Оглавление

[1.Системы контроля версий. Классификация. Git, основные возможности 2](#_Toc502351717)

[2.Состав. NET Framework. Структура среды выполнения CLR. 5](#_Toc502351718)

[3. Структура управляемого модуля - portable executable (PE). Понятие и исполнение сборки. CIL. 6](#_Toc502351719)

[4. CTS (Common Type System). Типы данных C#. Ссылочные и типы значений. 7](#_Toc502351720)

[5. Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, null-объединение 8](#_Toc502351721)

[6. Тип данных String: операции, литералы, пустые и нулевые строки, форматированный вывод. 11](#_Toc502351722)

[7. Неявная типизация – назначение и использование. 12](#_Toc502351723)

[8. Массивы C# одномерные, прямоугольные и ступенчатые. 13](#_Toc502351724)

[9. Понятие кортежей. Свойства, создание 14](#_Toc502351725)

[10. Принципы объектно-ориентированного программирования 15](#_Toc502351726)

[11. Класс. Элементы класса. Свойства и индексаторы. 17](#_Toc502351727)

[12. Класс. Константы. Поля только для чтения. Инициализаторы класса 20](#_Toc502351728)

[13. Спецификаторы доступа C#. Видимость типов. Доступ к членам типов. 21](#_Toc502351729)

[14.Класс. Конструкторы и их свойства. Деструкторы 23](#_Toc502351730)

[15.Класс и методы System.Object. 24](#_Toc502351731)

[16.Статические методы и статические конструкторы класса. 25](#_Toc502351732)

[17.Статические классы. Методы расширения и правила их определения. 26](#_Toc502351733)

[18. Анонимные типы. 28](#_Toc502351734)

[19. Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы. 28](#_Toc502351735)

[20. Перегрузка методов и операторов. Правила перегрузки операторов. 32](#_Toc502351736)

[21. Операции преобразования типа. Явная и неявная форма. Ограничения. 33](#_Toc502351737)

[22. Вложенные типы, объекты 35](#_Toc502351738)

[23. Правила наследования C#. 37](#_Toc502351739)

[24. Сокрытие имен при наследовании. Обращение к срытым членам 38](#_Toc502351740)

[25.Использование операций is и as 39](#_Toc502351741)

[26.Полиморфизм. Виртуальные методы, свойства и индексаторы. Правила переопределения. 42](#_Toc502351742)

[27.Понятие раннего и позднего связывания. 44](#_Toc502351743)

[28. Абстрактные классы и методы. Бесплодные классы. 47](#_Toc502351744)

[29. Структур в C#. 49](#_Toc502351745)

[30. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов. 50](#_Toc502351746)

[31. Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы. 51](#_Toc502351747)

[32. Ковариантность интерфейсов. Контравариантность интерфейсов 53](#_Toc502351748)

[33. Стандартные интерфейсы .NET. Назначение и применение. 55](#_Toc502351749)

[34. Исключительные ситуации. Генерация и обработка. 57](#_Toc502351750)

[35. Обобщения (generics). Свойства обобщений. Статические члены обощений. 59](#_Toc502351751)

[36. Концепция ограничений обобщений 61](#_Toc502351752)

[37. Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты. 63](#_Toc502351753)

[38. Анонимные функции. Лямбда-выражения. 65](#_Toc502351754)

[39. Обобщённые делегаты .NET. Action, Func, Predicate 66](#_Toc502351755)

[40. События. 67](#_Toc502351756)

[41. Стандартные коллекции .NET. Типы коллекций. 68](#_Toc502351757)

[42. IEnumerable и IEnumerator 69](#_Toc502351758)

[43. LINQ to Objects. Синтаксис. Грамматика выражений запросов. 70](#_Toc502351759)

[44. LINQ to Objects. Операции Where, Select, Take, OrderB, Join, GroupBy 71](#_Toc502351760)

[45. Рефлексия 72](#_Toc502351761)

[46. Классы для работы с файловой системой. 74](#_Toc502351762)

[47. Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы. 75](#_Toc502351763)

[48. Классы адаптеры потоков. 76](#_Toc502351764)

[49. Сериализация. Форматы сериализации. 78](#_Toc502351765)

[50. Сериализация контрактов данных. интерфейс ISerializable. 83](#_Toc502351766)

[51. Атрибуты. Создание собственного атрибута. 83](#_Toc502351767)

[52. Процесс. Домен приложений. Поток выполнения. 84](#_Toc502351768)

[53. Создание потоков , классы приоритетов. Состояния потоков 86](#_Toc502351769)

[54. Синхронизация потоков. Lock. Monitor. Мьютекс. Семафор 87](#_Toc502351770)

[55. Библиотека параллельных задач TPL. Класс Task. Состояние задачи. 90](#_Toc502351771)

[56. Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения. 90](#_Toc502351772)

[57. Параллелизм при императивной обработке данных. Класс Parallel 92](#_Toc502351773)

[58. Асинхронные методы. async и await 94](#_Toc502351774)

### 1.Системы контроля версий. Классификация. Git, основные возможности

### Система контроля версий — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии.

Системы кв бывают таких типов как:

* Локальные(Revision control System , Rcs)
* Централизованные(Centraliazed version control system)
* Распределенные(Distributed version control system, DVCS)

**Локальные** :

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельную директорию (возможно даже, директорию с отметкой по времени, если они достаточно сообразительны). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть, в какой директории вы находитесь, и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели.

Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий.

**Репозиторий** - хранилище (место) , где хранятся и поддерживаются какие-либо данные.

**Централизованные:**

Следующая серьёзная проблема, с которой сталкиваются люди, — это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища.

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать.

Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

**Распределенные:**

В РСКВ клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) — они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.

Более того, многие РСКВ могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

**Git** – хранит снимки состояний, а не изменений

Git – это распределённая система управления версиями файлов

Три состояния:

У Git’а есть три основных состояния, в которых могут находиться ваши файлы: зафиксированное (committed), изменённое (modified) и подготовленное (staged). \* Зафиксированный значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

**Ветвления** – отклонение от основной линии разработки, после которог работа перестает затрагивать основную линию



**Слияние**

****



**Распределенная разработка:**

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

.Net Framework – программная платформа, содержит следующие компоненты:

* CLR (Common Language Runtime) – общеязыковая среда исполнения, виртуальная машина на которой исполняются все приложения, работавшие в среде .Net.
* JIT –компилятор (Just in time)
* FCL( Fremework Class Library) – объектно-ориентированная библиотека классов, интерфейсов и системы типов(тип-значения)

.*NET Framework – решение следующих проблем:*

* Интеграция языков программирования
* CLS(common language specification) – общеязыковая спецификация, предназначенная для разработчиков компиляторов
* CTS(common type systems) – спецификация типов, которые должны поддерживаться всеми языками, ориентированными на CLR. Microsoft выпустил несколько компиляторов соответствующих этой спецификации: C++, C#, VB, .Net, Jscript

*Работает на многих платформах*

*Упрощенное повторное использование кода* - CLR позволяет типы, разработанные на одном языке, использовать в других языках.

*Автоматическое управление памятью* – CLR автоматически отслеживает использование ресурсов. Сборщик мусор.

*Проверка безопасности типов* – при работе в CLR практически исключена возможность записать(стереть) данные в область памяти, которая для этого не предназначена. Нет возможности передать управление в произвольную точку.

*Единый принцип обработки сбоев* – посредствам механизма исключений.

*Структура среды выполнения CLR*:



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

Файл PE состоит из нескольких заголовков и секций, которые указывают динамическому компоновщику, как отображать файл в память. Исполняемый образ состоит из нескольких различных областей (секций), каждая из которых требует различных прав доступа к памяти; таким образом, начало каждой секции должно быть выровнено по границе страницы. Например, обычно секция .text, которая содержит код программы, отображена как исполняемая/доступная только для чтения, а секция .data, содержащая глобальные переменные, отображена как неисполняемая/доступная для чтения и записи. Однако, чтобы не тратить впустую пространство на жёстком диске, различные секции на нём на границу страницы не выровнены. Часть работы динамического компоновщика состоит в том, чтобы отобразить каждую секцию в память отдельно и присвоить корректные права доступа получившимся областям согласно указаниям, содержащимся в заголовках.

IL – объектно-ориентированный машинный язык, не зависящий от процессора

1. Компиляция исходного кода в управляемые модули:



1. Выполнение кода в среде CLR – посредствам JIT – компилятора IL-код преобразуется в машинный код

Метаданные – это двоичный набор таблиц данных: типы и их члены.

Управляемый модуль – portable executable(PE)

* Заголовок PE 32(+)
* Заголовок CLR – точки входа и месторасположения ресурсов и метаданных
* Метаданные
* Код IL

Сборка(assembly) – это абстрактное понятие для логической группировки одного или нескольких управляемых модулей или файлов ресурсов, либо это дискретная единица многократно используемого кода внутри среды разработки



**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

*Развертывание сборки:*

1. *Закрытым – развертываются в базовом каталоге приложения в одном из его подкаталогов. Для сборки с нестрогим именем возможно лишь закрытое развертывание*
2. *Глобальное*

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

Способ определения и поведения типов:

* Состав членов: 0, поле, метод, свойство
* Правило видимости типов и доступа к членам типа
* Правила, управляющие наследованием, работой виртуальных методов, сроком жизни объектов
* Все типы должны быть производными (прямо или опосредованно) от типа System.Object

Манифест – набор таблиц метаданных – файлы, которые входят в сборку, обзедоступные экспортируемые типы, файлы ресурсов или данных.

*Типы данных С#:*

* *Типы – значения – структуры, перечисления и простые типы*

*Размещаются в стеке потока*

*Поля экземпляра размещаются в самой переменной*

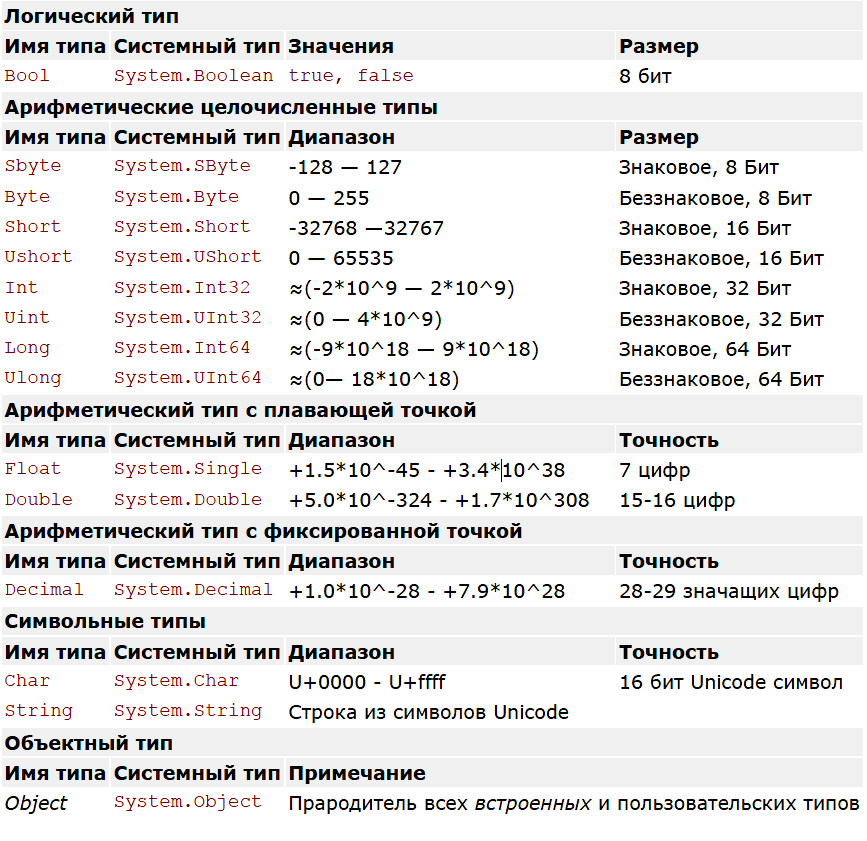
*Не обрабатываются сборщиком мусора*

* *Ссылочные типы – классы, строки, делегаты, массивы, интерфейсы*

*Определяются в куче*

**

Типы данных, которые поддерживаются компилятором напрямую, называются **примитивными или встроенными.**



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

class Account

{

    public object Id { get; set; }

    public int Sum { get; set; }

}

Account account1 = new Account { Sum = 5000 };

Account account2 = new Account { Sum = 4000 };

account1.Id = 2;

account2.Id = "4356";

int id1 = (int)account1.Id;

string id2 = (string)account2.Id;

Console.WriteLine(id1);

Console.WriteLine(id2);

Все вроде замечательно работает, но такое решение является не очень оптимальным. Дело в том, что в данном случае мы сталкиваемся с такими явлениями как **упаковка (boxing)** и **распаковка (unboxing)**.

Так, при присвоении свойству Id значения типа int, происходит упаковка этого значения в тип Object:

account1.Id = 2;        // упаковка в значения int в тип Object

Чтобы обратно получить данные в переменную типов int, необходимо выполнить распаковку:

int id1 = (int)account1.Id;     // Распаковка в тип int

Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа **System.Object** и сохраняет его в управляемой куче (хипе). Распаковка (unboxing), наоборот, предполагает получение указателя на исходный тип, преобразование объекта типа object к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.

Кроме того, существует другая проблема - проблема безопасности типов. Так, мы получим ошибку во время выполнения программы, если напишем следующим образом:

Account account2 = new Account { Sum = 4000 };

account2.Id = "4356";

int id2 = (int)account2.Id;     // Исключение InvalidCastException

Мы можем не знать, какий именно объект представляет Id, и при попытке получить число в данном случае мы столкнемся с исключением InvalidCastException.

