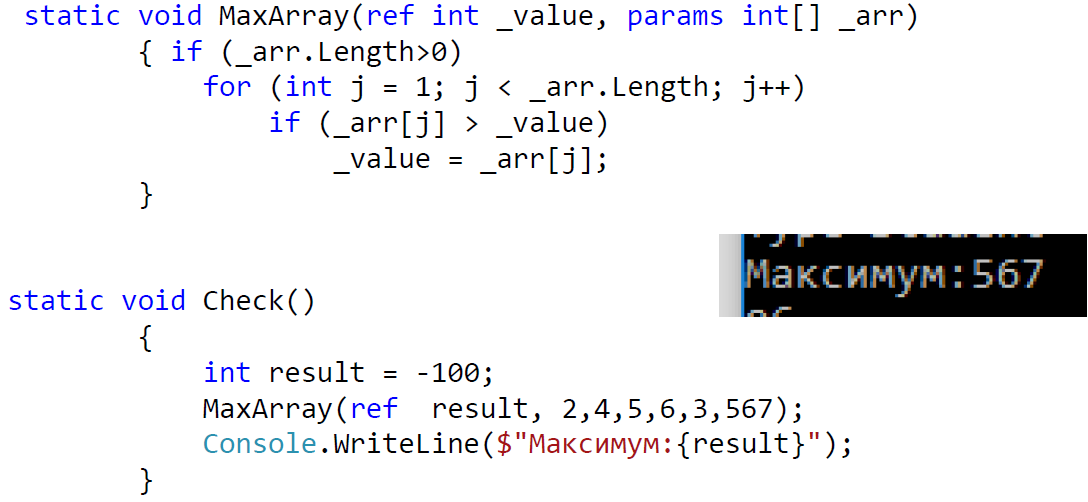
**Модификаторы параметров - ref , out, params**

**ref заставляет С# организовать вместо вызова по значению вызов по ссылке**

**Модификатор out** подобен модификатору ref за одним исключением: его можно использовать для передачи значения из методаout-параметр "поступает" в метод без начального значения, но метод (до своего завершения) обязательно должен присвоить этому параметру значение

**params** позволяет передавать методу переменное количество аргументов одного типа





**Необязательыне и именованные аргументы**

// Аргументы b и с указывать при вызове необязательно

static int mySum(int a, int b = 5, int c = 10)

Именованные аргументы освобождают от сопоставления порядка аргументов с порядком параметров в списках параметров вызываемых методов

PrintOrderDetails("Gift Shop", 31, "Red Mug");

PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");

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

**Перегрузка методов**

**►**один и тот же метод, но с разным набором параметров

► позволяет обращаться к связанным методам посредством одного, общего для всех имени.

►никакие два метода внутри одного и того же класса не должны иметь одинаковую сигнатуру сигнатура (signature) = имя метода + список его параметров (не включает тип значения, возвращаемого методом, не включает paramsпараметр)

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



**Правила:**

►префиксные операции ++ и – – перегружаются парами;

►операции сравнения перегружаются парами: == и != ; < и >;<= и >=.

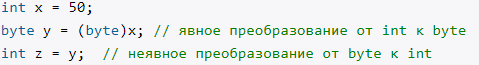
►Перегруженные операции обязаны возвращать значения

►Должны объявляться как public и static

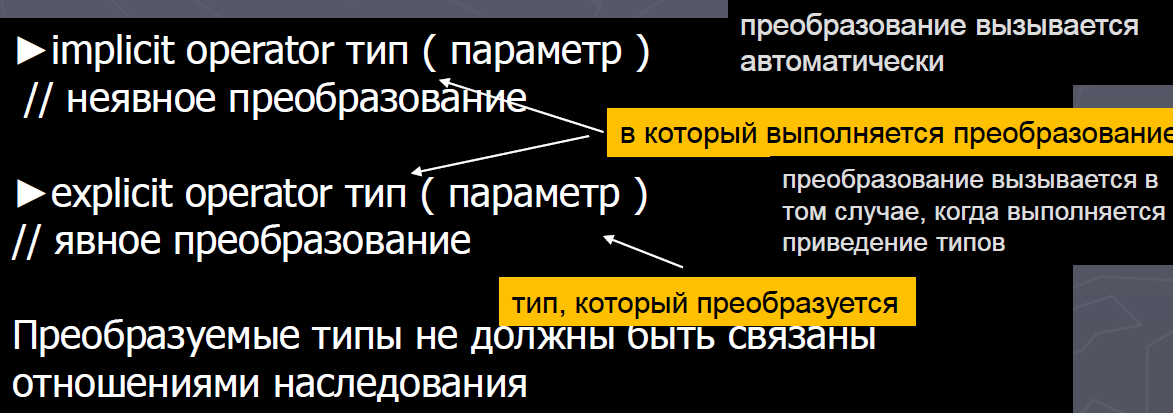
►префиксная и постфиксная формы операций ++ и --, в отличие от оригинальных операций, семантически НЕ различаются.

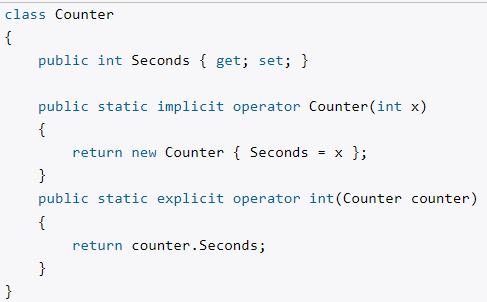
# 20. Операции преобразования типа. Явная и неявная форма. Ограничения.

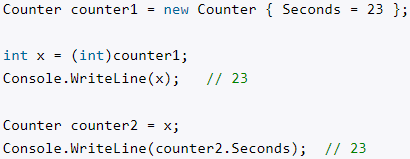
**преобразования примитивных типов**

****

**Операции преобразования типа**►преобразует объект исходного класса в другой тип   
►явная и неявна форма - будет ли этот алгоритм выполняться неявно или необходимо





Примение операторов преобразования в программе: 

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

# 21. Вложенные типы. Вложенные объекты

Тип, определенный внутри [класса](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/class), [структуры](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/struct) или [интерфейса](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/interface), называется **вложенным типом**.

вложенным типам по умолчанию присваивается модификатор [private](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/private).

В языке C# любой класс в своей реализации может содержать объявление другого класса. Класс, который объявляется в пределах фигурных скобок другого класса, называется ***вложенным классом***

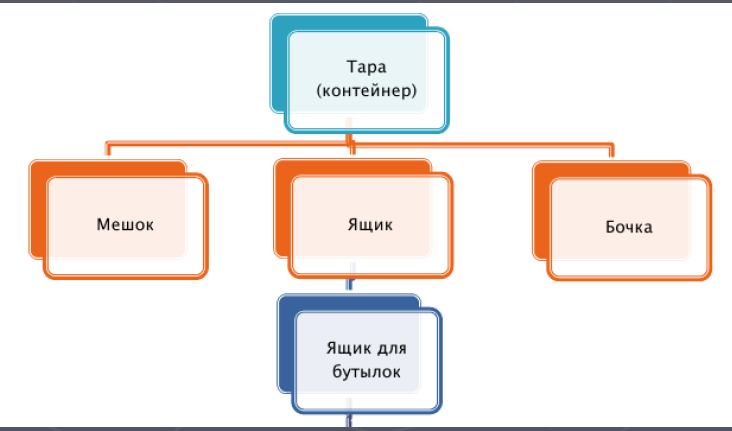
К вложенным классам могут применяться точно такие же модификаторы доступа как и к обычным – невложенным классам: private. Public. protected. internal. protected internal

# 22. Правила наследования C#.

**Наследование -** это механизм получения нового класса на основе уже существующего  
**Роль наследования**

**►**формирует иерархию

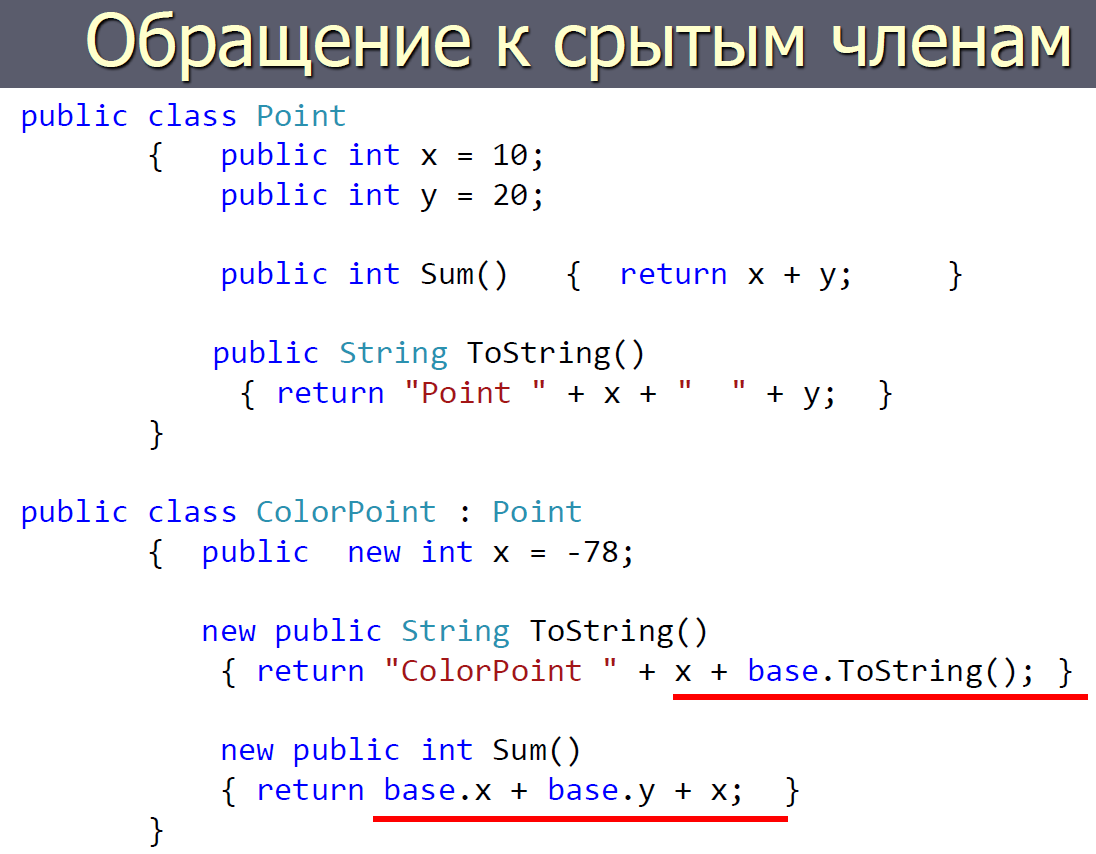
►поощряет повторное использование кода



**Правила наследования:**1) В C# наследование всегда подразумевается открытым  
2) 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 строгим