**Назначение:**

* Позволяет использовать типы-значени в коллекциях9где элементы являются элементами типа объект
* Внутренний механизм, который обеспечивает возможность вызывать для типов-значени, подобных int, struct, методы объекта

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

object o = null;

string s = null;

Переменные значимых типов мы так задать не сможем, а если попробуем написать что-то вроде int x = null;, то получим ошибку.Однако в различных ситуациях бывает удобно, чтобы объекты числовых типов данных имели значение null, то есть были бы не определены. Стандартный пример - работа с базой данных, которая может содержать значения null. И мы можем заранее не знать, что мы получим из базы данных - какое-то определенное значение или же null. Для этого надо использовать знак вопроса ? после типа значений. Например:

int? z = null;

bool? enabled = null;

Но фактически запись ? является упрощенной формой использования структуры **System.Nullable<T>**. Параметр T в угловых скобках представляет универсальный параметр, вместо которого в конкретной задача уже подставляется конкретный тип данных. Следующие виды определения переменных будут эквивалентны:

int? z1 = 5;

bool? enabled1 = null;

Double?  d1 = 3.3;

Nullable<int> z2 = 5;

Nullable<bool> enabled2 = null;

Nullable<System.Double> d2 = 3.3;

Для всех типов Nullable определено два свойства: **Value**, которое представляет значение объекта, и **HasValue**, которое возвращает true, если объект Nullable хранит некоторое значение.

Причем свойство Value хранит объект того типа, которым типизируется Nullable:

class Program

{

    static void Main(string[] args)

    {

        int? x = 7;

        Console.WriteLine(x.Value); // 7

        Nullable<State> state = new State() { Name = "Narnia" };

        Console.WriteLine(state.Value.Name);    // Narnia

        Console.ReadLine();

    }

}

struct State

{

    public string Name { get; set; }

}

Однако если мы попробуем получить значение переменной, которая рана null, то мы столкнемся с ошибкой:

**Преобразование типов Nullable**

явное преобразование от T? к T

int? x1 = null;

if(x1.HasValue)

{

    int x2 = (int)x1;

    Console.WriteLine(x2);

}

неявное преобразование от T к T?

int x1 = 4;

int? x2 = x1;

Console.WriteLine(x2);

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. Посмотрим на примере:

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

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

В языке C# строковые значения представляет тип string, а вся функциональность работы с данным типом сосредоточена в классе **System.String**. Собственно string является псевдонимом для класса System.String.

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

string s1 = "hello";

string s2 = null;

string s3 = new String('a', 6); // результатом будет строка "aaaaaa"

string s4 = new String(new char[]{'w', 'o', 'r', 'l', 'd'});

Конструктор String имеет различное число версий. Так, вызов конструктора new String('a', 6) создаст строку "aaaaaa". И так как строка представляет ссылочный тип, то может хранить значение null.

Так как строка хранит коллекцию символов, в ней определен индексатор для доступа к этим символам:

public char this[int index] {get;}

Применяя индексатор, мы можем обратиться к строке как к массиву символов и получить по индексу любой из ее символов:

string s1 = "hello";

char ch1 = s1[1]; // символ 'e'

Console.WriteLine(ch1);

Console.WriteLine(s1.Length);

Операции:

* Конкатенация

string s4 = String.Concat(s3, "!!!");

* Сравнение строк

int result = String.Compare(s1, s2);

* Поиск в строке

char ch = 'o';

int indexOfChar = s1.IndexOf(ch); // равно 4

* Разделение строк

string[] words = text.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

* Обрезка строки

text = text.Insert(8, subString);

* Удаление строк

text = text.Remove(ind);

* Замена

text = text.Replace("хороший", "плохой");

* Смена регистра

Для приведения строки к верхнему и нижнему регистру используются соответственно функции **ToUpper()** и **ToLower()**

Форматированные ввод\вывод:

class Program

{

    static void Main(string[] args)

    {

        Person person = new Person { Name = "Tom", Age = 23 };

        Console.WriteLine("Имя: {0}  Возраст: {1}", person.Name, person.Age);

        Console.Read();

    }

}

class Person

{

    public string Name { get; set; }

    public int Age { get; set; }

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

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.

### 8. 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:

int[] numbers;

int[] nums = new int[4];

Здесь вначале мы объявили массив nums, который будет хранить данные типа int. Далее используя операцию new, мы выделили память для 4 элементов массива: new int[4]. Число 4 еще называется **длиной массива**. При таком определении все элементы получают значение по умолчанию, которое предусмотренно для их типа. Для типа int значение по умолчанию - 0.

Также мы сразу можем указать значения для этих элементов:

int[] nums2 = new int[4] { 1, 2, 3, 5 };

int[] nums3 = new int[] { 1, 2, 3, 5 };

int[] nums4 = new[] { 1, 2, 3, 5 };

int[] nums5 = { 1, 2, 3, 5 };

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);

}

Многомерные массивы - зубчатый:

int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };

int[,] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };

Массив массивов - ступенчатые:

int[][] nums = new int[3][];

nums[0] = new int[2] { 1, 2 };          // выделяем память для первого подмассива

nums[1] = new int[3] { 1, 2, 3 };       // выделяем память для второго подмассива

nums[2] = new int[5] { 1, 2, 3, 4, 5 }; // выделяем память для третьего подмассива

### 9. Понятие кортежей. Свойства, создание

Кортежи предоставляют удобный способ для работы с набором значений, который был добавлен в версии C# 7.0.

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();

}

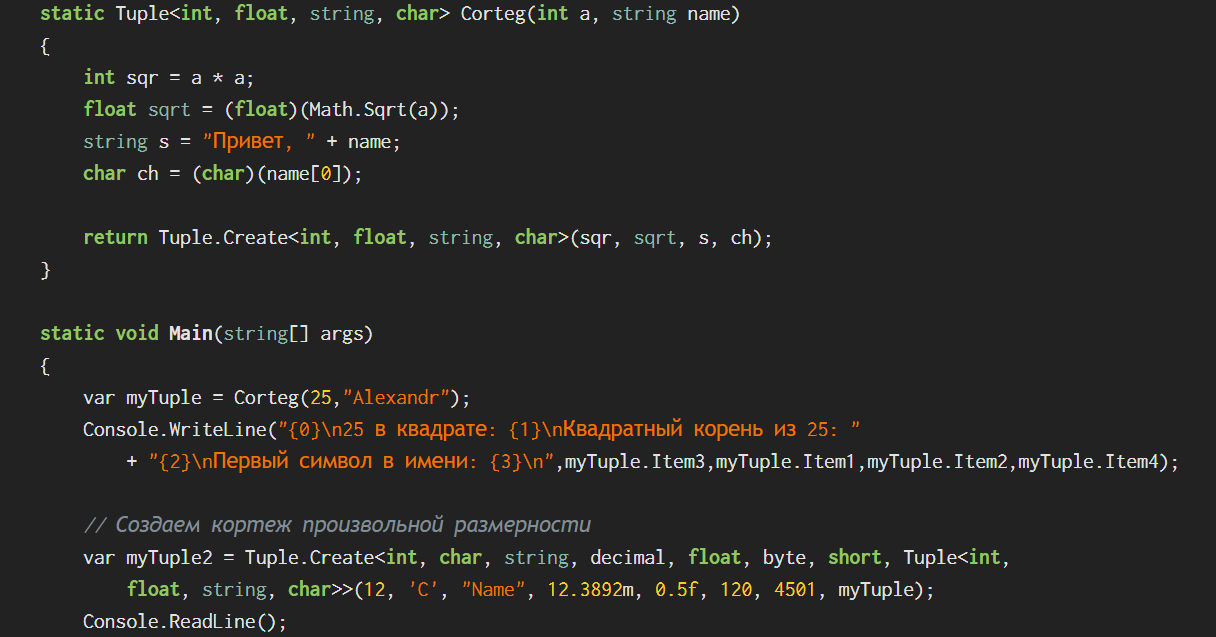
private static (int, int) GetValues()

{

    var result = (1, 3);

    return result;

}



1. Принципы объектно-ориентированного программирования**.**

Все основанные на объектах языки (C#, Java, С++, Smalltalk, Visual Basic и т.п.) должны отвечать трем основным принципам объектно-ориентированного программирования (ООП), которые перечислены ниже:

***Инкапсуляция***

Как данный язык скрывает детали внутренней реализации объектов и предохраняет целостность данных?

***Наследование***

Как данный язык стимулирует многократное использование кода?

***Полиморфизм***

Как данный язык позволяет трактовать связанные объекты сходным образом?

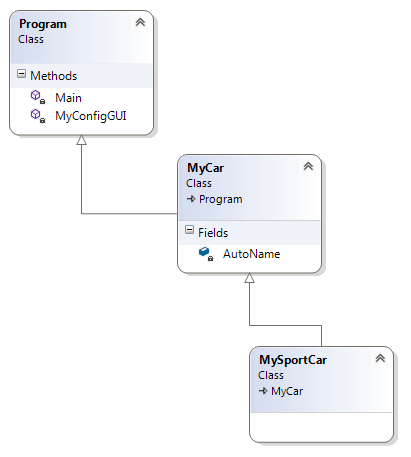
Прежде чем погрузиться в синтаксические детали реализации каждого принципа, важно понять базовую роль каждого из них.

Роль инкапсуляции

Инкапсуляция — это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных. В объектно-ориентированном языке данные и код могут быть объединены в совершенно автономный черный ящик. Внутри такого ящика находятся все необходимые данные и код. Когда код и данные связываются вместе подобным образом, создается объект. Иными словами, **объект** — это элемент, поддерживающий инкапсуляцию.

**Роль наследования**

Следующий принцип ООП — **наследование** — касается способности языка позволять строить новые определения классов на основе определений существующих классов. По сути, наследование позволяет расширять поведение базового (или родительского) класса, наследуя основную функциональность в производном подклассе (также именуемом дочерним классом):

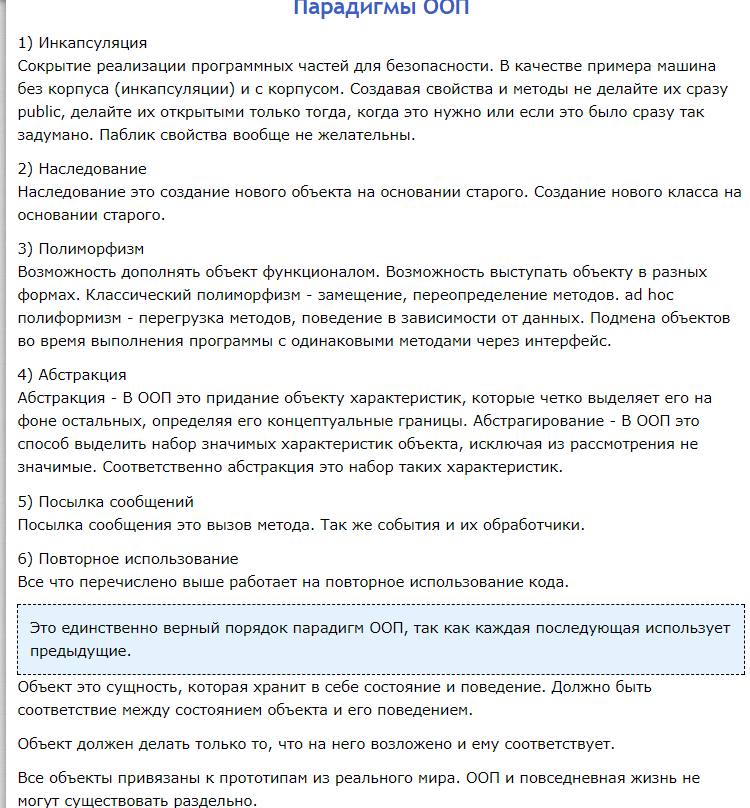


Т.е. наследование представляет собой процесс, в ходе которого один объект приобретает свойства другого объекта. Это очень важный процесс, поскольку он обеспечивает принцип иерархической классификации. Если вдуматься, то большая часть знаний поддается систематизации благодаря иерархической классификации по нисходящей.

Роль полиморфизма

Последний принцип ООП — **полиморфизм**. Он обозначает способность языка трактовать связанные объекты в сходной манере. В частности, этот принцип ООП позволяет базовому классу определять набор членов (формально называемый полиморфным интерфейсом), которые доступны всем наследникам. Полиморфный интерфейс класса конструируется с использованием любого количества виртуальных или абстрактных членов.

По сути, виртуальный член — это член базового класса, определяющий реализацию по умолчанию, которая может быть изменена (или, говоря более формально, переопределена) в производном классе. В отличие от него, абстрактный метод — это член базового класса, который не предусматривает реализации по умолчанию, а предлагает только сигнатуру. Когда класс наследуется от базового класса, определяющего абстрактный метод, этот метод обязательно должен быть переопределен в производном классе. В любом случае, когда производные классы переопределяют члены, определенные в базовом классе, они по существу переопределяют свою реакцию на один и тот же запрос.



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

**Индексаторы** позволяют индексировать объекты и использовать их как массивы. Фактически индексаторы позволяют нам создавать специальные хранилища объектов или коллекции. По форме они напоминают свойства со стандартными методами get и set, которые возвращают и присваивают значение.Посмотрим на примере. Допустим, у нас есть класс Book, представляющий книгу, и класс Library, который представляет библиотеку и используется для хранения набора книг. Используем индексаторы для определения класса Library:

class Program

{

    static void Main(string[] args)

    {

        Library library = new Library();

        Console.WriteLine(library[0].Name);

        library[0] = new Book("Преступление и наказание");

        Console.WriteLine(library[0].Name);

        Console.ReadLine();

    }

}

class Book

{

    public Book(string name)

    {

        this.Name=name;

    }

    public string Name { get; set; }

}

class Library

{

    Book[] books;

    public Library()

    {

        books = new Book[] { new Book("Отцы и дети"), new Book("Война и мир"),

                new Book("Евгений Онегин") };

    }

    public int Length

    {

        get { return books.Length; }

    }

    public Book this[int index]

    {

        get

        {

            return books[index];

        }

        set

        {

            books[index] = value;

        }

    }

}

Описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями. Например, определим в классе Person поля и метод:

class Person

    {

        public string name; // имя

        public int age;     // возраст

        public void GetInfo()

        {

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

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            Person tom;

            Console.Read();

        }

    }

Кроме обычных методов в классах используются также и специальные методы, которые называются **конструкторами**. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор без параметров.Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения).

***Поля (field) -*** Это любые переменные, ассоциированные с классом.

Константы могут быть ассоциированы с классом тем же способом, что и переменные. Константа объявляется с помощью ключевого слова **const**. Если она объявлена как public, то в этом случае становится доступной извне класса.

***События-***Это члены класса, позволяющие объекту уведомлять вызывающий код о том, что случилось нечто достойное упоминания, например, изменение свойства класса либо некоторое взаимодействие с пользователем. Клиент может иметь код, известный как обработчик событий, реагирующий на них.

***Методы (method)-*** Это функции, ассоциированные с определенным классом. Как и данные-члены, по умолчанию они являются членами экземпляра. Они могут быть объявлены статическими с помощью модификатора static.

***Свойства (property) -***Это наборы функций, которые могут быть доступны клиенту таким же способом, как общедоступные поля класса. В C# предусмотрен специальный синтаксис для реализации чтения и записи свойств для классов, поэтому писать собственные методы с именами, начинающимися на Set и Get, не понадобится.

***Конструкторы (constructor)***

***Индексаторы (indexer)-***Позволяют индексировать объекты таким же способом, как массив или коллекцию.

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

Полями класса называются обычные переменные уровня класса. Мы уже ранее рассматривали переменные - их объявление и инициализацию. Однако некоторые моменты мы еще не затрагивали, например, константы и поля для чтения.

Константы:Особенностью констант является то, что их значение можно установить только один раз. Например, если у нас в программе есть некоторые переменные, которые не должны изменять значение (к примеру, число PI, число e и т.д.), мы можем объявить их константами. Для этого используется ключевое слово **const**:

|  |  |
| --- | --- |
|  | const double PI = 3.14;  сonst double E = 2.71; |

При использовании констант надо помнить, что объявить мы их можем только один раз и что к моменту компиляции они должны быть определены.

|  |  |
| --- | --- |
|  | class MathLib  {      public const double PI=3.141;      public const double E = 2.81;      public const double K;      public MathLib()      {        K = 2.5; // ошибка - константа должна быть определена до компиляции    }}    class Program  {    static void Main(string[] args)      {        MathLib.E=3.8; // константу нельзя установить несколько з      }  } |

Также обратите внимание на синтаксис обращения к константе. Так как это статическое поле, нам необязательно создавать объект класса с помощью конструктора. Мы можем обратиться к ней, используя имя класса.

**Поля для чтения**

Поля для чтения можно инициализировать:

* при их объявлении
* на уровне класса
* инициилизировать и изменять в конструкторе.

Инициализировать или изменять их значение в других местах нельзя, можно только считывать их значение.

Поле для чтения объявляется с ключевым словом **readonly**:

|  |  |
| --- | --- |
|  | class MathLib  {    public readonly double K = 23;  // можно так инициализировать        public MathLib(double \_k)      {          K = \_k; // поле для чтения может быть инициализировано или изменено в конструкторе после компиляции    }      public void ChangeField()      {          // так нельзя          //K = 34;      }}    class Program  {      static void Main(string[] args)      {          MathLib mathLib = new MathLib(3.8);          Console.WriteLine(mathLib.K); // 3.8            //mathLib.K = 7.6; // поле для чтения нельзя установить вне своего класса          Console.ReadLine();        }} |

**Инициализаторы объектов**

Для инициализации объектов классов можно применять **инициализаторы**. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

|  |  |
| --- | --- |
|  | Person tom = new Person { name = "Tom", age=31 };  tom.GetInfo();          // Имя: Tom  Возраст: 31 |

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.

Модификаторы доступа позволяют задать допустимую область видимости для членов класса. То есть контекст, в котором можно употреблять данную переменную или метод.

В C# применяются следующие модификаторы доступа:

* **public**: публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.
* **private**: закрытый класс или член класса. Представляет полную противоположность модификатору public. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.
* **protected**: такой член класса доступен из любого места в текущем классе или в производных классах.
* **internal**: класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором public).
* **protected internal**: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.

Объявление полей класса без модификатора доступа равнозначно их объявлению с модификатором **private**. Классы, объявленные без модификатора, по умолчанию имеют доступ **internal**.

Каждая переменная доступна в рамках определенного контекста или области **видимость**. Вне этого контекста переменная уже не существует.

Существуют различные контексты:

* Контекст класса. Переменные, определенные на уровне класса, доступны в любом методе этого класса
* Контекст метода. Переменные, определенные на уровне метода, являются локальными и доступны только в рамках данного метода. В других методах они недоступны
* Контекст блока кода. Переменные, определенные на уровне блока кода, также являются локальными и доступны только в рамках данного блока. Вне своего блока кода они не доступны.

**Организация закрытого и открытого доступа:**

* Члены, используемые только в классе, должны быть закрытыми
* Если изменение члена приводит к последствиям, распространяющимся за пределы области действия самого члена, т.е. оказывает влияние на другие аспекты объекта, то этот член должен быть закрытым, а доступ к нему — контролируемым
* Члены, способные нанести вред объекту, если они используются неправильно, должны быть закрытыми. Доступ к этим членам следует организовать с помощью открытых методов, исключающих неправильное их использование
* Методы, получающие и устанавливающие значения закрытых данных, должны быть открытыми
* Переменные экземпляра допускается делать открытыми лишь в том случае, если нет никаких оснований для того, чтобы они были закрытыми

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

Конструктор инициализирует объект при его создании. У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу. Но у конструкторов нет возвращаемого типа, указываемого явно.

Как правило, конструктор используется для задания первоначальных значений переменных экземпляра, определенных в классе, или же для выполнения любых других установочных процедур, которые требуются для создания полностью сформированного объекта. Кроме того, доступ обычно представляет собой модификатор доступа типа public, поскольку конструкторы зачастую вызываются в классе. А список\_параметров может быть как пустым, так и состоящим из одного или более указываемых параметров.

Конструктор также может принимать один или несколько параметров. В конструктор параметры вводятся таким же образом, как и в метод. Для этого достаточно объявить их в скобках после имени конструктора.

При использовании оператора **new** свободная память для создаваемых объектов динамически распределяется из доступной буферной области [оперативной памяти](https://professorweb.ru/my/glossariy/glossy/files/101.php#o2). Разумеется, оперативная память не бесконечна, и поэтому свободно доступная память рано или поздно исчерпывается. Это может привести к неудачному выполнению оператора new из-за нехватки свободной памяти для создания требуемого объекта.

Именно по этой причине одной из главных функций любой схемы динамического распределения памяти является освобождение памяти от неиспользуемых объектов, чтобы сделать ее доступной для последующего перераспределения. Во многих языках программирования освобождение распределенной ранее памяти осуществляется вручную. Например, в С++ для этой цели служит оператор delete. Но в C# применяется другой, более надежный подход: "сборка мусора".

Если ссылки на объект отсутствуют, то такой объект считается ненужным, и занимаемая им память в итоге освобождается и накапливается. Эта утилизированная память может быть затем распределена для других объектов.

"Сборка мусора" происходит лишь время от времени по ходу выполнения программы. Она не состоится только потому, что существует один или более объектов, которые больше не используются. Следовательно, нельзя заранее знать или предположить, когда именно произойдет "сборка мусора".

**Деструкторы**

В языке C# имеется возможность определить метод, который будет вызываться непосредственно перед окончательным уничтожением объекта системой "сборки мусора". Такой метод называется ***деструктором*** и может использоваться в ряде особых случаев, чтобы гарантировать четкое окончание срока действия объекта. Например, деструктор может быть использован для гарантированного освобождения системного ресурса, задействованного освобождаемым объектом.

В силу того что порядок вызова деструкторов не определен точно, их не следует применять для выполнения действий, которые должны происходить в определенный момент выполнения программы. В то же время имеется возможность запрашивать "сборку мусора". Тем не менее инициализация "сборки мусора" вручную в большинстве случаев не рекомендуется, поскольку это может привести к снижению эффективности программы. Кроме того, у системы "сборки мусора" имеются свои особенности — даже если запросить "сборку мусора" явным образом, все равно нельзя заранее знать, когда именно будет утилизирован конкретный объект.

### 15.Класс и методы System.Object.

Все остальные классы в .NET, даже те, которые мы сами создаем, а также базовые типы, такие как **System.Int32**, являются неявно производными от класса Object. Даже если мы не указываем класс Object в качестве базового, по умолчанию неявно класс Object все равно стоит на вершине иерархии наследования. Поэтому все типы и классы могут реализовать те методы, которые определены в классе System.Object. Рассмотрим эти методы.

* **ToString**

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

Для классов же этот метод выводит полное название класса с указанием пространства имен, в котором определен этот класс.

int i = 5;

Console.WriteLine(i.ToString()); // выведет число 5

* **GetHashCode**

Метод **GetHashCode** позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты. Можно определять самые разные алгоритмы генерации подобного числа или взять реализаци базового типа:

class Person

{

    public string Name { get; set; }

    public override int GetHashCode()

    {

        return Name.GetHashCode();

    }

}

* **GetType**

Метод GetType позволяет получить тип данного объекта:

Person person = new Person { Name = "Tom" };

Console.WriteLine(person.GetType());    // Person

С помощью ключевого слова **typeof** мы получаем тип класса и сравниваем его с типом объекта. И если этот объект представляет тип Client, то выполняем определенные действия.

object person = new Person { Name = "Tom" };

if (person.GetType() == typeof(Person))

    Console.WriteLine("Это реально класс Person");

Причем поскольку класс Object является базовым типом для всех классов, то мы можем переменной типа object присвоить объект любого типа. Однако для этого переменной метод GetType все равно вернет тот тип, на объект которого ссылается переменная. То есть в данном случае объект типа Person.

В отличие от методов ToString, Equals, GetHashCode метод GetType не переопределяется.

* **Метод Equals**

Метод Equals позволяет сравнить два объекта на равенство:

Person person1 = new Person { Name = "Tom" };

Person person2 = new Person { Name = "Bob" };

Person person3 = new Person { Name = "Tom" };

bool p1Ep2 = person1.Equals(person2);   // false

bool p1Ep3 = person1.Equals(person3);   // true

Метод Equals принимает в качестве параметр объект любого типа, который мы затем приводим к текущему, если они являются объектами одного класса. Затем сравниваем по именам. Если имена равны, возвращаем true, что будет говорить, что объекты равны. Однако при необходимости реализацию метода можно сделать более сложной, например, сравнивать по нескольким свойствам при их наличии.

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

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

Статические члены класса являются общими для всех объектов этого класса, поэтому к ним надо обращаться по имени класса:

Account.MinSum = 560;

decimal result = Account.GetSum(1000, 10, 5);

При использовании статических членов класса нам необязательно создавать экземпляр класса, а мы можем обратиться к ним напрямую. На уровне памяти для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса.

Сл

едует учитывать, что статические методы могут обращаться только статическим членам класса. Обращаться к нестатическим методам, полям, свойствам внутри статического метода мы не можем.

**Статический конструктор**

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы выполняются при самом первом создании объекта данного класса или первом обращении к его статическим членам (если таковые имеются).

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы имеют следующие отличительные черты:

* Статические конструкторы не должны иметь модификатор доступа и не принимают параметров
* 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
* 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)

Статические конструкторы обычно используются для инициализации статических данных, либо же выполняют действия, которые требуется выполнить только один раз

class User

{

    static User()

    {

        Console.WriteLine("Создан первый пользователь");

    }

}

class Program

{

    static void Main(string[] args)

    {

        User user1 = new User(); // здесь сработает статический конструктор

        User user2 = new User();

        Console.Read();

    }

}

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

**Статические классы**

Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы. Например, если бы класс Account имел бы только статические переменные, свойства и методы, то его можно было бы объявить как статический.

static class Account

{

    private static decimal minSum = 100; // минимальная допустимая сумма для всех счетов

    public static decimal MinSum

    {

        get { return minSum; }

        set { if(value>0) minSum = value; }

    }

    // подсчет суммы на счете через определенный период по определенной ставке

    public static decimal GetSum(decimal sum, decimal rate, int period)

    {

        decimal result = sum;

        for (int i = 1; i <= period; i++)

            result = result + result \* rate / 100;

        return result;

    }

}

**Методы расширения** (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем.

class Program

{

    static void Main(string[] args)

    {

        string s = "Привет мир";

        char c = 'и';

        int i = s.WordCount(c);

        Console.WriteLine(i);

        Console.ReadLine();

    }

}

public static class StringExtension

{

    public static int WordCount(this string str, char c)

    {

        int counter = 0;

        for (int i = 0; i<str.Length; i++)

        {

            if (str[i] == c)

                counter++;

        }

        return counter;

    }

}

Для того, чтобы создать метод расширения, вначале надо создать статический класс, который и будет содержать этот метод. В данном случае это класс StringExtension. Затем объявляем статический метод. Суть нашего метода расширения - подсчет количества определенных символов в строке.

Собственно метод расширения - это обычный статический метод, который в качестве первого параметра всегда принимает такую конструкцию: this имя\_типа название\_параметра, то есть в нашем случае this string str. Так как наш метод будет относиться к типу string, то мы и используем данный тип.

Затем у всех строк мы можем вызвать данный метод: int i = s.WordCount(c);. Причем нам уже не надо указывать первый параметр. Значения для остальных параметров передаются в обычном порядке.

Применение методов расширения очень удобно, но при этом надо помнить, что метод расширения никогда не будет вызван, если он имеет ту же сигнатуру, что и метод, изначально определенный в типе.

Также следует учитывать, что методы расширения действуют на уровне пространства имен. То есть, если добавить в проект другое пространство имен, то метод не будет применяться к строкам, и в этом случае надо будет подключить пространство имен метода через директиву using.

### 18. Анонимные типы.

Анонимные типы позволяют создать объект с некоторым набором свойств без определения класса. Анонимный тип определяется с помощью ключевого слова **var** и инициализатора объектов:

var user = new { Name = "Tom", Age = 34 };

Console.WriteLine(user.Name);

В данном случае user - это объект анонимного типа, у которого определены два свойства Name и Age. И мы также можем использовать его свойства, как и у обычных объектов классов. Однако тут есть ограничение - свойства анонимных типов доступны только для чтения.

При этом во время компиляции компилятор сам будет создавать для него имя типа и использовать это имя при обращении к объекту. Нередко анонимные типы имеют имя наподобие "<>f\_\_AnonymousType0'2".

Для исполняющей среды CLR анонимные типы будут также, как и классы, представлять ссылочный тип

Зачем нужны анонимные типы? Иногда возникает задача использовать один тип в одном узком контексте или даже один раз. Создание класса для подобного типа может быть избыточным. Если нам захочется добавить свойство, то мы сразу же на месте анонимного объекта это можем сделать. В случае с классом придется изменять еще и класс, который может больше нигде не использоваться. Типичная ситуация - получение результата выборки из базы данных: объекты используются только для получения выборки, часто больше нигде не используются, и классы для них создавать было бы излишне. А вот анонимный объект прекрасно подходит для временного хранения выборки.