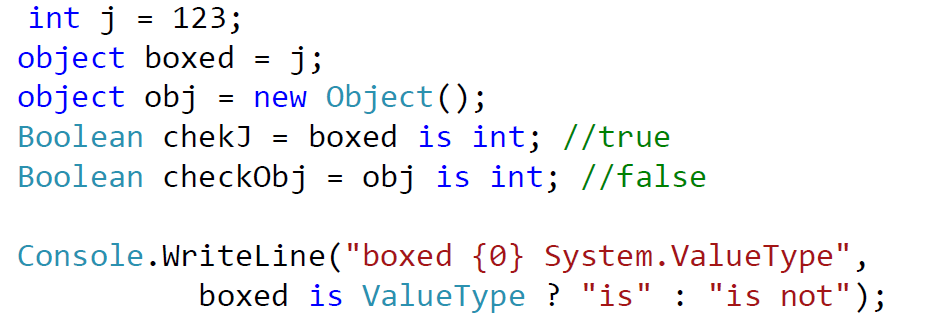
# 23. Сокрытие имен при наследовании. Обращение к срытым членам



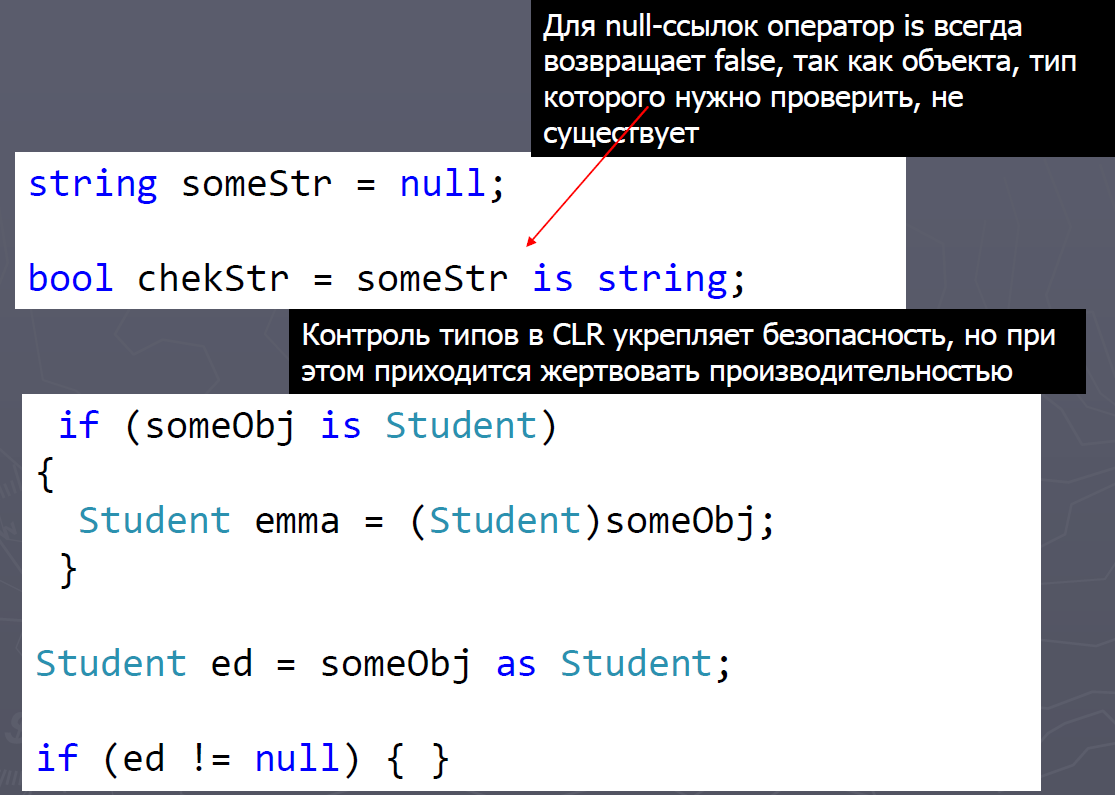


# 24. Использование операций is и as

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



**Операция as**позволяет преобразовывать тип в определенный ссылочный тип с применением следующего синтаксиса:  ***операнд as <тип>***



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

**Полиморфизм** — это возможность использовать классы – потомки в контексте, который был предназначен для класса – предка. Классическая фраза, которая коротко объясняет полиморфизм – «Один интерфейс, множество реализаций».

**Виртуальный метод** – это метод, который МОЖЕТ быть переопределен в классе-наследнике. Такой метод может иметь стандартную реализацию в базовом классе. (ключевое слово virtual)

Метод объявляется как виртуальный в базовом классе с помощью ключевого слова virtual, указываемого перед его именем. Когда же виртуальный метод переопределяется в производном классе, то для этого используется модификатор override.

**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

Нельзя использовать модификатор virtual с модификаторами static, abstract, private, или override

# 26. Понятие раннего и позднего связывания.

Раннее связывание – связанное с формированием кода на этапе компиляции. При раннем связывании, программный код формируется на основе известной информации о типе (класс) ссылки. Как правило, это ссылка на базовый класс в иерархии классов.

Позднее связывание – связанное с формированием кода на этапе выполнения.

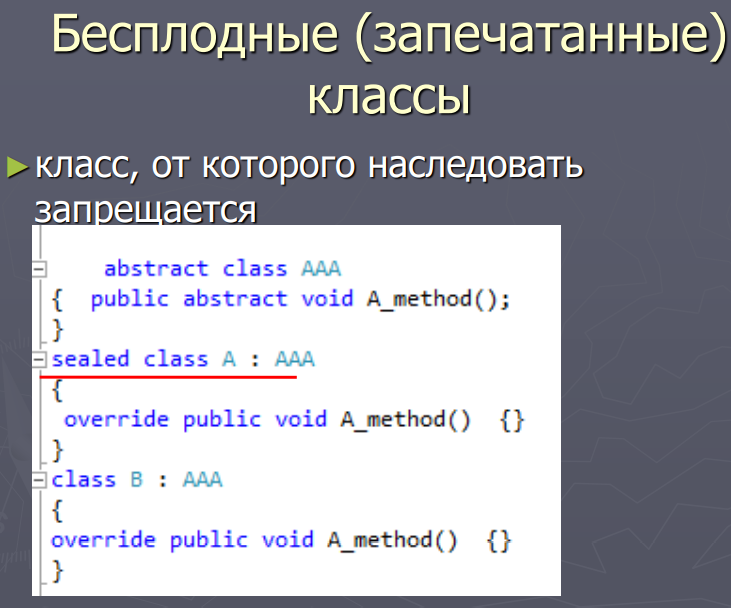
раннее связывание – адрес функции назначается во время компиляции, и именно этот адрес используется при вызове функции

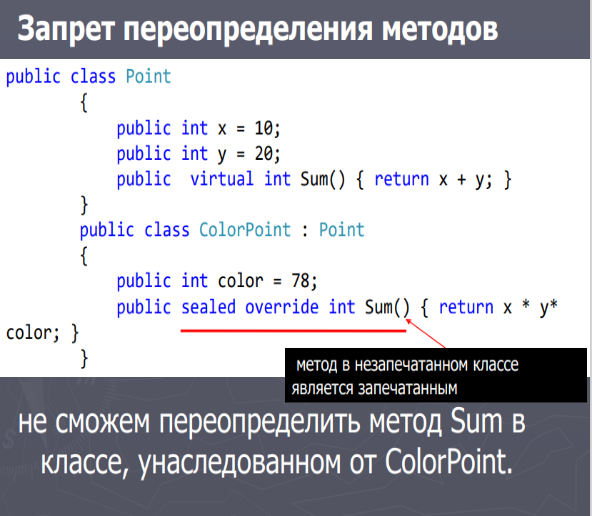
**Позднее связывание**

позднее связывание (только для методов классов) – во время выполнения приложения определяется действительный класс объекта, адрес которого находится в указателе, и вызывается метод нужного класса.



# 27. Абстрактные классы и методы. Бесплодные классы.





**Абстрактный класс** – это класс объявленный с ключевым словом *abstract*:

► Служит только для порождения потомков - предоставляют базовый функционал для классов-наследников.

► Задает интерфейс для всей иерархии

► Может содержать и полностью определенные методы, переменные, конструкторы, свойства

► Создавать объект абстрактного класса нельзя!

**Абстрактный метод** – это метод, который не имеет своей реализации в базовом классе, и он ДОЛЖЕН быть реализован в классе-наследнике. Абстрактный метод может быть объявлен только в абстрактном классе.

**Свойства abstract методов**

►1) абстрактные методы автоматически виртуальные (virtual не ставится)

►2) абстрактные методы не используются со static

# 28. Структур в C#.

**Структура** ( struct ) в C# — это пользовательский тип данных, который используется наряду с классами и может содержать какие-либо данные и методы.

Назначение : повышении эффективности и производительности программ (тип значения)

Структуры

►1) struct

►2) Может иметь конструктор c парам.

►3) до C# 10 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) или default

►5) размещение в стеке

►6) До C# 10 Нельзя инициализировать поля структуры при объявлении

►7) До C# 10 нет автоматической инициализации полей компилятором

►8) структуры не поддерживают наследование

►9) Могут реализовывать интерфейсы

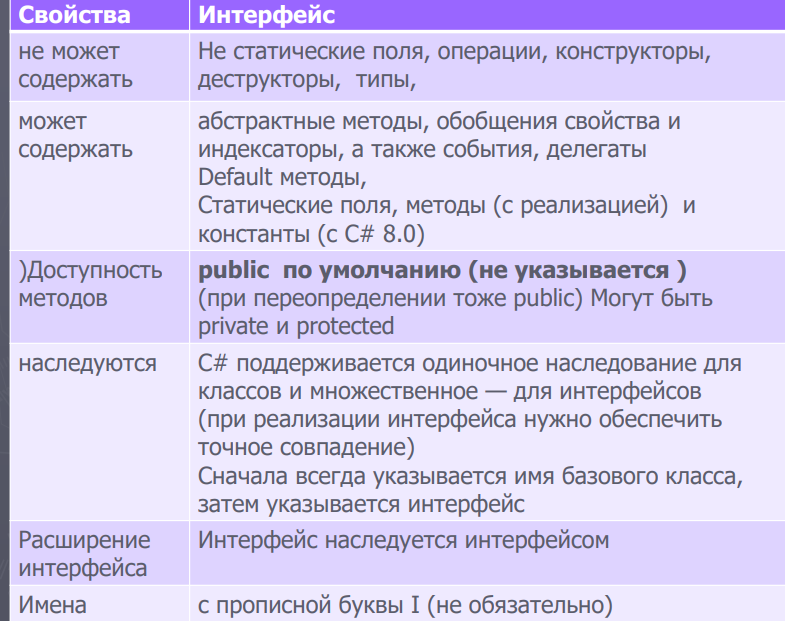
►10 ) нельзя объявить деструктор (метод завершения) в типе структуры

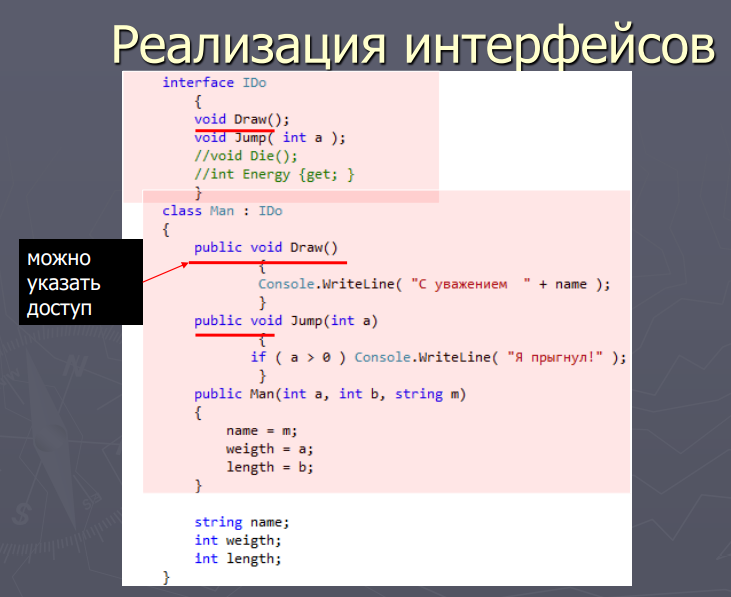
**Конструкторы в структурах**

Кроме обычных методов структура может содержать специальный метод - конструктор, который выполняет некую начальную инициализацию объекта

# 29. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов.

**Интерфе́йс** — структура программы, определяющая отношение с объектами, объединенными только некоторым поведением. При проектировании классов, разработка интерфейса тождественна разработке спецификации.