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

**Параметры** представляют собой переменные, которые определяются в сигнатуре метода и создаются при его вызове.

Существует два способа передачи параметров в метод в языке C#: *по значению* и *по ссылке*.

Наиболее простой способ передачи параметров представляет передача по значению:

static int Sum(int x, int y)

{

    return x + y;

}

И при вызове этого метода в программе нам обязательно надо передать на место параметров значения, которые соответствуют типу параметра. Данные значения еще называют **аргументами**

static void Main(string[] args)

{

    int x = 10;

    int z = Sum(x, 15);

    Console.WriteLine(z);

    Console.ReadLine();

}

**Передача параметров по ссылке и модификатор ref**

При передаче параметров по ссылке перед параметрами используется модификатор **ref**:

В чем отличие двух способов передачи параметров? При передаче по значению метод получает не саму переменную, а ее копию. А при передаче параметра по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место.

Обратите внимание, что модификатор ref указывается, как при объявлении метода, так и при его вызове в методе Main.

static void Main(string[] args)

{

    //Начальные значения переменных a и b

    int a = 5;

    int b = 6;

    Console.WriteLine("Начальное значение переменной a  = {0}", a);

    //Передача переменных по значению

    //После выполнения этого кода по-прежнему a = 5, так как мы передали лишь ее копию

    AdditionVal(a, b);

    Console.WriteLine("Переменная a после передачи по значению равна = {0}", a);

    //Передача переменных по ссылке

    //После выполнения этого кода a = 11, так как мы передали саму переменную

    AdditionRef(ref a, b);

    Console.WriteLine("Переменная a после передачи ссылке по значению равна = {0}", a);

    Console.ReadLine();

}

// передача по ссылке

static void AdditionRef(ref int x, int y)

{

    x = x + y;

    Console.WriteLine("x + y = {0}", x);

}

// передача по значению

static void AdditionVal(int x, int y)

{

    x = x + y;

    Console.WriteLine("x + y = {0}", x);

}

**Модификатор out**

Выше мы использовали входные параметры. Но параметры могут быть также выходными. Чтобы сделать параметр выходным, перед ним ставится модификатор out:

static void Main(string[] args)

{

    int x = 10;

    int area;

    int perimetr;

    GetData(x, 15, out area, out perimetr);

    Console.WriteLine("Площадь : " + area);

    Console.WriteLine("Периметр : " + perimetr);

    Console.ReadLine();

}

static void GetData(int x, int y, out int area, out int perim)

{

    area= x \* y;

    perim= (x + y)\*2;

}

Причем, как и в случае с ref ключевое слово out используется как при определении метода, так и при его вызове.

Также обратите внимание, что методы, использующие такие параметры, обязательно должны присваивать им определенное значение.

По сути, как и в случае с ключевым словом **ref**, ключевое слово **out** применяется для для передачи аргументов по ссылке. Однако в отличие от ref для переменных, которые передаются с ключевым словам out, не требуется инициализация. И кроме того, вызываемый метод должен обязательно присвоить им значение.

**Необязательные параметры**

C# позволяет использовать необязательные параметры. Для таких параметров нам необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

static int OptionalParam(int x, int y, int z=5, int s=4)

{

    return x + y + z + s;

}

Так как последние два параметра объявлены как необязательные, то мы можем один из них или оба опустить:

static void Main(string[] args)

{

    OptionalParam(2, 3);

    OptionalParam(2,3,10);

    Console.ReadLine();

}

средство называется **необязательными аргументами** и позволяет определить используемое по умолчанию значение для параметра метода. Данное значение будет использоваться по умолчанию в том случае, если для параметра не указан соответствующий аргумент при вызове метода. Следовательно, указывать аргумент для такого параметра не обязательно. Необязательные аргументы позволяют упростить вызов методов, где к некоторым параметрам применяются аргументы, выбираемые по умолчанию.

Следует иметь в виду, что все необязательные аргументы должны непременно указываться справа от обязательных. Помимо методов, необязательные аргументы можно применять в конструкторах, индексаторах и делегатах.

**Именованные аргументы**

Как известно, при передаче аргументов методу порядок их следования, как правило, должен совпадать с тем порядком, в котором параметры определены в самом методе. Иными словами, значение аргумента присваивается параметру по его позиции в списке аргументов.

Данное ограничение призваны преодолеть именованные аргументы. Именованный аргумент позволяет указать имя того параметра, которому присваивается его значение. И в этом случае порядок следования аргументов уже не имеет никакого значения. Таким образом, именованные аргументы в какой-то степени похожи на упоминавшиеся ранее инициализаторы объектов, хотя и отличаются от них своим синтаксисом. Для указания аргумента по имени служит следующая форма синтаксиса:

*имя\_параметра : значение*

**class** **Program**

{

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

{

**return** a + b + c;

}

**static** **void** **Main**()

{

*// Использование именованных аргументов*

*// при вызове метода*

**int** sum1 = mySum(a: 3, b: 10);

Console.WriteLine("Sum1 = " + sum1);

Console.ReadLine();

}

}

Во всех предыдущих примерах мы использовали постоянное число параметров. Но, используя ключевое слово **params**, мы можем передавать неопределенное количество параметров:

static void Addition(params int[] integers)

{

    int result = 0;

    for (int i = 0; i < integers.Length; i++)

    {

        result += integers[i];

    }

    Console.WriteLine(result);

}

Причем, как видно из примера, на место параметра с модификатором params мы можем передать как отдельные значения, так и массив значений, либо вообще не передавать параметры.

Однако данный способ имеет ограничения: после параметра с модификатором params мы не можем указывать другие параметры.

### 

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

Иногда возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определенную версию метода. Такая возможность еше называется **перегрузкой методов** (method overloading).

И в языке C# мы можем создавать в классе несколько методов с одним и тем же именем. Но при этом мы должны учитывать, что методы с одним и тем же именем должны иметь либо разное количество параметров, либо параметры разных типов.

Стоит отметить, что разные версии метода могут иметь разные возвращаемые значения.

Наряду с методами мы можем также перегружать операторы. Например, пусть у нас есть следующий класс Counter:

class Counter

{

    public int Value { get; set; }

}

И допустим, у нас есть два объекта класса Counter - два счетчика, которые мы хотим сравнивать или складывать на основании их свойства Value, используя стандартные операции сравнения и сложения.

Но на данный момент ни операция сравнения, ни операция сложения для объектов Counter не доступны. И для этого нам надо выполнить перегрузку нужных нам операторов.

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

public static возвращаемый\_тип operator оператор(параметры)

{  }

Этот метод должен иметь модификаторы **public static**, так как перегружаемый оператор будет использоваться для всех объектов данного класса. Далее идет название возвращаемого типа. Возвращаемый тип представляет тот тип, объекты которого мы хотим получить. К примеру, в результате сложения двух объектов Counter мы ожидаем получить новый объект Counter. А в результате сравнения двух мы хотим получить объект типа bool, который указывает истинно ли условное выражение или ложно.

public static Counter operator +(Counter c1, Counter c2)

{

    return new Counter { Value = c1.Value + c2.Value };

}

Также переопределены две операции сравнения. Если мы переопределяем одну из этих операций сравнения, то мы также должны переопределить вторую из этих операций. Сами операторы сравнения сравнивают значения свойств Value и в зависимости от результата сравнения возвращают либо true, либо false.

При перегрузке операторов надо учитывать, что не все операторы можно перегрузить. В частности, мы можем перегрузить следующие операторы:

* унарные операторы +, -, !, ~, ++, --
* бинарные операторы +, -, \*, /, %
* операции сравнения ==, !=, <, >, <=, >=
* логические операторы &&, ||
* операторы присваивания +=, -=, \*=, /=, %=

И есть ряд операторов, которые нельзя перегрузить, например, операцию равенства = или тернарный оператор ?:, а также ряд других.

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

**Автоматическое преобразование типов**

Когда данные одного типа присваиваются переменной другого типа, **неявное преобразование** типов происходит автоматически при следующих условиях:

* оба типа совместимы
* диапазон представления чисел целевого типа шире, чем у исходного типа

Если оба эти условия удовлетворяются, то происходит **расширяющее преобразование**. Например, тип int достаточно крупный, чтобы вмещать в себя все действительные значения типа byte, а кроме того, оба типа, int и byte, являются совместимыми целочисленными типами, и поэтому для них вполне возможно неявное преобразование.

Числовые типы, как целочисленные, так и с плавающей точкой, вполне совместимы друг с другом для выполнения расширяющих преобразований. Рассмотрим пример:

**class** **Program**

{

**static** **void** **Main**(**string**[] args)

{

**short** num1, num2;

num1 = 10;

num2 = 15;

Console.WriteLine("{0} + {1} = {2}",num1,num2,Sum(num1,num2));

Console.ReadLine();

}

**static** **int** **Sum**(**int** x, **int** y)

{

**return** x + y;

}

}

**Приведение несовместимых типов**

Несмотря на всю полезность неявных преобразований типов, они неспособны удовлетворить все потребности в программировании, поскольку допускают лишь расширяющие преобразования совместимых типов. А во всех остальных случаях приходится обращаться к приведению типов. Приведение — это команда компилятору преобразовать результат вычисления выражения в указанный тип. А для этого требуется явное преобразование типов. Ниже приведена общая форма приведения типов:

   (целевой\_тип) выражение

Здесь *целевой\_тип* обозначает тот тип, в который желательно преобразовать указанное выражение.

Если приведение типов приводит к сужающему преобразованию, то часть информации может быть потеряна. Например, в результате приведения [типа long](https://professorweb.ru/my/csharp/charp_theory/level3/3_6.php) к типу int часть информации потеряется, если значение типа long окажется больше диапазона представления чисел для типа int, поскольку старшие разряды этого числового значения отбрасываются. Когда же значение с плавающей точкой приводится к целочисленному, то в результате усечения теряется дробная часть этого числового значения.

после модификаторов **public static** идет ключевое слово **explicit** (если преобразование явное, то есть нужна операция приведения типов) или **implicit** (если преобразование неявное). Затем идет ключевое слово **operator** и далее возвращаемый тип, в который надо преобразовать объект. В скобках в качестве параметра передается объект, который надо преобразовать.

Например, пусть у нас есть следующий класс Counter, который представляет счетчик-секундомер и который хранит количество секунд в свойстве Seconds:

class Counter

{

    public int Seconds { get; set; }

    public static implicit operator Counter(int x)

    {

        return new Counter { Seconds = x };

    }

    public static explicit operator int(Counter counter)

    {

        return counter.Seconds;

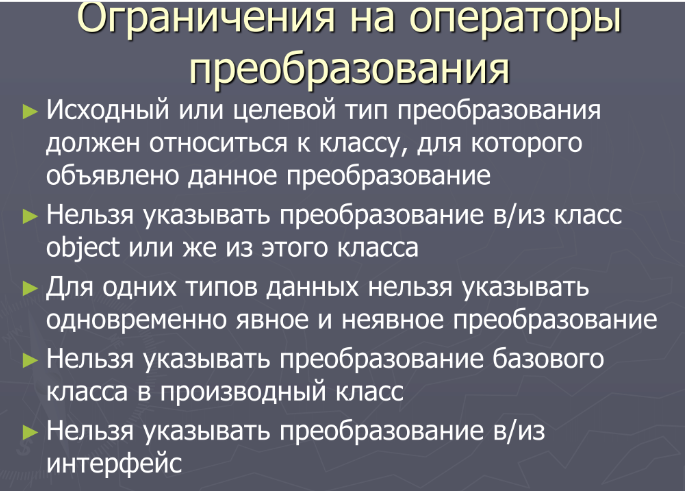
    }

}

Поскольку операция преобразования из Counter в int определена с ключевым словом explicit, то есть как явное преобразование, то в этом случае необходимо применить операцию приведения типов:

int x = (int)counter1;

В случае с операцией преобразования от int к Counter ничего подобного делать не надо, поскольку данная операция определена с ключевым словом implicit, то есть как неявная. Какие операции преобразования делать явными, а какие неявные, в данном случае не столь важно, это решает разработчик по своему усмотрению.



### 22. Вложенные типы, объекты

Тип, определенный внутри [класса](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/class) или [структуры](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/struct), называется вложенным типом. Пример:

class Container

{

class Nested

{

Nested() { }

}

}

Независимо от того, является ли внешний тип классом или структурой, вложенным типам по умолчанию присваиваеawтся модификатор [private](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/private), из-за чего они доступны только из содержащего их типа. В предыдущем примере класс Nested недоступен для внешних типов.

Также можно указать [модификатор доступа](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/access-modifiers), определяющий доступность вложенного типа, как показано ниже:

* Вложенные типы **класса** могут иметь модификаторы [public](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/public), [protected](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/protected), [internal](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/internal), [protected internal](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/protected-internal), [private](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/private) или [private protected](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/private-protected).

Тем не менее при определении вложенного класса protected, protected internal или private protected внутри [запечатанного класса](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/sealed) возникает предупреждение компилятора [CS0628](https://docs.microsoft.com/ru-ru/dotnet/csharp/misc/cs0628)"Новый защищенный член объявлен в запечатанном классе".

* Вложенные типы **структуры** могут иметь модификаторы [public](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/public), [internal](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/internal) или [private](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/private).

В следующем примере класс Nested определяется как открытый:

class Container

{

public 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. Пример:

public class Container

{

public class Nested

{

private Container parent;

public Nested()

{

}

public Nested(Container parent)

{

this.parent = parent;

}

}

}

Вложенный тип имеет доступ ко всем членам, которые доступны вмещающему типу. Он может получать доступ к закрытым и защищенным членам вмещающего типа, включая любые наследуемые защищенные члены.

В предыдущем объявлении полным именем класса Nested является Container.Nested. Это имя используется для создания нового экземпляра вложенного класса, как показано ниже:

Container.Nested nest = new Container.Nested();

### 23. Правила наследования C#.

Наследование (inheritance) является одним из ключевых моментов ООП. Благодаря наследованию один класс может унаследовать функциональность другого класса.

Пусть у нас есть следующий класс Person, описывающий отдельного человека:

class Person

{

    public string FirstName { get;  set; }

    public string LastName {get; set; }

    public Person(string firstName, string lastName)

    {

        FirstName = firstName;

        LastName = lastName;

    }

    public void Display()

    {

        Console.WriteLine($"{FirstName} {LastName}");

    }

}

class Employee : Person

{

    public string Company { get; set; }

    public Employee(string firstName, string lastName, string comp)

        : base(firstName, lastName)

    {

        Company = comp;

    }

}

Но вдруг нам потребовался класс, описывающий сотрудника предприятия - класс Employee. Поскольку этот класс будет реализовывать тот же функционал, что и класс Person, так как сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (или наследником, или подклассом) от класса Person, который, в свою очередь, называется базовым классом или родителем (или суперклассом):

И поскольку объект Employee является также и объектом Person, то мы можем так определить переменную: Person p = new Employee().

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

* Не поддерживается множественное наследование, класс может наследоваться только от одного класса. Хотя проблема множественного наследования реализуется с помощью концепции интерфейсов, о которых мы поговорим позже.
* При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**.
* Если класс объявлен с модификатором **sealed**, то от этого класса нельзя наследовать и создавать производные классы.

### 24. 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.

**Пример** (на основании предыдущего примера):

//квадрат наследует прямоугольник

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();

}

**Ключевое слово base**

С помощью ключевого слова **base** мы можем обратиться к базовому классу. В нашем случае в конструкторе класса Employee нам надо установить имя, фамилию и компанию. Но имя и фамилию мы передаем на установку в конструктор базового класса, то есть в конструктор класса Person, с помощью выражения base(fName, lName).

static void Main(string[] args)

{

    Person p = new Person("Bill", "Gates");

    p.Display();

    Employee emp = new Employee ("Tom", "Simpson", "Microsoft");

    emp.Display();

    Console.Read();

}

### 25.Использование операций is и as

Поскольку явное преобразование может потерпеть неудачу с генерацией искл??чения, бывает так, что необходимо проверить тип переменной без выполнения приведения и наблюдения, получится оно или нет.

Проверка типа через приведение утомительна и неэффективна, к тому же исключения дорого обходятся во время выполнения. По этой причине в С# предусмотрены операции, которые приходят на помощь в таких ситуациях и применение которых гарантированно не приведет к исключениям:

•is is  
• as

Операция is дает в результате булевское значение, говорящее о том, можно ли преобразовать данное выражение в указанный тип, как с помощью преобразования ссылки, так и посредством операций упаковки и распаковки. Например, рассмотрим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | using System;  public class EntryPoint  {  static void Main() {  String derivedObj = "Dummy";  Object baseObjl = new Object ();  Object baseObj2 = derivedObj;  Console.WriteLine ( "baseObj2 {0} String",  baseObj2 is String ? "является" : "не является" );  Console.WriteLine ( "baseObjl {0} String",  baseObjl is String ? "является" : "не является" );  Console.WriteLine ( "derivedObj {0} Object",  derivedObj is Object ? "является" : "не является" );  int j = 123;  object boxed = j;  object obj = new Object () ;  Console.WriteLine ( "boxed {0} int",  boxed is int ? "является" : "не является" ) ;  Console.WriteLine ( "obj {0} int",  obj is int ? "является" : "не является" );  Console.WriteLine( "boxed {0} System.ValueType",  boxed is ValueType ? "является" : "не является" ) ;  } |

**Вывод этого кода будет таким:**  
baseObj2 является String  
baseObjl не является String  
derivedObj является Object  
boxed является int  
obj не является int  
boxed является System.ValueType

Как уже упоминалось, операция is принимает во внимание только преобразования ссылок. Это значит, что она не может проверять определенные пользователем преобразования, имеющиеся в типах.

Операция as подобна is за исключением того, что она возвращает ссылку на целевой тип. Поскольку гарантируется, что она никогда не сгенерирует исключения, то если данное преобразование невозможно, просто возвращается null-ссылка. Подобно is, операция as принимает во внимание только преобразования ссылок или преобразования с упаковкой/распаковкой.

Например, взгляните на следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | using System;  public class BaseType {}  public class DerivedType : BaseType {}  public class EntryPoint  {  static void Main()  {  DerivedType derivedObj = new DerivedType ();  BaseType baseObjl = new BaseType ();  BaseType base0bj2 = derivedObj;  DerivedType derivedObj2 = base0bj2 as DerivedType;  if( derivedObj2 != null ) {  Console.WriteLine ( "Преобразование успешно" );  } else {  Console.WriteLine ( "Преобразование не удалось" );  }  derivedObj2 = baseObjl as DerivedType;  if( derivedObj2 != null ) {  Console.WriteLine ( "Преобразование успешно" );  } else {  Console.WriteLine( "Преобразование не удалось" );  }  BaseType base0bj3 = derivedObj as BaseType;  if ( base0bj3 != null ) {  Console.WriteLine ( "Преобразование успешно" );  } else {  Console.WriteLine ( "Преобразование не удалось" ); |

**Вывод этого кода будет таким:**  
Преобразование успешно  
Преобразование не удалось  
Преобразование успешно

Иногда требуется проверить, относится ли переменная к определенному типу, и если да, то выполнить какую-то операцию над нужным типом. Проверить переменную на принадлежность к типу можно с использованием операции is, а затем, если она вернет true, привести переменную к этому типу. Однако это будет не эффективно.

Более удачный подход заключается в том, чтобы следовать идиоме применения операции as для получения ссылки на переменную с нужным типом, а затем проверить ее на неравенство null, что будет означать успешность преобразования.

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

Сам термин полиморфизм можно перевести как **«много форм»**. А если говорить проcтыми словами, **полиморфизм** – это различная реализация однотипных действий. Классическая фраза, которая коротко объясняет полиморфизм – «Один интерфейс, множество реализаций». Приведу примеры из жизни. В автомобилях есть рулевое колесо. Это колесо является интерфейсом между водителем и автомобилем, который позволяет поворачивать автомобиль. Механическая реализация руля у автомобилей может быть разная, но при этом результат получается одинаковым – колесо вправо – автомобиль вправо, и наоборот.

**Виртуальный метод** – это метод, который МОЖЕТ быть переопределен в классе-наследнике. Такой метод может иметь стандартную реализацию в базовом классе.  
  
**Абстрактный метод** – это метод, который ДОЛЖЕН быть реализован в классе-наследнике. При этом, абстрактный метод не может иметь своей реализации в базовом классе (тело пустое), в отличии от виртуального.  
  
**Переопределение метода** – это изменение реализации метода, установленного как виртуальный (в классе наследнике метод будет работать отлично от базового класса).

При наследовании нередко возникает необходимость при наследовании изменить в классе-наследнике функционал метода, который был унаследован от базового класса. В этом случае класс-наследник может переопределять методы и свойства базового класса.

Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечается модификатором **virtual**. Такие методы и свойства называют виртуальными.

А чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором **override**. Переопределенный метод в класе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

Например, рассмотрим следующие классы:

class Person

{

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public Person(string firstName, string lastName)

    {

        FirstName = firstName;

        LastName = lastName;

    }

    public virtual void Display()

    {

        Console.WriteLine($"{FirstName} {LastName}");

    }

}

class Employee : Person

{

    public string Company { get; set; }

    public Employee(string firstName, string lastName, string company)

        : base(firstName, lastName)

    {

        Company = company;

    }

}

Запрет переопределения методов

Также можно запретить переопределение методов и свойств. В этом случае их надо объявлять с модификатором **sealed**:

class Employee : Person

{

    public string Company { get; set; }

    public Employee(string firstName, string lastName, string company)

                : base(firstName, lastName)

    {

        Company = company;

    }

    public override sealed void Display()

    {

        Console.WriteLine($"{FirstName} {LastName} работает в {Company}");

    }

}

При создании методов с модификатором sealed надо учитывать, что sealed применяется в паре с override, то есть только в переопределяемых методах.

И в этом случае мы не сможем переопределить метод Display в классе, унаследованном от Employee.

### 27.Понятие раннего и позднего связывания.

**Ранее связывание**

Возьмем пример с сокрытием метода:

class Person

{

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public Person(string firstName, string lastName)

    {

        FirstName = firstName;

        LastName = lastName;

    }

    public void Display()

    {

        Console.WriteLine($"{FirstName} {LastName}");

    }

}

class Employee : Person

{

    public string Company { get; set; }

    public Employee(string firstName, string lastName, string company)

            : base(firstName, lastName)

    {

        Company = company;

    }

    public new void Display()

    {

        Console.WriteLine($"{FirstName} {LastName} работает в {Company}");

    }

}

И посмотрим, что будет в следующем случае:

Person tom = new Employee("Tom", "Smith", "Microsoft");

tom.Display();      // Tom Smith

Переменная tom представляет тип Person, но хранит ссылку на объект Employee. Однако при вызове метода Display будет выполняться та версия метода, которая определена именно в классе Person, а не в классе Employee. Почему?

При присвоении переменной базового класса объекта производного класса по умолчанию через эту переменную можно вызвать только те методы и свойства, которые есть в базовом классе. Компилятор до выполнения программы на этапе компиляции должен определить адрес метода, который будет вызываться. Данный процесс называется **ранним связыванием** (early binding). При этом при выборе реализации метода компилятор руководствуется типом переменной, а не типом объекта, ссылку на который хранится в этой переменной. Поэтому при вызове tom.Display() вызывается метод Display из класса Person.

**Позднее связывание**

Но также есть механизм **позднего связывания**, который представляет выбор реализации метода на этапе выполнения, когда известно, ссылку на объект какого типа будет хранить переменная. И в зависимости от типа объекта выбирается нужная реализация для метода. Позднее связывание реализуется с помощью виртуальных методов.

Теперь возьмем те же классы Person и Employee, но сделаем метод Display виртуальным и переопределим его в классе Employee:

class Person

{

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public Person(string firstName, string lastName)

    {

        FirstName = firstName;

        LastName = lastName;

    }

    public virtual void Display()

    {

        Console.WriteLine($"{FirstName} {LastName}");

    }

}

class Employee : Person

{

    public string Company { get; set; }

    public Employee(string firstName, string lastName, string company)

        : base(firstName, lastName)

    {

        Company = company;

    }

    public override void Display()

    {

        Console.WriteLine($"{FirstName} {LastName} работает в {Company}");

    }

}

Также создадим объект Employee и передадим его переменной типа Person:

Person tom = new Employee("Tom", "Smith", "Microsoft");

tom.Display();      // Tom Smith работает в Microsoft

Теперь мы получаем иной результат, нежели при сокрытии. А при вызове tom.Display()выполняется реализация метода Display из класса Employee. В этом как раз и проявляется механизм позднего связывания.

Для реализации позднего связывания компилятор формирует таблицу виртуальных методов (Virtual Method Table или VMT). В нее записывается адреса виртуальных методов. Для каждого класса создается своя таблица.

Когда создается объект класса, то компилятор передает в конструктор объекта специальный код, который связывает объект и таблицу VMT.

А при вызове виртуального метода из объекта берется адрес его таблицы VMT. Затем из VMT извлекается адрес метода и ему передается управление. Собственно так и выполняется виртуальный метод. Следует учитывать, что так как среде выполнения вначале необходимо получить из таблицы VMT адрес нужного метода, то это немного замедляет выполнение программы.

### 28. Абстрактные классы и методы. Бесплодные классы.

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

abstract class [имя\_класса]  
{  
   //тело  
}

Такой класс имеет следующие особенности:  
- нельзя создавать экземпляры (объекты) абстрактного класса;  
- абстрактный класс может содержать как абстрактные методы/свойства, так и обычные;  
- в классе наследнике должны быть реализованы все абстрактные методы и свойства, объявленные в базовом классе.  
  
**Зачем нужны абстрактные классы?**  
  
В самом по себе абстрактном классе, от которого никто не наследуется, смысла нет, так как нельзя создавать его экземпляры. В абстрактном классе обычно реализуется некоторая общая часть нескольких сущностей или другими словами - абстрактная сущность, которая, как объект, не может существовать, и эта часть необходима в классах наследниках. Конкретные примеры будут дальше.  
  
**Абстрактные методы**  
  
Понимание абстрактных методов не будет для вас сложной задачей, если вы уже ознакомились с [виртуальными методами](http://mycsharp.ru/post/33/2013_10_19_virtualnye_metody_v_si-sharp_pereopredelenie_metodov.html).  
  
**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.  
  
Какая же разница между виртуальным и абстрактным методом?  
  
- Виртуальный метод может иметь свою реализацию в базовом классе, абстрактный – нет (тело пустое);  
- Абстрактный метод должен быть реализован в классе наследнике, виртуальный метод переопределять необязательно.  
  
Объявление абстрактного метода происходит при помощи ключевого слова *abstract*, и при этом фигурные скобки опускаются, точка с запятой ставится после заголовка метода:

[модификатор доступа] abstract [тип] [имя метода] ([аргументы]);

Реализация абстрактного метода в классе наследнике происходит так же, как и переопределение метода – при помощи ключевого слова *override*:

[модификатор доступа] override [тип] [имя метода] ([аргументы])  
{  
  // реализация метода  
}

**Абстрактные свойства**  
  
Создание абстрактных [свойств](http://mycsharp.ru/post/27/2013_07_12_svojstva_v_si-sharp_avtomaticheskie_svojstva_aksessory_get_i_set.html) не сильно отличается от методов:

protected [тип] [поле, которым управляет свойство];  
[модификатор доступа] abstract [тип] [имя свойства] { get; set; }

Реализация в классе-наследнике:

[модификатор доступа] override [тип] [имя свойства]   
{   
  get { тело аксессора get }  
  set { тело аксессора set }  
}

**Пример**  
  
В качестве примера, приведу программу похожую на ту, которая была в предыдущем уроке о виртуальных методах, где выводилась информация о человеке/студенте/школьнике. Сейчас уже будут животные. Тогда мы могли создать человека без статуса (не студент, не школьник), у которого была некоторая информация, а сейчас у нас будет абстрактная сущность *Животное*, объект которой создавать нельзя и нет смысла, так как каждое животное будет конкретного подцарства – млекопитающее, рыба, птица.

В С# есть ключевое слово sealed, позволяющее описать класс, от которого, в про­тивоположность абстрактному, наследовать запрещается:

sealed class Spirit

{

. . .

}

// class Monster : Spirit { ... } ошибка!

Большинство встроенных типов данных описано как sealed.Если необходимо ис­пользовать функциональность бесплодного класса, применяется не наследование, а*вложение*, или*включение*: в классе описывается поле соответствующего типа.