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





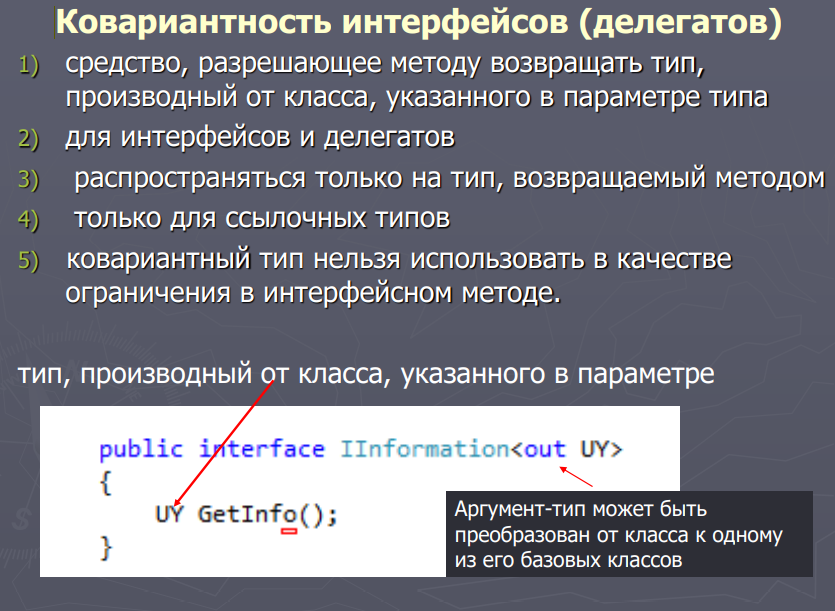
**Интерфейс** описывает только поведение (методы), и у него нет полей. Точнее, есть возможность их объявить, но они будут public static final. В то же время **абстрактный класс** может содержать классические поля, которые будут принадлежать разным объектам.

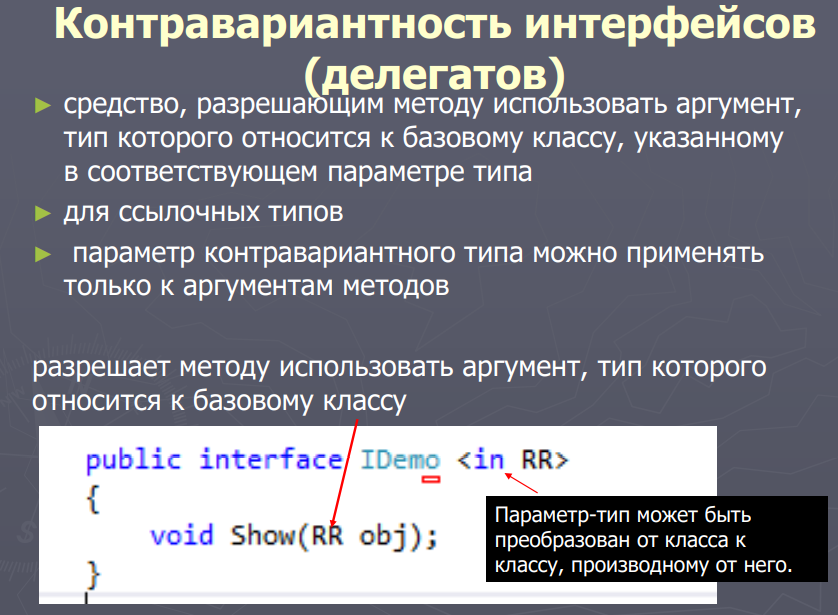
# 31. Ковариантность интерфейсов. Контравариантность интерфейсов

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

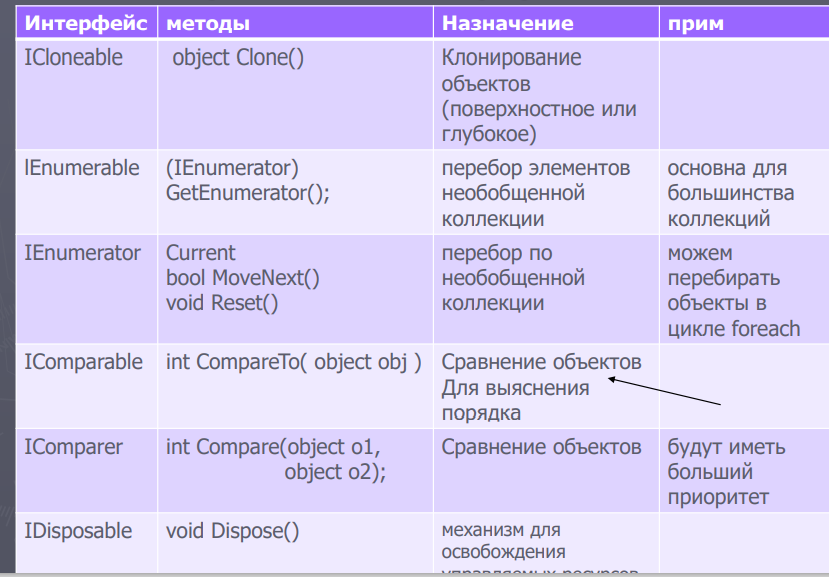
Имеется три возможных варианта поведения:

* **Ковариантность**: позволяет использовать более конкретный тип, чем заданный изначально
* **Контравариантность**: позволяет использовать более универсальный тип, чем заданный изначально
* **Инвариантность**: позволяет использовать только заданный тип



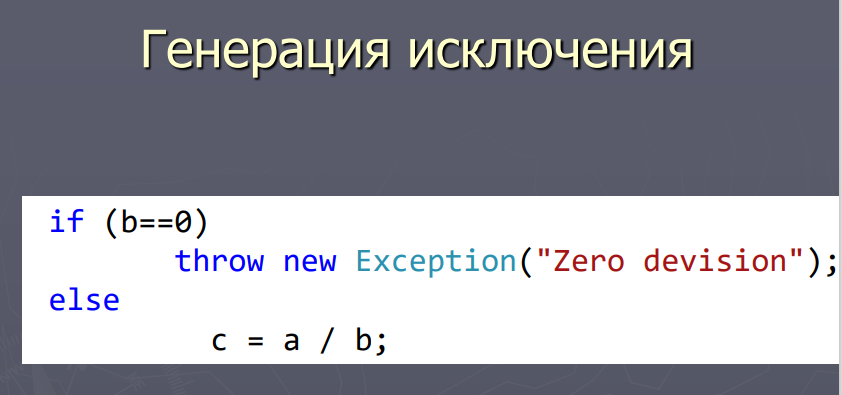


# 32. Стандартные интерфейсы .NET. Назначение и применение.



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

**Исключительная ситуация exception** - Это состояние ошибки, обнаруженное в программе в ходе ее выполнения.



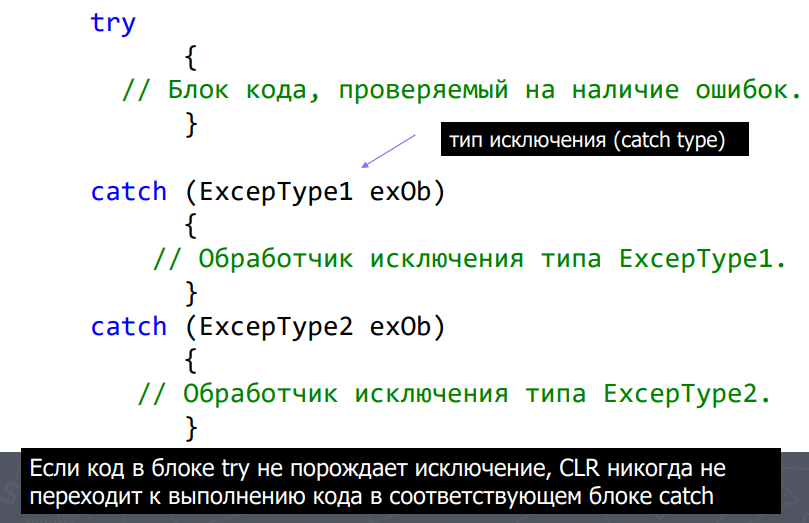
ГЕНЕРИРОВАНИЕ И РАСПОЗНАВАНИЕ ИСКЛЮЧЕНИЙ

►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

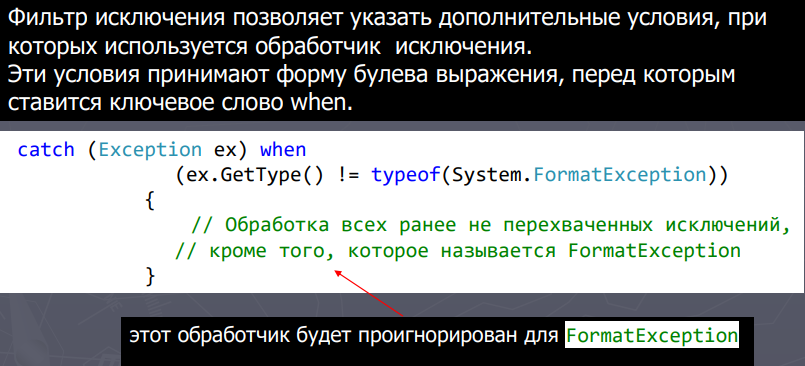


# 34. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений

**Варианты обработки исключений**

Exception универсальный обработчик исключений. Можно указать отлов конкретной ошибки, например FormatException.

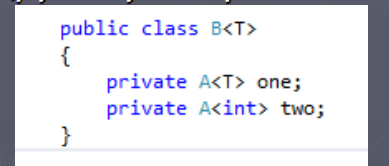
**Фильтры исключений**

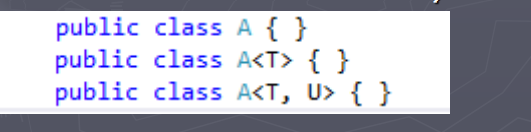
****

# 35. Обобщения (generics). Свойства обобщений.

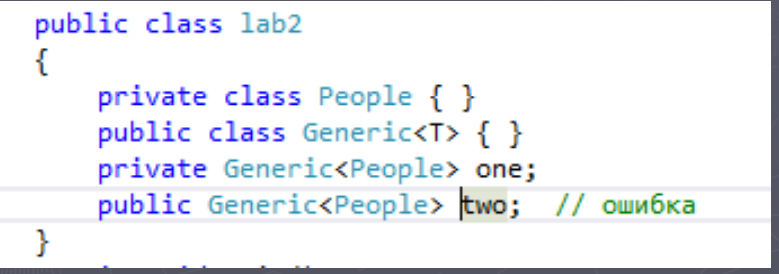
Обобщенные типы позволяют указать конкретный тип, который будет использоваться.

**свойства**

►1)Универсальный тип может содержать другой универсальный тип

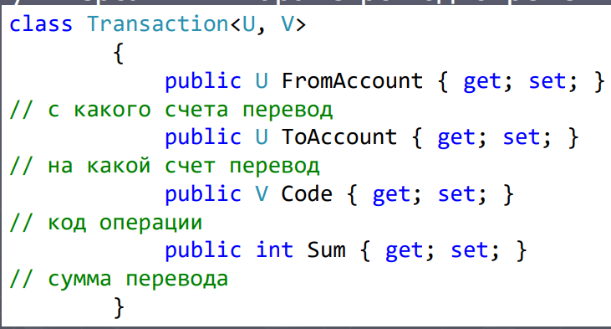
►2) Универсальные типы перегружаются на основе количества параметров (арность)

►3) Универсальными могут быть классы, структуры , интерфейсы, делегаты, методы public void Method (A iA, B iB) логическое выведение типов (type inference) ◊ используется тип данных переменной, а не фактический тип объекта, на который ссылается

 ►4) Могут содержать статические типы

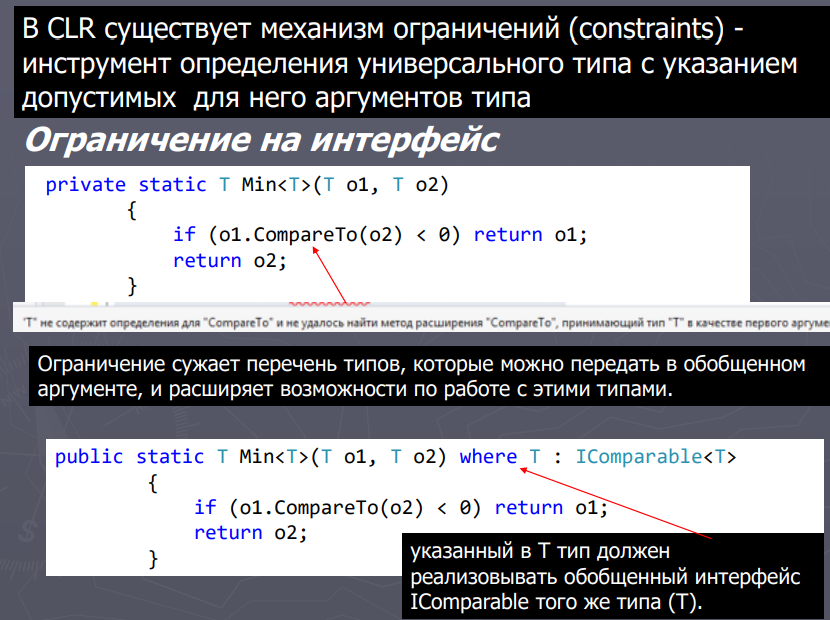
►5) Доступность конструируемых типов определяется на основе пересечения доступности универсального типа и типа в списке аргументов

►6) могут использовать несколько универсальных параметров одновременно

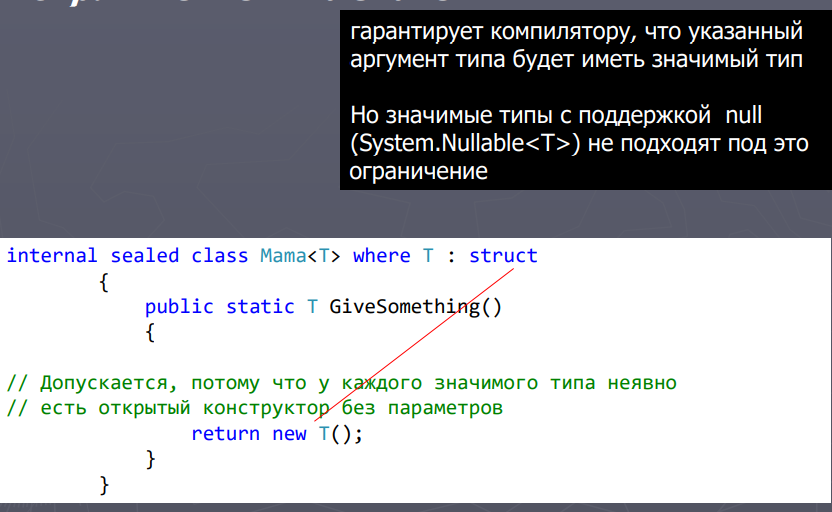


►6) поддерживает механизм ограничений

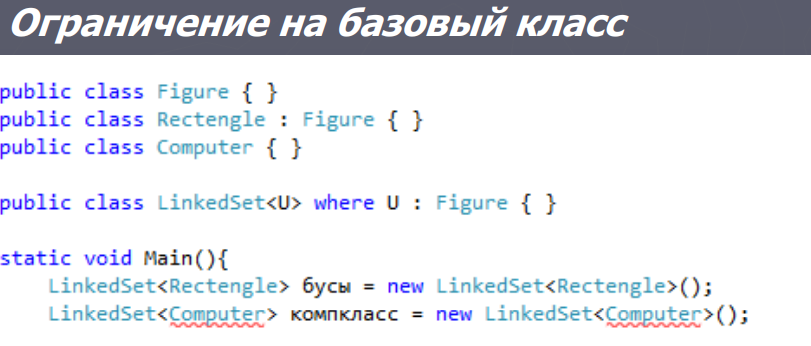
# 36. Концепция ограничений обобщений. Статические члены обобщений.

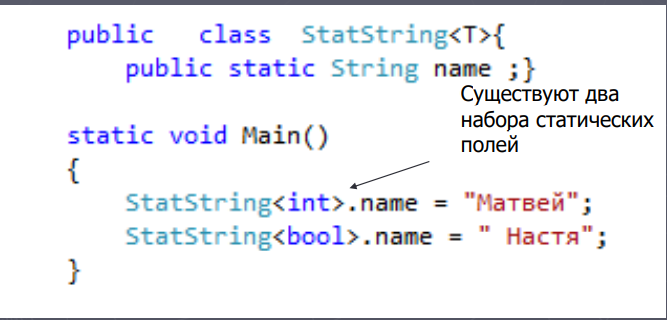


**Ограничение типа значения**

****

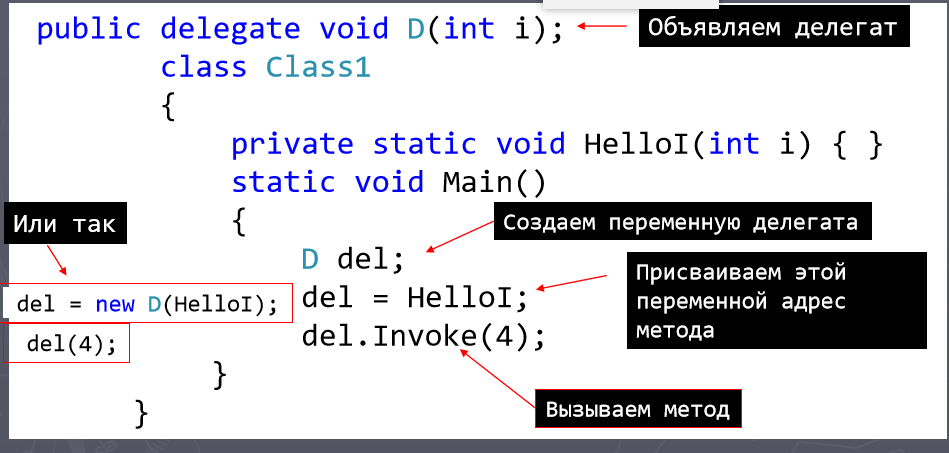
**Ограничения на базовый класс**



**Статические члены**

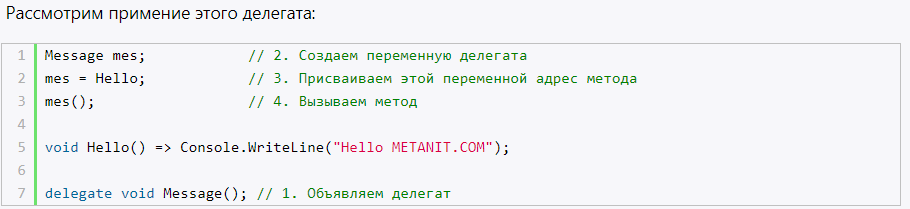
в CLR размещает статические поля типа в самом объекте-типе , каждый закрытый тип имеет свои статические поля

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

**Делегаты** представляют такие объекты, которые указывают на методы. То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

Для объявления делегата используется ключевое слово delegate





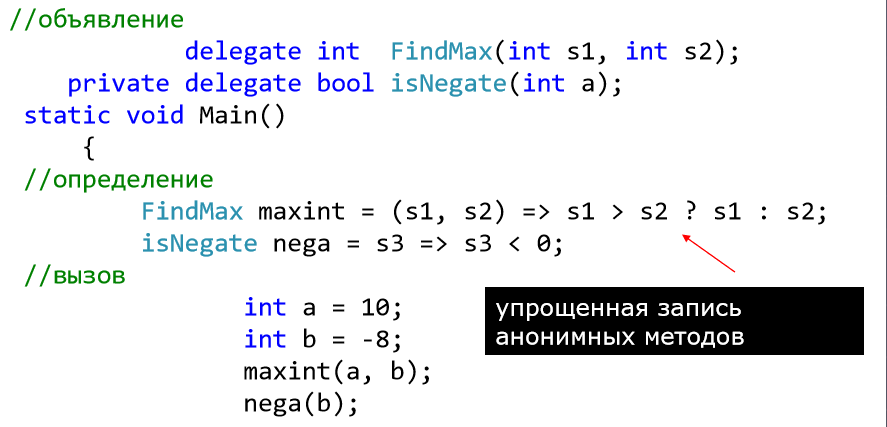
**Назначение делегатов  
►1)возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы;   
►2)обеспечения связи между объектами по типу «источник — наблюдатель»;   
►3)создания универсальных методов, в которые можно передавать другие методы;   
►4)поддержки механизма обратных вызовов.**

**Обобщенные делегаты**

delegate T MyDel<T> (T obj1, T obj2);

# 38. Анонимные функции. Лямбда-выражения.

Создание анонимных методов является, по существу, способом передачи блока кода в качестве параметра делегата.