Вложение классов, когда один класс включает в себя поля, являющиеся класса­ми, является *альтернативой наследованию*при проектировании. Например, если есть объект «двигатель», а требуется описать объект «самолет», логично сделать двигатель полем этого объекта, а не его предком.

Поскольку поля класса обычно закрыты, возникает вопрос, как же пользоваться методами включенного объекта. Общепринятый способ состоит в том, чтобы описать метод объемлющего класса, из которого вызвать метод включенного класса. Такой способ взаимоотношений классов известен как модель бключения-делегирования.

### 29. Структур в C#.

Кроме базовых элементарных типов данных и перечислений в C# имеется и составной тип данных, который называется **структурой**. Структуры могут содержать в себе обычные переменные и методы.

Структура относится к типу значения, класс – к ссылочному типу данных, т.е. структуры размещаются в стеке, а классы – в куче.  
Также структуры не поддерживают наследование.

Может содержать: структура этого же типа, указатель на такую же структуру, указатель на функцию, прототип функции, объединение, перечисление, реализовывать интерфейсы, значение NULL, объявление конструкторов с параметрами.

Не может: инициализированные при объявлении поля (если не константы и не статические), не может объявлять конструктор (деструктор) по умолчанию.

Для примера создадим структуру Book, в которой будут храниться переменные для названия, автора и года издания книги. Кроме того, структура будет содержать метод для вывода информации о книге на консоль:

struct Book

{

    public string name;

    public string author;

    public int year;

    public void Info()

    {

        Console.WriteLine($"Книга '{name}' (автор {author}) была издана в {year} году");

    }

}

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 (как в данном случае), так и внутри класса, но не внутри метода.

По сути структура Book представляет новый тип данных. Мы также можем использовать массив структур:

Book[] books=new Book[3];

books[0].name = "Война и мир";

books[0].author = "Л. Н. Толстой";

books[0].year = 1869;

books[1].name = "Преступление и наказание";

books[1].author = "Ф. М. Достоевский";

books[1].year = 1866;

books[2].name = "Отцы и дети";

books[2].author = "И. С. Тургенев";

books[2].year = 1862;

foreach (Book b in books)

{

    b.Info();

}

**Конструкторы в структурах**

Кроме обычных методов структура может содержать специальный метод - конструктор, который выполняет некую начальную инициализацию объекта, например, присваивает всем полям некоторые значения по умолчанию. В принципе для структуры необязательно использовать конструктор:

*Book book1;*

Однако в этом случае, чтобы использовать структуру, нам надо будет первоначально проинициализировать все ее поля:

book1.name = "Война и мир";

book1.author = "Л. Н. Толстой";

book1.year = 1869;

### 30. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов.

В языке C# подобную проблему позволяют решить интерфейсы. Они играют важную роль в системе ООП. Интерфейсы позволяют определить некоторый функционал, не имеющий конкретной реализации. Затем этот функционал реализуют классы, применяющие данные интерфейсы.

Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы **I**, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования. Интерфейсы также, как и классы, могут содержать свойства, методы и события, только без конкретной реализации.

Изменим пустой код интерфейса IAccount на следующий:

interface IAccount

{

int CurrentSum { get; }

void Put(int sum);

void Withdraw(int sum); }

У интерфейса методы и свойства не имеют реализации, в этом они сближаются с абстрактными методами абстрактных классов. Сущность данного интерфейса проста: он определяет свойство для текущей суммы денег на счете и два метода для добавления денег на счет и изъятия денег.

Еще один момент в объявлении интерфейса: все его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Как и в случае с абстрактными методами абстрактного классом класс Client реализует все методы и свойства интерфейса. При этом поскольку все методы и свойства интерфейса являются публичными, при реализации этих методов и свойств в классе к ним можно применять только модификатор public. Поэтому если класс должен иметь метод с каким-то другим модификатором, например, protected, то интерфейс не подходит для определения подобного метода.

Если класс применяет интерфейс, то этот класс должен реализовать все методы и свойства интерфейса. Однако также можно и не реализовать методы, сделав их абстрактными, переложив право их реализации на производные классы.

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

Единственный класс или структура может реализовать любое количество интерфейсов. Из-за этого всегда существует вероятность реализации интерфейсов с членами, имеющими идентичные имена, и, следовательно, возникает необходимость в устранении конфликтов на уровне имен. При реализации члена интерфейса имеется возможность указать его имя полностью вместе с именем самого интерфейса. В этом случае получается явная реализация члена интерфейса, или просто **явная реализация**.

Когда один интерфейс наследует другой, то в производном интерфейсе может быть объявлен член, скрывающий член с аналогичным именем в базовом интерфейсе. Такое сокрытие имен происходит в том случае, если член в производном интерфейсе объявляется таким же образом, как и в базовом интерфейсе. Но если не указать в объявлении члена производного интерфейса ключевое слово new, то компилятор выдаст соответствующее предупреждающее сообщение.

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

Важно уяснить, что интерфейсы являются фундаментальным компонентом .NET Framework. Какого бы типа приложение не разрабатывалось (веб-приложение, приложение с настольным графическим интерфейсом, библиотека доступа к данными и т.п.), работа с интерфейсами будет обязательной частью этого процесса. Подводя итог всему изложенному, отметим, что интерфейсы могут приносить чрезвычайную пользу в следующих случаях:

При наличии единой иерархии, в которой только какой-то набор производных типов поддерживает общее поведение.

При необходимости моделировать общее поведение, которое должно встречаться в нескольких иерархиях, не имеющих общего родительского класса помимо System.Object.

* **using** System;
* **namespace** ConsoleApplication1
* {
* **public** interface IName
* {
* **void** **WriteName**();
* }
* **public** interface INameFamily
* {
* *// Объявляем в данном интерфейсе такой же метод*
* **void** **WriteName**();
* **void** **WriteFamily**();
* }
* **public** interface IUserInfo : INameFamily
* {
* *// Обязательно нужно указать ключевое слово new*
* *// чтобы не скрывались методы базового интерфейса*
* **new** **void** **WriteName**();
* **void** **WriteUserInfo**();
* }
* *// Класс, реализующий два интерфейса*
* **class** UserInfo : IUserInfo,IName
* {
* string ShortName, Family, Name;
* **public** **UserInfo**(string Name, string Family, string ShortName)
* {
* **this**.Name = Name;
* **this**.Family = Family;
* **this**.ShortName = ShortName;
* }
* *// Используем явную реализацию интерфейсов*
* *// для исключения неоднозначности*
* **void** IName.WriteName()
* {
* Console.WriteLine("Короткое имя: " + ShortName);
* }
* **void** INameFamily.WriteFamily()
* {
* Console.WriteLine("Фамилия: " + Family);
* }
* **void** INameFamily.WriteName()
* {
* Console.WriteLine("Полное имя: " + Name);
* }
* **void** IUserInfo.WriteName() { }
* **public** **void** **WriteUserInfo**()
* {
* UserInfo obj = **new** UserInfo(Name, Family, ShortName);
* *// Для использования закрытых методов необходимо*
* *// создать интерфейсную ссылку*
* IName link1 = (IName)obj;
* link1.WriteName();
* INameFamily link2 = (INameFamily)obj;
* link2.WriteName();
* link2.WriteFamily();
* IUserInfo link3 = (IUserInfo)obj;
* link3.WriteName();
* }
* }
* **class** Program
* {
* **static** **void** **Main**()
* {
* UserInfo obj = **new** UserInfo(Name: "Alexandr", ShortName: "Alex", Family: "Erohin");
* obj.WriteUserInfo();
* Console.ReadLine();
* }
* }

### 32. Ковариантность интерфейсов. Контравариантность интерфейсов

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

Имеется три возможных варианта поведения:

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

Начиная с .NET 4.0 в C# была добавлена возможность создания ковариантных и контравариантных обобщенных интерфейсов. Это функциональность повышает гибкость при использовании обобщенных интерфейсов в программе. По умолчанию все обобщенные интерфейсы, например, IAccout<T> являются инвариантными.

Обобщенные интерфейсы могут быть ковариантными, если к универсальному параметру применяется ключевое слово **out**. Например:

lass Account

{

    static Random rnd = new Random();

    public void DoTransfer()

    {

        int sum = rnd.Next(10, 120);

        Console.WriteLine("Клиент положил на счет {0} долларов", sum);

    }

}

class DepositAccount : Account

{

}

interface IBank<out T>  where T : Account

{

    T DoOperation();

}

class Bank: IBank<DepositAccount>

{

    public DepositAccount DoOperation()

    {

        DepositAccount acc = new DepositAccount();

        acc.DoTransfer();

        return acc;

    }

}

Для создания контравариантного интерфейса надо использовать ключевое слово in. Для его применения изменим интерфейс IBank и класс Bank:

interface IBank<in T>  where T : Account

{

    void DoOperation(T account);

}

class Bank<T>: IBank<T> where T : Account

{

    public void DoOperation(T account)

    {

        account.DoTransfer();

    }

}

То есть объект интерфейса с более универсальным типом приводится к объекту интерфейса с более конкретным типом.

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

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

#### Стандартные интерфейсы .NET

В библиотеке классов .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Например, интерфейс IComparable задает метод сравнения объектов на больше-меньше, что позволяет выполнять их сортировку. Реализация интерфейсов IEnumerable и IEnumerator дает возможность просматривать содержимое объекта с помощью конструкции foreach, а реализация интерфейса *ICloneable* — клонировать объекты.

Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки. Например, работа с массивами с помощью цикла foreach возможна именно потому, что тип Array реализует интерфейсы IEnumerable и IEnumerator. Можно создавать и собственные классы, поддерживающие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами.

##### Сравнение объектов

Интерфейс IComparable определен в пространстве имен System. Он содержит всего один метод CompareTo, возвращающий результат сравнения двух объектов — текущего и переданного ему в качестве параметра:

interface IComparable

{

int CompareTo( object obj )

}

Метод должен возвращать:

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

##### Сортировка по разным критериям (интерфейс IComparer)

Интерфейс IComparer определен в пространстве имен System.Collections. Он содержит один метод Compare, возвращающий результат сравнения двух объектов, переданных ему в качестве параметров:

interface IComparer

{

int Compare( object ob1, object ob2 )

}

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

##### Перегрузка операций отношения

Если класс реализует интерфейс IComparable, его экземпляры можно сравнивать между собой на больше-меньше. Логично разрешить использовать для этого операции отношения, перегрузив их. Операции должны перегружаться парами: < и >, <= и >=, == и !=. *Перегрузка операций* обычно выполняется путем делегирования, то есть обращения к переопределенным методам CompareTo и Equals.

**Примечание**

*Если класс реализует интерфейс*IComparable, *требуется переопределить метод*Equals *и связанный с ним метод* GetHashCode. *Оба метода унаследованы от базового класса*object.

##### Клонирование объектов

*Клонирование* — это создание копии объекта. Копия объекта называется клоном. Как известно, при присваивании одного объекта ссылочного типа другому копируется ссылка, а не сам объект. Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом MemberwiseClone, который любой объект наследует от класса object. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются. Это называется *поверхностным клонированием*.

Для создания полностью независимых объектов необходимо *глубокое клонирование*, когда в памяти создается дубликат всего дерева объектов, то есть объектов, на которые ссылаются поля объекта, поля полей, и так далее. Алгоритм глубокого клонирования весьма сложен, поскольку требует *рекурсивного обхода* всех ссылок объекта и отслеживания циклических зависимостей.

Объект, имеющий собственные алгоритмы клонирования, должен объявляться как наследник интерфейса *ICloneable* и переопределять его единственный метод Clone. В [листинге 9.4](https://www.intuit.ru/studies/courses/629/485/lecture/11015?page=4#example.9.4) приведен пример создания поверхностной копии объекта класса Monster с помощью метода MemberwiseClone, а также реализован интерфейс *ICloneable*. В демонстрационных целях в имя клона объекта добавлено слово "Клон".

Вася. Следовательно, если мы внесем изменения в один из этих объектов, это отразится на другом. Объекты Y и Z, созданные путем клонирования, обладают собственными копиями значений полей и независимы от исходного объекта.

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

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

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

Интерфейс IEnumerator задает три элемента:

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

Цикл foreach использует эти методы для перебора элементов, из которых состоит объект.

Таким образом, если требуется, чтобы для перебора элементов класса мог применяться цикл foreach, необходимо реализовать четыре метода: GetEnumerator, Current, MoveNext и Reset. Это не интересная работа, а выполнять ее приходится часто, поэтому в версию 2.0 были введены средства, облегчающие выполнение перебора в объекте — итераторы.

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

### 34. Исключительные ситуации. Генерация и обработка.

Иногда при выполнении программы возникают ошибки, которые трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Например, при передачи файла по сети может неожиданно оборваться сетевое подключение. такие ситуации называются **исключениями**. Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция **try...catch...finally**. При возникновении исключения среда CLR ищет блок catch, который может обработать данное исключение. Если такого блока не найдено, то пользователю отображается сообщение о необработанном исключении, а дальнейшее выполнение программы останавливается. И чтобы подобной остановки не произошло, и надо использовать блок try..catch. Например

static void Main(string[] args)

{

    int[] a = new int[4];

    try

    {

        a[5] = 4; // тут возникнет исключение, так как у нас в массиве только 4 элемента

        Console.WriteLine("Завершение блока try");

    }

    catch (Exception ex)

    {

        Console.WriteLine("Ошибка: " + ex.Message);

    }

    finally

    {

        Console.WriteLine("Блок finally");

    }

    Console.ReadLine();

}

При использовании блока **try...catch..finally** вначале выполняются все инструкции между операторами try и catch. Если между этими операторами вдруг возникает исключение, то обычный порядок выполнения останавливается и переходит к инструкции сatch. В данном случае у нас явно возникнет исключение в блоке try, так как мы пытаемся присвоить значение шестому элементу массива в то время, как в массиве всего 4 элемента. И дойдя до строки a[5] = 4;, выполнение программы остановится и перейдет к блоку catch

Инструкция **catch** имеет следующий синтаксис: catch (тип\_исключения имя\_переменной). В нашем случае объявляется переменная ex, которая имеет тип Exception. Но если возникшее исключение не является исключением типа, указанного в инструкции сatch, то оно не обрабатывается, а программа просто зависает или выбрасывает сообщение об ошибке.