**Лямбда-выражения**упрощенная запись анонимных методов  
**параметр => выражение   
i => i \* i;  
(список\_параметров) => выражение**

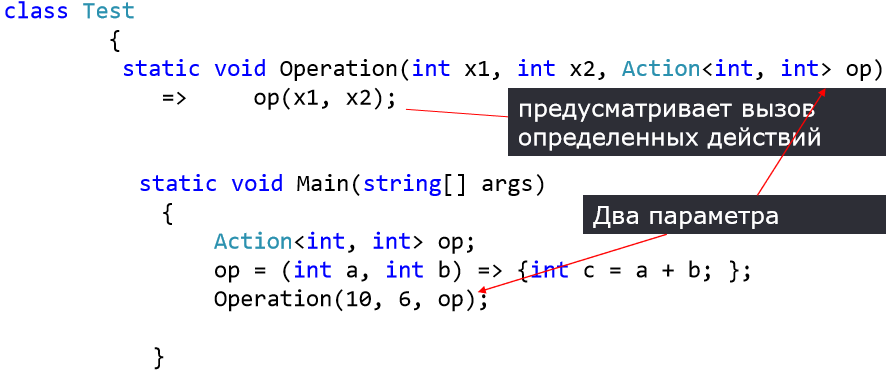
# 39. Обобщённые делегаты .NET. Action, Func, Predicate

**Action**

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

public delegate void Action()

public delegate void Action<in T>(T obj)

**Predicate**

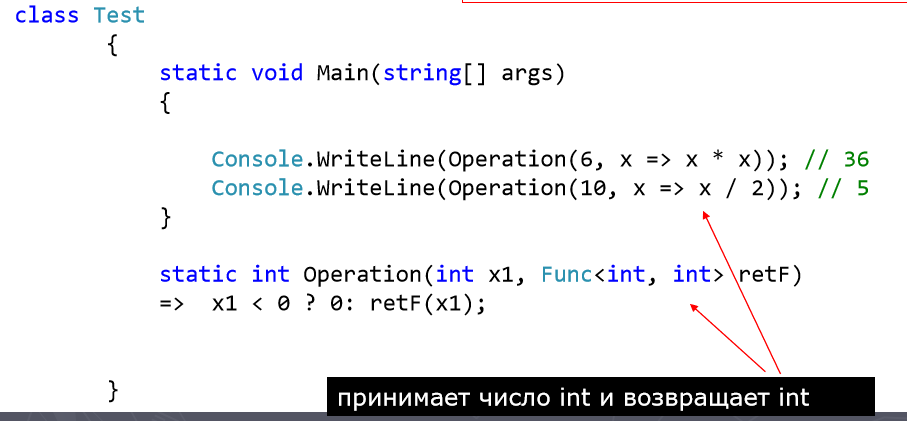
**Делегат Predicate**<T> принимает один параметр и возвращает значение типа bool: delegate bool Predicate<in T>(T obj);

**Func**

Еще одним распространенным делегатом является Func. Он возвращает результат действия и может принимать параметры.

TResult Func<out TResult>()

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



# 40. События и делегаты.

**События** сигнализируют системе о том, что произошло определенное действие. И если нам надо отследить эти действия, то как раз мы можем применять события.

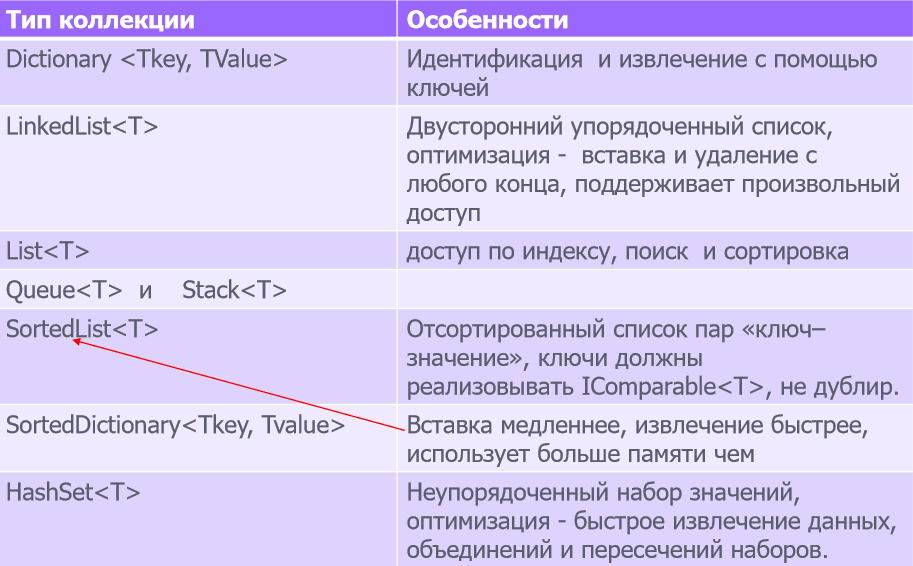
## Для возвращаемых значений требуются делегаты

|  |
| --- |
| namespace ConsoleApplication1{  delegate void UI ();  class MyEvent {  // Объявляем событие  public event UI UserEvent;  // Используем метод для запуска события  public void OnUserEvent() {  UserEvent();  }  }  class UserInfo {  string uiName, uiFamily;  int uiAge;  public UserInfo(string Name, string Family, int Age) {  this.Name = Name;  this.Family = Family;  this.Age = Age;  }  public string Name { set { uiName = value; } get { return uiName; } }  public string Family { set { uiFamily = value; } get { return uiFamily; } }  public int Age { set { uiAge = value; } get { return uiAge; } }  // Обработчик события  public void UserInfoHandler() {  Console.WriteLine("Событие вызвано!\n");  Console.WriteLine("Имя: {0}\nФамилия: {1}\nВозраст: {2}",Name,Family,Age);  }  }  class Program {  static void Main()  {  MyEvent evt = new MyEvent();  UserInfo user1 = new UserInfo(Name: "Alex", Family: "Erohin", Age: 26);  // Добавляем обработчик события  evt.UserEvent += user1.UserInfoHandler;    // Запустим событие  evt.OnUserEvent();  Console.ReadLine();  }  }  } |

# 41. Стандартные коллекции .NET. Типы коллекций.

NET Framework поддерживаются пять типов коллекций: необобщенные, специальные, с поразрядной организацией, обобщенные и параллельные.

Реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также словари, в которых можно хранить пары "ключ-значение".



# 42. Стандартные интерфесы коллекций.

**Интерфейсы, используемые в коллекциях C#   
►IEnumerable<T> (**перечислитель, с помощью которого становится возможен последовательный перебор коллекции**)** для foreach   
 GetEnumerator()   
**►IEnumerator<> (**позволяет перебирать элементы коллекции**)  
►ICollection<T>   
►IList (**описывает набор данных, которые проецируются на массив**)**

**►IList<T> (**позволяет получать элементы коллекции по порядку**)** Индексатор   
 Insert()   
 Remove()  
**►ISet<T>**

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

**►IDictionary<TKey, TValue>**

**►ICollection**

**►IComparer<T> (**сравнения двух объектов**)  
►IComparer** содержит и требует его реализации int Compare(object x, object y)   
►**IComparable** Для сортировки и сравнения объектов (SortedList)   
 Требует реализации **int CompareTo(object obj)**

# 43. IEnumerable и IEnumerator

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

public interface IEnumerable{ IEnumerator GetEnumerator();}

|  |
| --- |
| Week week = new Week();  foreach (var day in week)  {  Console.WriteLine(day);  }    class Week : IEnumerable  {  string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",  "Friday", "Saturday", "Sunday" };  public IEnumerator GetEnumerator() => days.GetEnumerator();  } |

**Интерфейс IEnumerator** определяет функционал для перебора внутренних объектов в контейнере:

public interface IEnumerator{

bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов

object Current {get;} // текущий элемент в контейнере

void Reset(); // перемещение в начало контейнера

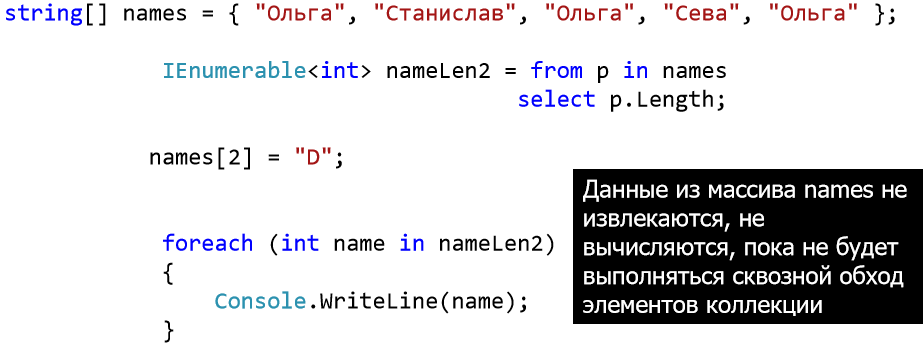
}

|  |
| --- |
| using System.Collections;  class WeekEnumerator : IEnumerator  {  string[] days;  int position = -1;  public WeekEnumerator(string[] days) => this.days = days;  public object Current {  get {  if (position == -1 || position >= days.Length)  throw new ArgumentException();  return days[position];  }  }  public bool MoveNext() {  if (position < days.Length - 1) {  position++;  return true;  }  else return false;  }  public void Reset() => position = -1;  }  class Week{  string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };  public IEnumerator GetEnumerator() => new WeekEnumerator(days);  } |

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

**LINQ(Language Integrated Query)** представляет простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс IEnumerable.

Существует несколько разновидностей LINQ:



При отложенном выполнении LINQ-выражение не выполняется, пока не будет произведена итерация или перебор по выборке, например, в цикле foreach. Обычно подобные операции возвращают объект IEnumerable<T> или IOrderedEnumerable<T>. Полный список отложенных операций LINQ:

AsEnumerable Cast Concat DefaultIfEmpty Distinct Except GroupBy GroupJoin Intersect Join OfType OrderBy OrderByDescending Range Repeat Reverse Select SelectMany Skip SkipWhile Take TakeWhile ThenBy ThenByDescending Union Where

Немедленное выполнение запроса

С помощью ряда методов мы можем применить немедленное выполнение запроса. Это методы, которые возвращают одно атомарное значение или один элемент или данные типов Array, List и Dictionary. Полный список подобных операций в LINQ:

Aggregate All Any Average Contains Count ElementAt ElementAtOrDefault Empty First FirstOrDefault Last LastOrDefault LongCount Max Min SequenceEqual Single SingleOrDefault Sum ToArray ToDictionary ToList ToLookup

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

Where – фильтрация по критерию

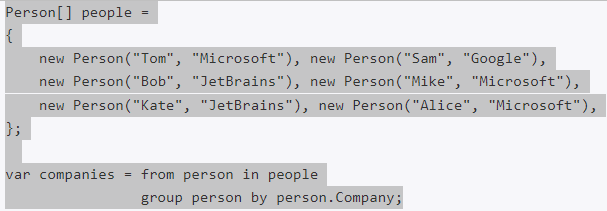
Select – выбираем объект в создаваемую коллекцию

Метод **Take()** извлекает определенное число элементов. Количество извлекаемых элементов передается в метод в качестве параметра. Например, извлечем три первых элемента: OrderB



Соединение в LINQ используется для объединения двух разнотипных наборов в один. Для соединения используется оператор join или метод **Join().** Как правило, данная операция применяется к двум наборам, которые имеют один общий критерий.

  
Для группировки данных по определенным параметрам применяется оператор **group by и метод GroupBy().**



# 46. 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

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

AssemblyName: класс, хранящий информацию о сборке

Класс **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:

FindMembers() возвращает массив объектов MemberInfo данного типа

GetConstructors() возвращает все конструкторы данного типа в виде набора объектов ConstructorInfo

# 47. Классы для работы с файловой системой.

Работу с файловой системой начнем с самого верхнего уровня - дисков. Для представления диска в пространстве имен System.IO имеется класс DriveInfo.

Этот класс имеет статический метод **GetDrives(),** который возвращает имена всех логических дисков компьютера. Также он предоставляет ряд полезных свойств:

**AvailableFreeSpace:** указывает на объем доступного свободного места на диске в байтах

**DriveFormat:** получает имя файловой системы

**DriveType:** представляет тип диска

**IsReady:** готов ли диск (например, DVD-диск может быть не вставлен в дисковод)

**Name:** получает имя диска

**RootDirectory:** возвращает корневой каталог диска

**TotalFreeSpace:** получает общий объем свободного места на диске в байтах

**TotalSize:** общий размер диска в байтах

**VolumeLabel**: получает или устанавливает метку тома

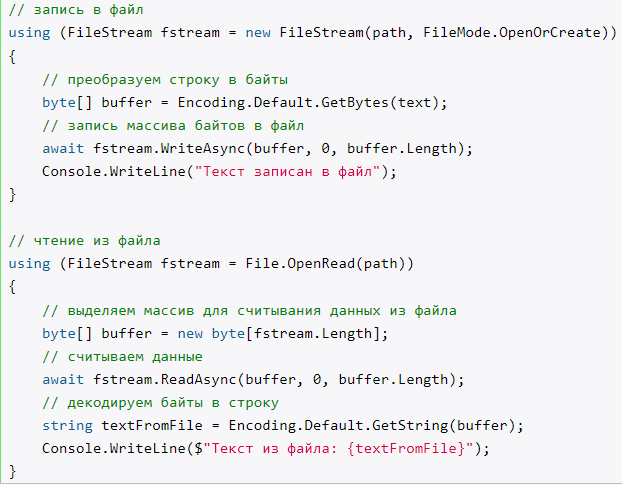
# 48. Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы.

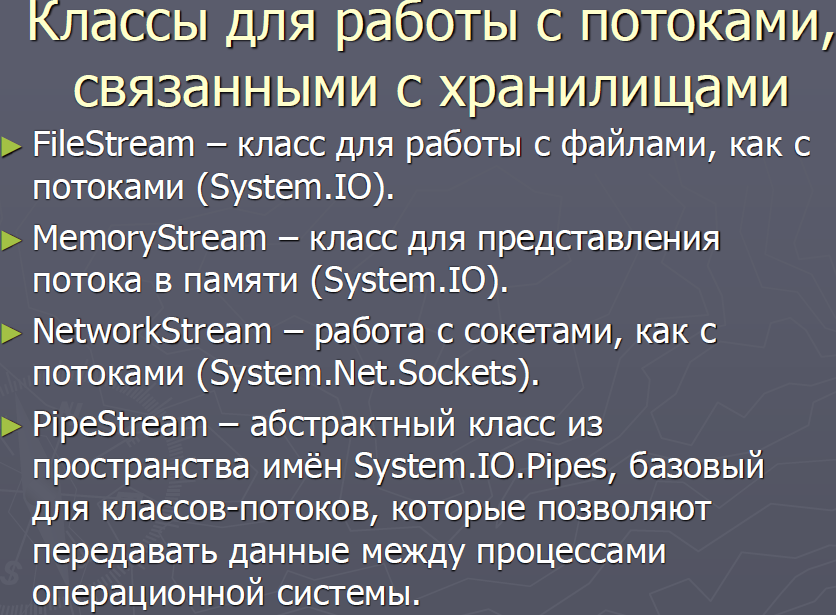
Конструкция using оформляет блок кода и создает объект некоторого типа, который реализует интерфейс IDisposable, в частности, его метод Dispose. При завершении блока кода у объекта вызывается метод Dispose – этот метод очищает память от неиспользуемых данных.

**Чтение и запись файлов**

**StreamWriter, StreamReader, BinaryWriter, BinaryReader, FileStream.**

FileStream(string filename, FileMode mode)





# 49. Классы адаптеры потоков.

Поток данных – это абстрактное представление данных в виде последовательности байт.

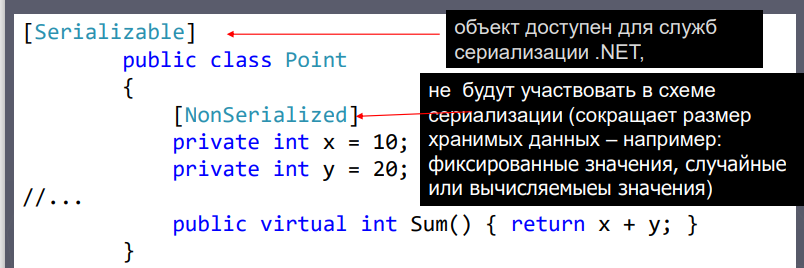
►Адаптеры потоков служат оболочкой потока, преобразуя информацию определённого формата в набор байт

Ниже перечислены основные классы, относящиеся к адаптерам потоков:

* TextReader — абстрактный класс, который может читать последовательности символов;
* TextWriter — абстрактный класс, который может 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.

# 50. Сериализация. Форматы сериализации.

Классы адаптеры потоков **Сериализация** представляет процесс преобразования какого-либо объекта в поток байтов. После преобразования мы можем этот поток байтов или записать на диск или сохранить его временно в памяти. А при необходимости можно выполнить обратный процесс - **десериализацию**, то есть получить из потока байтов ранее сохраненный объект.



**Формат сериализации**

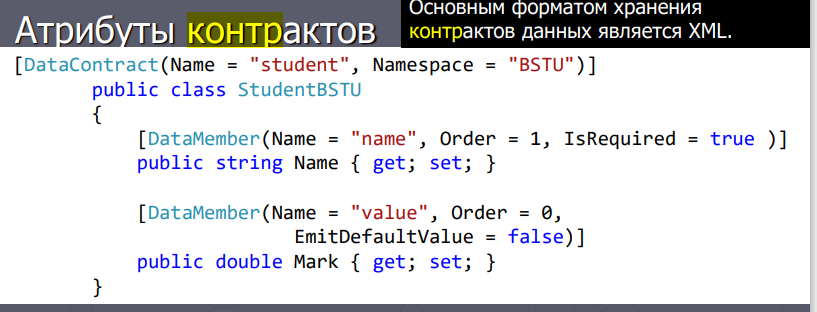
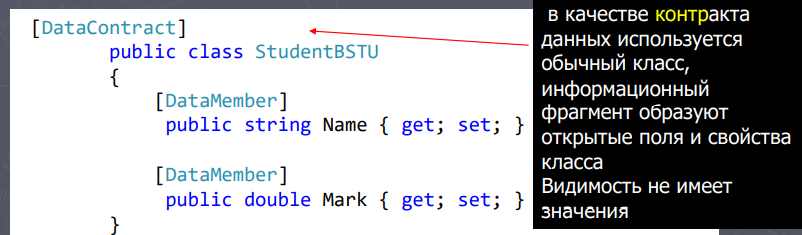
в .NET можно использовать следующие форматы:

* бинарный
* SOAP
* xml
* JSON

# 51. Сериализация контрактов данных. интерфейс ISerializable.

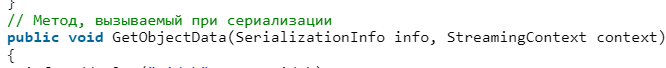
**Контракт данных** – это тип (класс или структура), объект которого описывает информационный фрагмент (открытые поля и свойства) - один из механизмов сериализации

Основным форматом хранения контрактов данных является XML



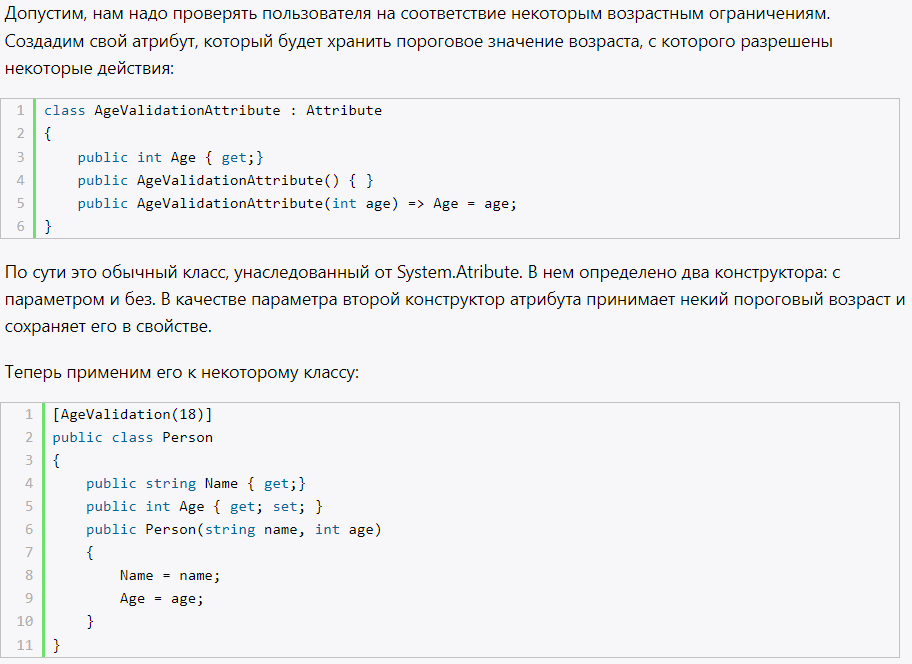
ISerializable Интерфейс

Позволяет объекту управлять его собственной сериализацией и десериализацией.



# 52. Атрибуты. Создание собственного атрибута.

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, от которого образованы все остальные классы атрибутов. В .NET имеется множество встроенных классов атрибутов. И также мы можем создавать свои собственные классы атрибутов, которые будут определять метаданные других типов.



# 53. 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.

В .NET процесс представлен классом **Process** из пространства имен System.Diagnostics. Этот класс позволяет управлять уже запущенными процессами, а также запускать новые. В данном классе определено ряд свойств и методов, позволяющих получать информацию о процессах и управлять ими.

Свойство Handle: возвращает дескриптор процесса

Свойство Id: получает уникальный идентификатор процесса в рамках текущего сеанса ОС

Свойство MachineName: возвращает имя компьютера, на котором запущен процесс

Свойство MainModule: представляет основной модуль - исполняемый файл программы, представлен объектом типа ProcessModule

Свойство Modules: получает доступ к коллекции ProcessModuleCollection, которая в виде объектов ProcessModule хранит набор модулей (например, файлов dll и exe), загруженных в рамках данного процесса

Свойство ProcessName: возвращает имя процесса, которое нередко совпадает с именем приложения

Свойство StartTime: возвращает время, когда процесс был запущен

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.