Однако так как тип Exception является базовым классом для всех исключений, то выражение catch (Exception ex) будет обрабатывать практически все исключения. Вся обработка исключения в нашем случае сводится к выводу на консоль сообщении об исключении, которое в свойстве message класса Exception.

Далее в любом случае выполняется блок finally. Однако этот блок необязательный, и его можно при обработке исключений опускать. Если же в ходе программы исключений не возникнет, то программа не будет выполнять блок catch, сразу перейдет к блоку finally, если он имеется.

**Оператор throw**

Чтобы сообщить о выполнении исключительных ситуаций в программе, можно использовать оператор **throw**. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения.

**Фильтры исключений**

В C# 6.0 (Visual Studio 2015) была добавлена такая функциональность, как фильтры исключений. Они позволяют обрабатывать исключения в зависимости от определенных условий:

int x = 1;

int y = 0;

try

{

    int result = x / y;

}

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

{

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

}

catch(Exception ex)

{

    Console.WriteLine(ex.Message);

}

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

### 35. Обобщения (generics). Свойства обобщений. Статические члены обощений.

Мы можем не знать, какий именно объект представляет Id, и при попытке получить число в данном случае мы столкнемся с исключением InvalidCastException.

Эти проблемы были призваны устранить обобщенные типы. Обобщенные типы позволяют указать конкретный тип, который будет использоваться. Поэтому определим класс Account как обощенный:

class Account<T>

{

    public T Id { get; set; }

    public int Sum { get; set; }

}

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

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

Account<int> account1 = new Account<int> { Sum = 5000 };

Account<string> account2 = new Account<string> { Sum = 4000 };

account1.Id = 2;        // упаковка не нужна

account2.Id = "4356";

int id1 = account1.Id;  // распаковка не нужна

string id2 = account2.Id;

Conso le.WriteLine(id1);

Console.WriteLine(id2);

**Значения по умолчанию**

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

T id = null;

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

class Account<T>

{

    T id = default(T);

}

**Статические поля обобщенных классов**

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

class Account<T>

{

    public static T session;

    public T Id { get; set; }

    public int Sum { get; set; }

}

Теперь типизируем класс двумя типами int и string:

Account<int> account1 = new Account<int> { Sum = 5000 };

Account<int>.session = 5436;

Account<string> account2 = new Account<string> { Sum = 4000 };

Account<string>.session = "45245";

Console.WriteLine(Account<int>.session);      // 5436

Console.WriteLine(Account<string>.session);   // 45245

Обобщенные методы

Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. Например:

class Program

{

    private static void Main(string[] args)

    {

        int x = 7;

        int y = 25;

        Swap<int>(ref x, ref y);

        Console.WriteLine($"x={x}    y={y}");   // x=25   y=7

        string s1 = "hello";

        string s2 = "bye";

        Swap<string>(ref s1, ref s2);

        Console.WriteLine($"s1={s1}    s2={s2}"); // s1=bye   s2=hello

        Console.Read();

    }

    public static void Swap<T> (ref T x, ref T y)

    {

        T temp = x;

        x = y;

        y = temp;

    }

}

Здесь определен обощенный метод Swap, который принимает параметры по ссылке и меняет их значения. При этом в данном случае не важно, какой тип представляют эти параметры.

В методе Main вызываем метод Swap, типизируем его определенным типом и передаем ему некоторые значения.

### 36. Концепция ограничений обобщений

**Ограничения универсальных типов**

С помощью универсальных параметров мы можем типизровать обобщенные классы любым типом. Однако иногда возникает необходимость конкретизировать тип. Например, у нас есть следующий класс Account, который представляет банковский счет:

class Account

{

    public int Id { get; private set;} // номер счета

    public int Sum { get; set; }

    public Account(int \_id)

    {

        Id = \_id;

    }

}

Для перевода средств с одного счета на другой мы можем определить класс Transaction, который для выполнения всех операций будет использовать объекты класса Account.

Но у класса Account может быть много наследников: DepositAccount (депозитный счет), DemandAccount (счет до востребования) и т.д. И мы не можем знать, какие именно типы счетов будут использоваться в классе Transaction. Возможно, транзации будут проводиться только между счетами до востребования. И в этом случае в качестве универсального параметра можно установить тип Account:

class Transaction<T> where T: Account

{

    public T FromAccount { get; set; }  // с какого счета перевод

    public T ToAccount { get; set; }    // на какой счет перевод

    public int Sum { get; set; }        // сумма перевода

    public void Execute()

    {

        if (FromAccount.Sum > Sum)

        {

            FromAccount.Sum -= Sum;

            ToAccount.Sum += Sum;

            Console.WriteLine($"Счет {FromAccount.Id}: {FromAccount.Sum}$ \nСчет {ToAccount.Id}: {ToAccount.Sum}$");

        }

        else

        {

            Console.WriteLine($"Недостаточно денег на счете {FromAccount.Id}");

        }

    }

}

С помощью выражения where T : Account мы указываем, что используемый тип T обязательно должен быть классом Account или его наследником. Благодаря подобному ограничению мы можем использовать внутри класса Transaction все объекты типа T именно как объекты Account и соответственно обращаться к их свойствам и методам.

Следует учитывать, что только один класс может использоваться в качестве ограничения.

**Стандартные ограничения**

Есть ряд стандартных ограничений, которые мы можем использовать. В частности, можно указать ограничение, чтобы использовались только структуры или другие типы значений:

class Account<T> where T : struct {}

class Transaction<T> where T : class {}

А также можно задать с помощью слова **new** в качестве ограничения класс или структуру, которые имеют общедоступный конструктор без параметров:

class Transaction<T> where T : new() {}

**Ограничения методов**

Подобным образом можно использовать и ограничения методов:

private static void Main(string[] args)

{

    Account<int> acc1 = new Account<int>(1857) { Sum = 4500 };

    Account<int> acc2 = new Account<int>(3453) { Sum = 5000 };

    Transact<Account<int>>(acc1, acc2, 900);

    Console.Read();

}

public static void Transact<T>(T acc1, T acc2, int sum) where T : Account<int>

{

    if (acc1.Sum > sum)

    {

        acc1.Sum -= sum;

        acc2.Sum += sum;

    }

    Console.WriteLine($"acc1: {acc1.Sum}   acc2: {acc2.Sum}");

}

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

***Делегат*** представляет собой объект, который может ссылаться на метод. Следовательно, когда создается делегат, то в итоге получается объект, содержащий ссылку на метод. Более того, метод можно вызывать по этой ссылке. Иными словами, делегат позволяет вызывать метод, на который он ссылается.

По сути, делегат — это безопасный в отношении типов объект, указывающий на другой метод (или, возможно, список методов) приложения, который может быть вызван позднее. В частности, объект делегата поддерживает три важных фрагмента информации:

адрес метода, на котором он вызывается;

аргументы (если есть) этого метода;

возвращаемое значение (если есть) этого метода.

Как только делегат создан и снабжен необходимой информацией, он может динамически вызывать методы, на которые указывает, во время выполнения. Каждый делегат в .NET Framework (включая специальные делегаты) автоматически снабжается способностью вызывать свои методы синхронно или асинхронно. Этот факт значительно упрощает задачи программирования, поскольку позволяет вызывать метод во вторичном потоке выполнения без ручного создания и управления объектом Thread.

**Определение типа делегата в C#**

Тип делегата объявляется с помощью ключевого слова **delegate**. Ниже приведена общая форма объявления делегата:

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

где возвращаемый\_тип обозначает тип значения, возвращаемого методами, которые будут вызываться делегатом; имя — конкретное имя делегата; список\_параметров — параметры, необходимые для методов, вызываемых делегатом. Как только будет создан экземпляр делегата, он может вызывать и ссылаться на те методы, возвращаемый тип и параметры которых соответствуют указанным в объявлении делегата.

Самое главное, что делегат может служить для вызова любого метода с соответствующей сигнатурой и возвращаемым типом. Более того, вызываемый метод может быть методом экземпляра, связанным с отдельным объектом, или же статическим методом, связанным с конкретным классом. Значение имеет лишь одно: возвращаемый тип и сигнатура метода должны быть согласованы с теми, которые указаны в объявлении делегата.

delegate int Operation(int x, int y);

delegate void GetMessage();

Однако наиболее сильная сторона делегатов состоит в том, что они позволяют создать функционал методов обратного вызова, уведомляя другие объекты о произошедших событиях.

Как и методы, делегаты также могут быть обобщенными. Ниже приведена общая форма объявления обобщенного делегата:

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

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

Обобщенные делегаты предоставляют более гибкий способ спецификации метода, подлежащего вызову в безопасной к типам манере. До появления обобщений (в .NET 2.0) того же конечного результата можно было достичь с использованием параметра System.Object:

**public** **delegate** **void** **MyDelegate**(**object** arg);

### 38. Анонимные функции. Лямбда-выражения.

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

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

class Program

{

    delegate void MessageHandler(string message);

    static void Main(string[] args)

    {

        MessageHandler handler = delegate(string mes)

        {

            Console.WriteLine(mes);

        };

        handler("hello world!");

        Console.Read();

    }

}

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата, как в данном случае переменная handler представляет анонимный метод. И через эту переменную делегата можно вызвать данный анонимный метод.

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

Если анонимный метод использует параметры, то они должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе опустить параметры.

То есть если анонимный метод содержит параметры, они обязательно должны соответствовать параметрам делегата. Либо анонимный метод вообще может не содержать никаких параметров, тогда он соответствует любому делегату, который имеет тот же тип возвращаемого значения.

В каких ситуациях используются анонимные методы? Когда нам надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется. Иногда такие методы нужны для обработки одного события и больше ценности не представляют и нигде не применяются.

**Лямбда-выражени**я представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы.

Ламбда-выражения имеют следующий синтаксис: слева от лямбда-оператора =>определяется список параметров, а справа блок выражений, использующий эти параметры: (список\_параметров) => выражение. Например:

class Program

{

    delegate int Operation(int x, int y);

    static void Main(string[] args)

    {

        Operation operation = (x, y) => x + y;

        Console.WriteLine(operation(10, 20));       // 30

        Console.WriteLine(operation(40, 20));       // 60

        Console.Read();

    }

}

Здесь код (x, y) => x + y; представляет лямбда-выражение, где x и y - это параметры, а x + y - выражение. При этом нам не надо указывать тип параметров, а при возвращении результата не надо использовать оператор return.

При этом надо учитывать, что каждый параметр в лямбда-выражении неявно преобразуется в соответствующий параметр делегата, поэтому типы параметров должны быть одинаковыми. Кроме того, количество параметров должно быть таким же, как и у делегата. И возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата. То есть в данном случае использованное лямбда-выражение соответствует делегату Operation как по типу возвращаемого значения, так и по типу и количеству параметров.

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

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

**Action**

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

public delegate void Action<T>(T obj)

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

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

**Predicate**

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

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

Console.WriteLine(isPositive(20));

Console.WriteLine(isPositive(-20));

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

**Func**

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

TResult Func<out TResult>()

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

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

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

### 40. События.

удобные и простые конструкции под названием **события**, которые сигнализируют системе о том, что произошло определенное действие.

События объявляются в классе с помощью ключевого слова **event**, после которого идет название делегата:

// Объявляем делегат

public delegate void AccountStateHandler(string message);

// Событие, возникающее при выводе денег

public event AccountStateHandler Withdrawn;

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры, что и делегат, и возвращать тот же тип, что и делегат.

Для прикрепления обработчика события к определенному событию используется операция += и соответственно для открепления - операция -=: событие += метод\_обработчика\_события. Опять же обращаю внимание, что метод обработчика должен иметь такие же параметры, как и делегат события, и возвращать тот же тип.

Класс данных события AccountEventArgs

Если вдруг вы когда-нибудь создавали графические приложения с помощью Windows Forms или WPF, то, вероятно, сталкивались с обработчиками, которые в качестве параметра принимают аргумент типа EventArgs, например, обработчик нажатия кнопки private void button1\_Click(object sender, System.EventArgs e){} Параметр e, будучи объектом класса EventArgs, содержит все данные события.

### 41. Стандартные коллекции .NET. Типы коллекций.

В C# ***коллекция*** представляет собой совокупность объектов.

Например, в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связных списков, стеков, очередей и хеш-таблиц.

Главное преимущество коллекций заключается в том, что они стандартизируют обработку групп объектов в программе. Все коллекции разработаны на основе набора четко определенных интерфейсов. Некоторые встроенные реализации таких интерфейсов, в том числе ArrayList, Hashtable, Stack и Queue, могут применяться в исходном виде и без каких-либо изменений. Имеется также возможность реализовать собственную коллекцию, хотя потребность в этом возникает крайне редко.

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

**Необобщенные коллекции**

Реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также словари, в которых можно хранить пары "ключ-значение". В отношении необобщенных коллекций важно иметь в виду следующее: они оперируют данными типа object. Таким образом, необобщенные коллекции могут служить для хранения данных любого типа, причем в одной коллекции допускается наличие разнотипных данных.

**Специальные коллекции**

Оперируют данными конкретного типа или же делают это каким-то особым образом. Например, имеются специальные коллекции для символьных строк, а также специальные коллекции, в которых используется однонаправленный список. Специальные коллекции объявляются в пространстве имен **System.Collections.Specialized**.

**Обобщенные коллекции**

Обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связные списки, стеки, очереди и словари. Такие коллекции являются типизированными в силу их обобщенного характера. Это означает, что в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Благодаря этому исключается случайное несовпадение типов. Обобщенные коллекции объявляются в пространстве имен **System.Collections.Generic**.

**Параллельные коллекции**

Поддерживают многопоточный доступ к коллекции. Это обобщенные коллекции, определенные в пространстве имен **System.Collections.Concurrent**.

В пространстве имен System.Collections.ObjectModel находится также ряд классов, поддерживающих создание пользователями собственных обобщенных коллекций.

***Поразрядная коллекция***

В прикладном интерфейсе Collections API определена одна коллекция с поразрядной организацией — это BitArray. Коллекция типа BitArray поддерживает поразрядные операции, т.е. операции над отдельными двоичными разрядами, например И, ИЛИ, исключающее ИЛИ, а следовательно, она существенно отличается своими возможностями от остальных типов коллекций. Коллекция типа BitArray объявляется в пространстве имен System.Collections.

### 42. IEnumerable и IEnumerator

Основой для создания всех коллекций является реализация интерфейсов **IEnumerator** и **IEnumerable** (и их обобщенных двойников **IEnumerator<T>** и **IEnumerable<T>**). Интерфейс IEnumerator представляет перечислитель, с помощью которого становится возможен последовательный перебор коллекции, например, в цикле **foreach**. А интерфейс IEnumerable через свой метод GetEnumerator предоставляет перечислитель всем классам, реализующим данный интерфейс. Поэтому интерфейс IEnumerable(IEnumerable<T>) является базовым для всех коллекций.

Рассмотрим создание и применение двух коллекций:

using System;

using System.Collections;

using System.Collections.Generic;

namespace Collections

{

    class Program

    {

        static void Main(string[] args)

        {

            // необобщенная коллекция ArrayList

            ArrayList objectList = new ArrayList() { 1, 2, "string", 'c', 2.0f };

            object obj = 45.8;

            objectList.Add(obj);

            objectList.Add("string2");

            objectList.RemoveAt(0); // удаление первого элемента

            foreach (object o in objectList)

            {

                Console.WriteLine(o);

            }

            Console.WriteLine("Общее число элементов коллекции: {0}", objectList.Count);

            // обобщенная коллекция List

            List<string> countries = new List<string>() { "Россия", "США", "Великобритания", "Китай" };

            countries.Add("Франция");

            countries.RemoveAt(1); // удаление второго элемента

            foreach (string s in countries)

            {

                Console.WriteLine(s);

            }

            Console.ReadLine();

        }

    }

}

Здесь используются две коллекции: необобщенная - ArrayList и обобщенная - List. Большинство коллекций поддерживают добавление элементов. Например, в данном случае добавление производится методом Add, но для других коллекций название метода может отличаться. Также большинство коллекций реализуют удаление (в данном примере производится с помощью метода RemoveAt).

С помощью свойства Count у коллекций можно посмотреть количество элементов.

И так как коллекции реализуют интерфейс IEnumerable/IEnumerable<T>, то все они поддерживают перебор в цикле foreach.

Конкретные методы и способы использования могут различаться от одного класса коллекции к другому, но общие принципы будут одни и те же для всех классов коллекций.

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

public interface IEnumerable

{

    IEnumerator 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

}

### 43. LINQ to Objects. Синтаксис. Грамматика выражений запросов.

LINQ (Language-Integrated Query) представляет 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, документ XML. Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};

var selectedTeams = from t in teams // определяем каждый объект из teams как t

                    where t.ToUpper().StartsWith("Б") //фильтрация по критерию

                    orderby t  // упорядочиваем по возрастанию

                    select t; // выбираем объект

foreach (string s in selectedTeams)

    Console.WriteLine(s);

Чтобы использовать функциональность LINQ, убедимся, что в файле подключено пространство имен **System.LINQ**.

Простейшее определение запроса LINQ выглядит следующим образом:

from переменная in набор\_объектов

select переменная;

Итак, что делает этот запрос LINQ? Выражение from t in teams проходит по всем элементам массива teams и определяет каждый элемент как t. Используя переменную tмы можем проводить над ней разные операции.

Несмотря на то, что мы не указываем тип переменной t, выражения LINQ являются строго типизированными. То есть среда автоматически распознает, что набор teams состоит из объектов string, поэтому переменная t будет рассматриваться в качестве строки.

Далее с помощью оператора where проводится фильтрация объектов, и если объект соответствует критерию (в данном случае начальная буква должна быть "Б"), то этот объект передается дальше.

Оператор orderby упорядочивает по возрастанию, то есть сортирует выбранные объекты.

Оператор select передает выбранные значения в результирующую выборку, которая возвращается LINQ-выражением.

В данном случае результатом выражения LINQ является объект IEnumerable<T>. Нередко результирующая выборка определяется с помощью ключевого слова var, тогда компилятор на этапе компиляции сам выводит тип.

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

**Список используемых методов расширения LINQ**

* **Select**: определяет проекцию выбранных значений
* **Where**: определяет фильтр выборки
* **OrderBy**: упорядочивает элементы по возрастанию
* **OrderByDescending**: упорядочивает элементы по убыванию
* **ThenBy**: задает дополнительные критерии для упорядочивания элементов возрастанию
* **ThenByDescending**: задает дополнительные критерии для упорядочивания элементов по убыванию
* **Join**: соединяет две коллекции по определенному признаку
* **GroupBy**: группирует элементы по ключу
* **ToLookup**: группирует элементы по ключу, при этом все элементы добавляются в словарь
* **GroupJoin**: выполняет одновременно соединение коллекций и группировку элементов по ключу
* **Reverse**: располагает элементы в обратном порядке
* **All**: определяет, все ли элементы коллекции удовлятворяют определенному условию
* **Any**: определяет, удовлетворяет хотя бы один элемент коллекции определенному условию

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

int[] numbers = { 3, 12, 4, 10, 34, 20, 55, -66, 77, 88, 4 };

var orderedNumbers = from i in numbers

                     orderby i

                     select i;

foreach (int i in orderedNumbers)

    Console.WriteLine(i);

С помощью методов расширения то же самое можно сделать через метод **ThenBy()**(для сортировки по возрастанию) и **ThenByDescending()** (для сортировки по убыванию):

var result = users.OrderBy(u => u.Name).ThenBy(u => u.Age).ThenBy(u=>u.Name.Length);

### 45. Рефлексия

В мире .NET ***рефлексией (reflection)*** называется процесс обнаружения типов во время выполнения. С применением служб рефлексии те же самые метаданные, которые отображает утилита ildasm.exe, можно получать программно в виде удобной объектной модели. Например, рефлексия позволяет извлечь список всех типов, которые содержатся внутри определенной сборки \*.dll или \*.ехе (или даже внутри файла \*.netmodule если речь идет о многофайловой сборке), в том числе методов, полей, свойств и событий, определенных в каждом из них

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
* **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

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**.

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 **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

**Получение типа**

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().

Здесь 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();

И третий способ получения типа - статический метод Type.GetType():

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

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

Для работы с каталогами в пространстве имен System.IO предназначены сразу два класса: **Directory** и **DirectoryInfo**.

**Класс Directory**

Класс Directory предоставляет ряд статических методов для управления каталогами. Некоторые из этих методов:

* **CreateDirectory(path)**: создает каталог по указанному пути path
* **Delete(path)**: удаляет каталог по указанному пути path
* **Exists(path)**: определяет, существует ли каталог по указанному пути path. Если существует, возвращается true, если не существует, то false
* **GetDirectories(path)**: получает список каталогов в каталоге path
* **GetFiles(path)**: получает список файлов в каталоге path
* **Move(sourceDirName, destDirName)**: перемещает каталог
* **GetParent(path)**: получение родительского каталога

**Класс DirectoryInfo**

Данный класс предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Во многом он похож на Directory. Некоторые из его свойств и методов:

* **Create()**: создает каталог
* **CreateSubdirectory(path)**: создает подкаталог по указанному пути path
* **Delete()**: удаляет каталог
* Свойство **Exists**: определяет, существует ли каталог
* **GetDirectories()**: получает список каталогов
* **GetFiles()**: получает список файлов
* **MoveTo(destDirName)**: перемещает каталог
* Свойство **Parent**: получение родительского каталога
* Свойство **Root**: получение корневого каталога

**Создание каталога**

string path = @"C:\SomeDir";

string subpath = @"program\avalon";

DirectoryInfo dirInfo = new DirectoryInfo(path);

if (!dirInfo.Exists)

{

    dirInfo.Create();

}

dirInfo.CreateSubdirectory(subpath);

Подобно паре Directory/DirectoryInfo для работы с файлами предназначена пара классов **File** и **FileInfo**. С их помощью мы можем создавать, удалять, перемещать файлы, получать их свойства и многое другое.

Некоторые полезные методы и свойства класса FileInfo:

* **CopyTo(path)**: копирует файл в новое место по указанному пути path
* **Create()**: создает файл
* **Delete()**: удаляет файл
* **MoveTo(destFileName)**: перемещает файл в новое место
* Свойство **Directory**: получает родительский каталог в виде объекта DirectoryInfo
* Свойство **DirectoryName**: получает полный путь к родительскому каталогу
* Свойство **Exists**: указывает, существует ли файл
* Свойство **Length**: получает размер файла
* Свойство **Extension**: получает расширение файла
* Свойство **Name**: получает имя файла
* Свойство **FullName**: получает полное имя файла

Класс File реализует похожую функциональность с помощью статических методов:

* **Copy()**: копирует файл в новое место
* **Create()**: создает файл
* **Delete()**: удаляет файл
* **Move**: перемещает файл в новое место
* **Exists(file)**: определяет, существует ли файл

**Получение информации о файле**

string path = @"C:\apache\hta.txt";

FileInfo fileInf = new FileInfo(path);

if (fileInf.Exists)

{

    Console.WriteLine("Имя файла: {0}", fileInf.Name);

    Console.WriteLine("Время создания: {0}", fileInf.CreationTime);

    Console.WriteLine("Размер: {0}", fileInf.Length);

}

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

**Интерфейс IDisposable**

Интерфейс IDisposable объявляет один единственный метод Dispose, в котором при реализации интерфейса в классе должно происходить освобождение неуправляемых ресурсов.

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

using (Font font1 = new Font("Arial", 10.0f))

{

byte charset = font1.GdiCharSet;

}

Класс **FileStream** представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Рассмотрим наиболее важные его свойства и методы:

* Свойство **Length**: возвращает длину потока в байтах
* Свойство **Position**: возвращает текущую позицию в потоке
* Метод **Read**: считывает данные из файла в массив байтов. Принимает три параметра: int Read(byte[] array, int offset, int count) и возвращает количество успешно считанных байтов. Здесь используются следующие параметры:
  + array - массив байтов, куда будут помещены считываемые из файла данные
  + offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены
  + count - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.
* Метод **long Seek(long offset, SeekOrigin origin)**: устанавливает позицию в потоке со смещением на количество байт, указанных в параметре offset.
* Метод **Write**: записывает в файл данные из массива байтов. Принимает три параметра: Write(byte[] array, int offset, int count)
  + array - массив байтов, откуда данные будут записываться в файла
  + offset - смещение в байтах в массиве array, откуда начинается запись байтов в поток
  + count - максимальное число байтов, предназначенных для записи

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если вам надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Поэтому для работы с текстовыми файлами применяются другие классы.

В то же время при работе с различными бинарными файлами, имеющими определенную структуру FileStream может быть очень даже полезен для извлечения определенных порций информации и ее обработки.

### 48. Классы адаптеры потоков.

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен **System.Threading**. В нем определен класс, представляющий отдельный поток - класс **Thread**.

Класс Thread определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства класса:

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

Некоторые методы класса Thread:

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

**Получение информации о потоке**

Используем вышеописанные свойства и методы для получения информации о потоке:

using System.Threading;

....................

static void Main(string[] args)

{

    // получаем текущий поток

    Thread t = Thread.CurrentThread;

    //получаем имя потока

    Console.WriteLine("Имя потока: {0}", t.Name);

    t.Name = "Метод Main";

    Console.WriteLine("Имя потока: {0}", t.Name);

    Console.WriteLine("Запущен ли поток: {0}", t.IsAlive);

    Console.WriteLine("Приоритет потока: {0}", t.Priority);

    Console.WriteLine("Статус потока: {0}", t.ThreadState);

    // получаем домен приложения

    Console.WriteLine("Домен приложения: {0}", Thread.GetDomain().FriendlyName);

    Console.ReadLine();

}

**Приоритеты потоков**

Приоритеты поток располагаются в перечислении **ThreadPriority**:

* **Lowest BelowNormal Normal AboveNormal Highest** По умолчанию потоку задается значение Norma

### 49. 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.

**Атрибут Serializable**

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

[Serializable]

class Person

{

    public string Name { get; set; }

    public int Year { get; set; }

    public Person(string name, int year)

    {

        Name = name;

        Year = year;

    }

}

**Формат сериализации**

Хотя сериализация представляет собой преобразование объекта в некоторый набор байтов, но в действительности только бинарным форматом она не ограничивается. Итак, в .NET можно использовать следующие форматы:

* бинарный
* SOAP
* xml
* JSON

Для каждого формата предусмотрен свой класс: для сериализации в бинарный формат - класс **BinaryFormatter**, для формата SOAP - класс **SoapFormatter**, для xml - **XmlSerializer**, для json - **DataContractJsonSerializer**.

Для классов BinaryFormatter и SoapFormatter сам функционал сериализации определен в интерфейсе **IFormatter**.

Для бинарной сериализации применяется класс BinaryFormatter:

**Так как класс BinaryFormatter** определен в пространстве имен System.Runtime.Serialization.Formatters.Binary, то в самом начале подключаем его.

У нас есть простенький класс Person, который объявлен с атрибутом Serilizable. Благодаря этому его объекты будут доступны для сериализации.

Далее создаем объект BinaryFormatter: BinaryFormatter formatter = new BinaryFormatter();

Затем последовательно выполняем сериализацию и десериализацию. Для обоих операций нам нужен поток, в который либо сохранять, либо из которого считывать данные. Данный поток представляет объект FileStream, который записывает нужный нам объект Person в файл people.dat.

Сериализация одним методом formatter.Serialize(fs, person) добавляет все данные об объекте Person в файл people.dat.

При десериализации нам нужно еще преобразовать объект, возвращаемый функцией Deserialize, к типу Person: (Person)formatter.Deserialize(fs).

Как вы видите, сериализация значительно упрощает процесс сохранения объектов в бинарную форму по сравнению, например, с использованием связки классов BinaryWriter/BinaryReader.

 // объект для сериализации

            Person person = new Person("Tom", 29);

            Console.WriteLine("Объект создан");

            // создаем объект BinaryFormatter

            BinaryFormatter formatter = new BinaryFormatter();

            // получаем поток, куда будем записывать сериализованный объект

            using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

            {

                formatter.Serialize(fs, person);

                Console.WriteLine("Объект сериализован");

            }

            // десериализация из файла people.dat

            using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

            {

                Person newPerson = (Person)formatter.Deserialize(fs);

                Console.WriteLine("Объект десериализован");

                Console.WriteLine("Имя: {0} --- Возраст: {1}", newPerson.Name, newPerson.Age);

            }

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. Посмотрим на примере.

Прежде чем использовать класс SoapFormatter, нам надо добавить в проект сборку *System.Runtime.Serialization.Formatters.Soap.dll*. После этого нам станет доступным