Свойство BaseDirectory: базовый каталог, который используется для получения сборок (как правило, каталог самого приложения)

Свойство CurrentDomain: домен текущего приложения

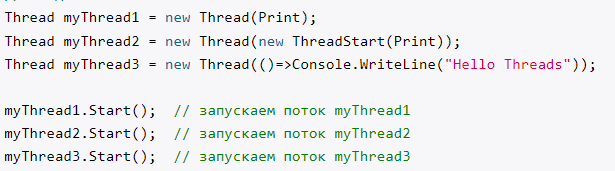
Свойство FriendlyName: имя домена приложения

Свойство SetupInformation: представляет объект AppDomainSetup и хранит конфигурацию домена приложения

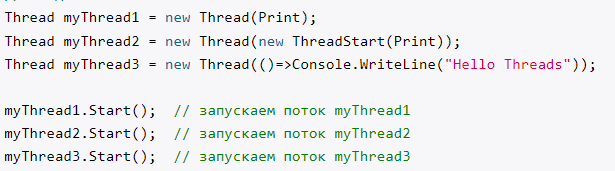
Метод ExecuteAssembly(): запускает сборку exe в рамках текущего домена приложения

Метод GetAssemblies(): получает набор сборок .NET, загруженных в домен приложения

Для создания потока применяется один из конструкторов класса **Thread**:



# 54. Создание потоков , классы приоритетов. Состояния потоков



Потоку может быть присвоено одно из следующих приоритет [ThreadPriority](https://msdn.microsoft.com/ru-ru/library/system.threading.threadpriority(v=vs.110).aspx) значения:

* **Highest**
* **AboveNormal**
* **Normal**
* **BelowNormal**
* **Lowest**

Статусы потока содержатся в перечислении **ThreadState**:

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

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

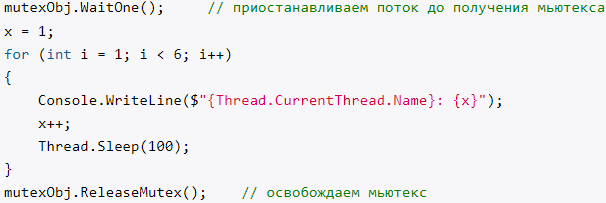
Чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово **lock**. Оператор **lock** определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока.

**Monitor**

Наряду с оператором **lock** для синхронизации потоков мы можем использовать мониторы, представленные классом System.Threading.Monitor.

**Мutex** Основную работу по синхронизации выполняют методы WaitOne() и ReleaseMutex().

****

Еще один инструмент управления синхронизацией потоков представляет класс Mutex или мьютекс, который также располагается в пространстве имен System.Threading.

**Семафоры** являются еще одним инструментом, который предлагает нам платформа .NET для управления синхронизацией.

Для работы с потоками класс Semaphore имеет два основных метода:

WaitOne(): ожидает получения свободного места в семафоре

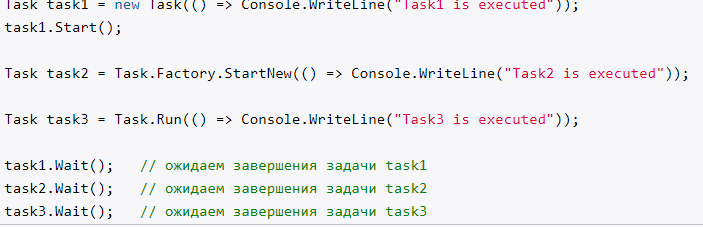
Release(): освобождает место в семафоре

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

Основной функционал **TPL** которой располагается в пространстве имен System.Threading.Tasks.

В основе библиотеки TPL лежит концепция 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.

Чтобы приложение ожидало завершения задачи, можно использовать метод Wait() объекта Task:



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

CurrentId: возвращает идентификатор текущей задачи (статическое свойство)

Id: возвращает идентификатор текущей задачи

Exception: возвращает объект исключения, возникшего при выполнении задачи

Status: возвращает статус задачи.

System.Threading.Tasks.TaskStatus, которое имеет следующие значения:

Canceled: задача отменена

Created: задача создана, но еще не запущена

Faulted: в процессе работы задачи произошло исключение

RanToCompletion: задача успешно завершена

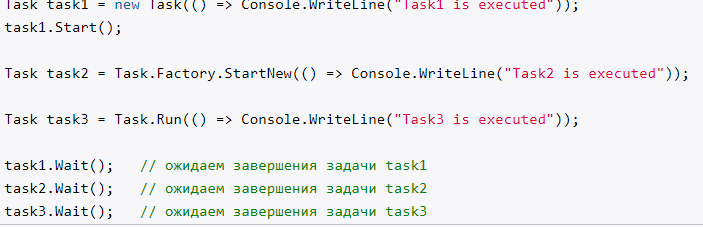
Running: задача запущена, но еще не завершена

WaitingForActivation: задача ожидает активации и постановки в график выполнения

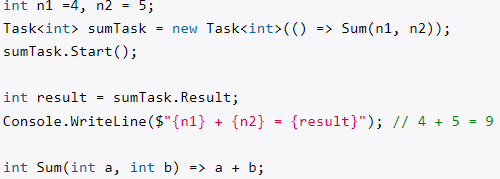
WaitingForChildrenToComplete: задача завершена и теперь ожидает завершения прикрепленных к ней дочерних задач

WaitingToRun: задача поставлена в график выполнения, но еще не начала свое выполнение

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

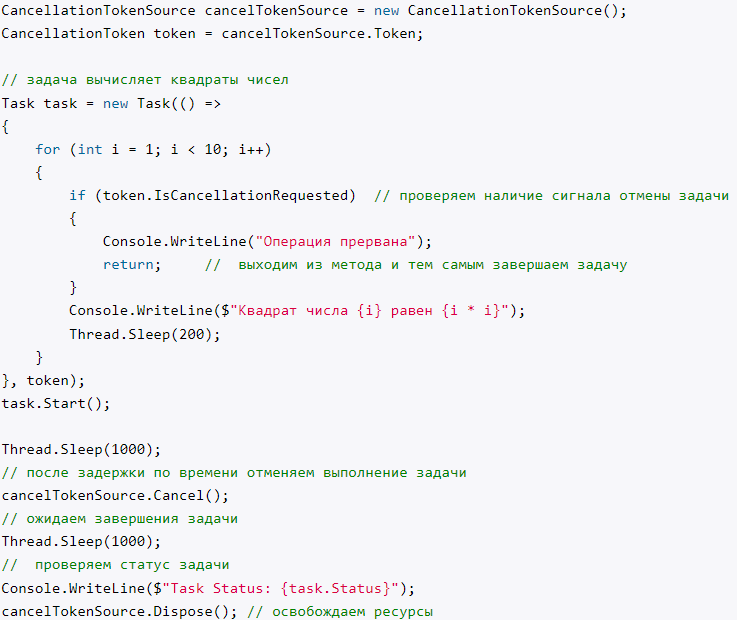


**Возвращение результатов из задач**

****

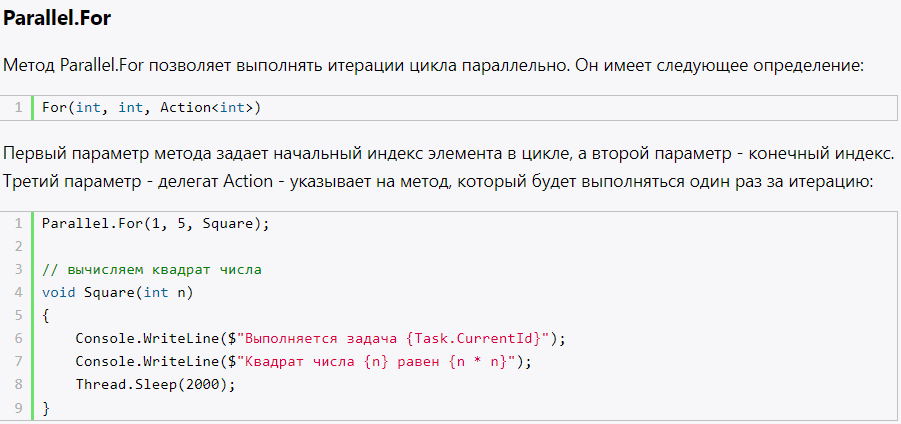
**Отмена выполнения задач**

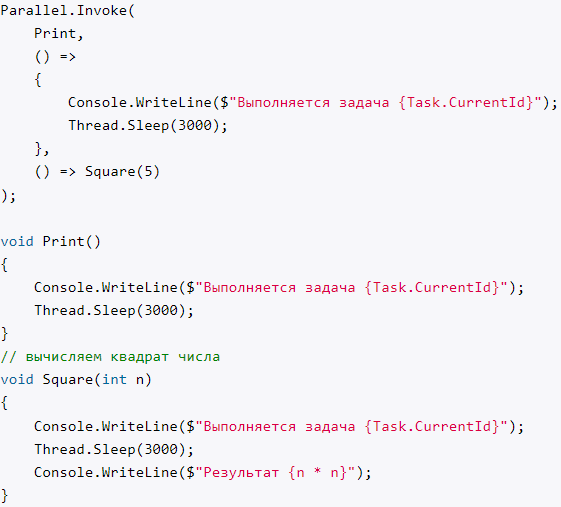
CancellationToken



# 58. Параллелизм при императивной обработке данных. Класс Parallel

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





# 59. Асинхронные методы. async и await

Асинхронность позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы и при этом более экономно использовать потоки

Асинхронный метод обладает следующими признаками:

В заголовке метода используется модификатор **async**

Метод содержит одно или несколько выражений **await**

# 60. Проектирование отношений. Агрегация, композиция и ассоциация

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. Например, игрок играет в определенной команде

Композиция определяет отношение HAS A, то есть отношение "имеет". Например, в класс автомобиля содержит объект класса электрического двигателя

Агрегация

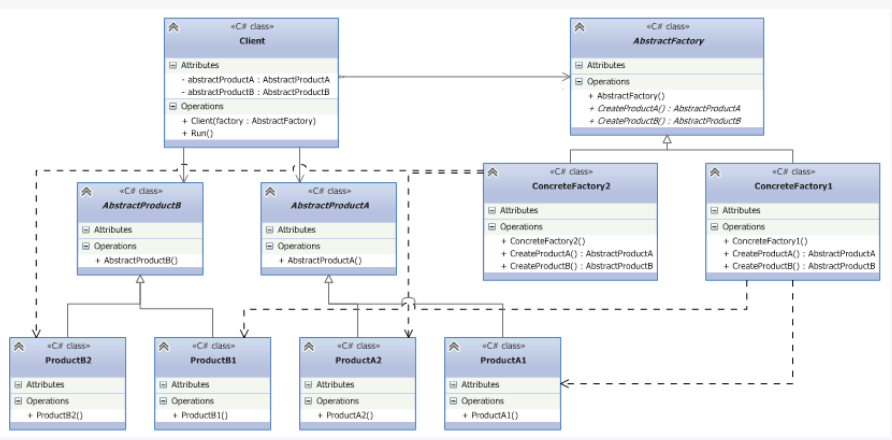
От композиции следует отличать агрегацию. Она также предполагает отношение HAS A, но реализуется она иначе:

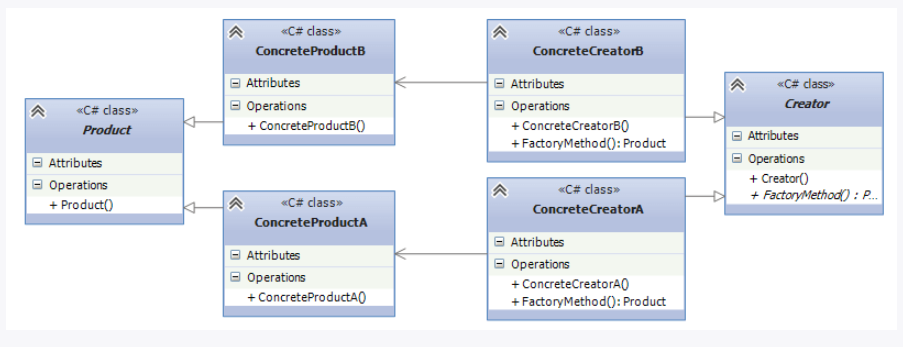
# 61. Антипаттерны проектирования. Рефакторинг. Методы рефакторинга.

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

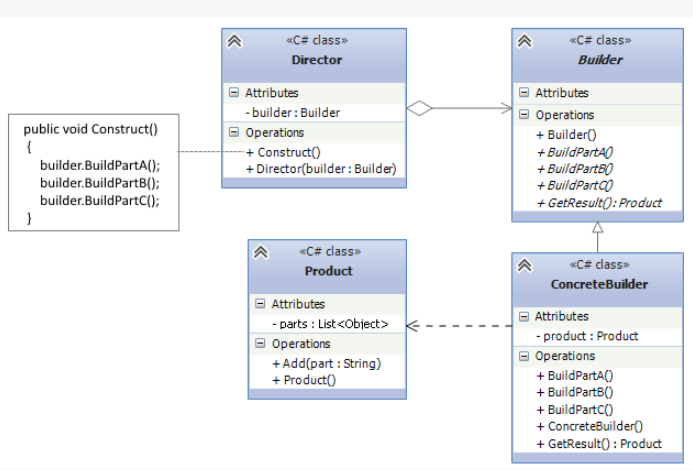
# 63. Чистый код. Требования к классам и объектам. Закон Деметры. DTO. Избыточный код.

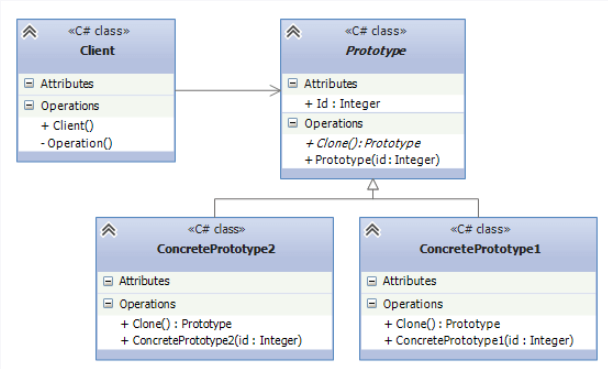
# 64. Паттерны проектирования. Классификация. Порождающие Abstarct Factory, Factory Method



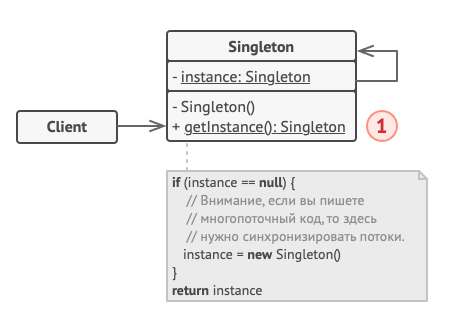


# 65. Порождающие паттерны проектирования :Builder, Object pool, Prototype.

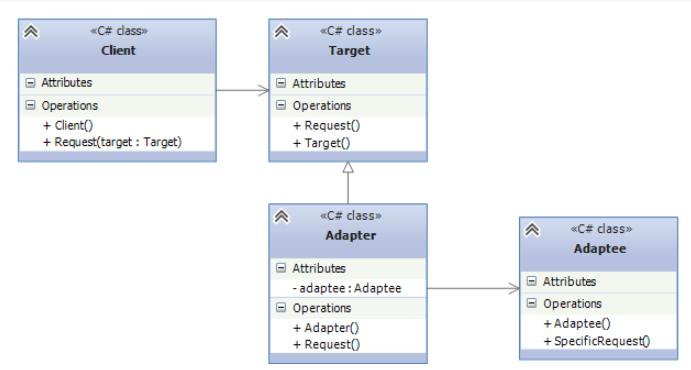


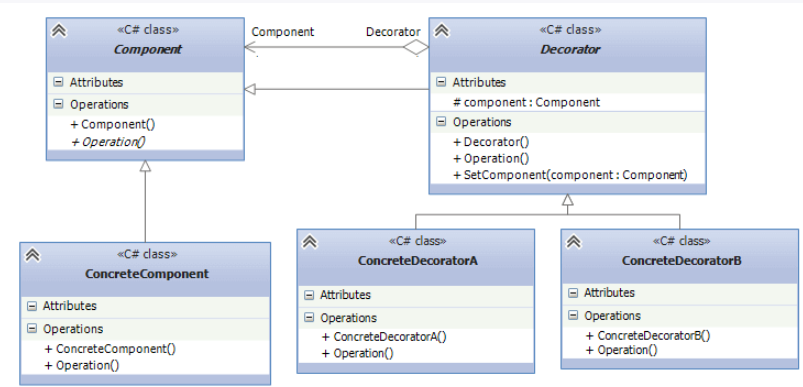


# 66. Порождающие паттерны проектирования: Singleton – 4 реализации, Lazy initialization,

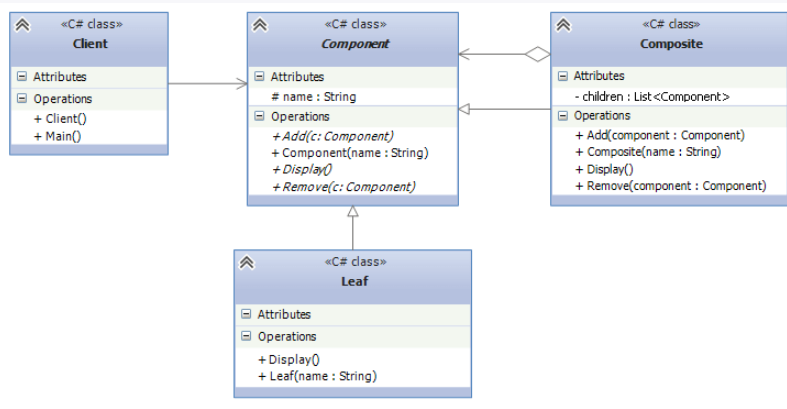


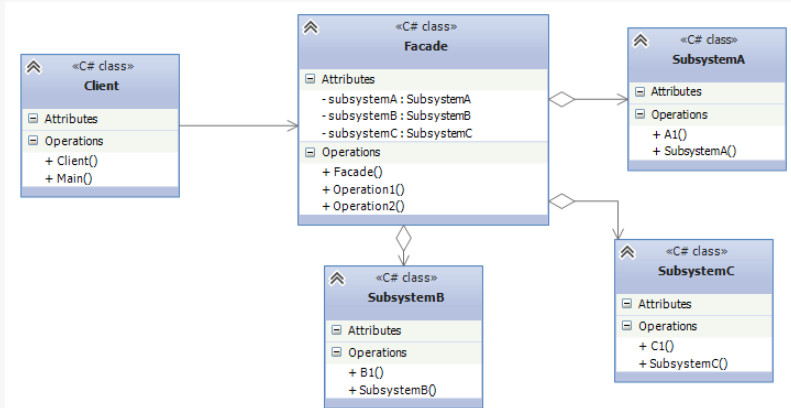
# 67. Структурные паттерны: Adapter, Decorator



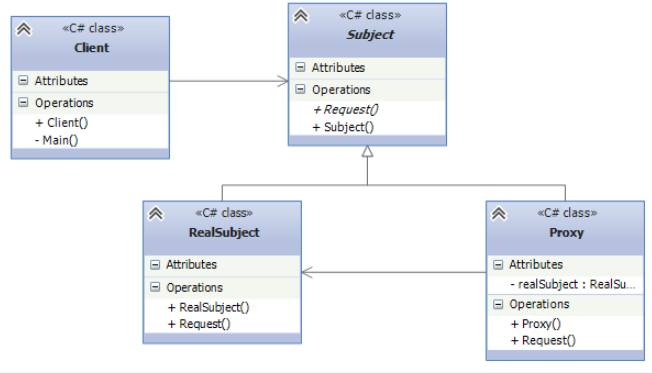


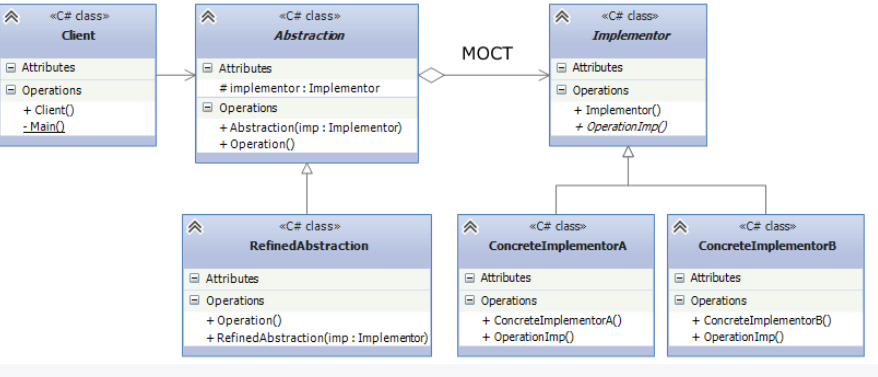
# 68. Структурные паттерны: Composite, Facade





# 69. Структурные паттерны: Proxy, Bridge





# 70. Паттерны поведения: Chain of Responsibility, Command

# 71. Паттерны поведения: Iterator, Mediator, Visitor

# 72. Паттерны поведения: Memento, Observer, Null Object, Strategy

# 73. Принципы проектирования SOLID.

* Single responsibility — принцип единственной ответственности
* Open-closed — принцип открытости / закрытости
* Liskov substitution — принцип подстановки Барбары Лисков
* Interface segregation — принцип разделения интерфейса
* Dependency inversion — принцип инверсии зависимостей
* **Принцип единственной обязанности / ответственности** (single responsibility principle / SRP) обозначает, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности. [Подробнее про SRP...](https://web-creator.ru/articles/solid_the_single_responsibility_principle)
* **Принцип открытости / закрытости** (open-closed principle / OCP) декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода. [Подробнее про OCP...](https://web-creator.ru/articles/solid_the_open_closed_principle)
* **Принцип подстановки Барбары Лисков** (Liskov substitution principle / LSP) в формулировке Роберта Мартина: «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом». [Подробнее про LSP...](https://web-creator.ru/articles/solid_the_liskov_substitution_principle)
* **Принцип разделения интерфейса** (interface segregation principle / ISP) в формулировке Роберта Мартина: «клиенты не должны зависеть от методов, которые они не используют». Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют. [Подробнее про ISP...](https://web-creator.ru/articles/solid_the_interface_segregation_principle)
* **Принцип инверсии зависимостей** (dependency inversion principle / DIP) — модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций.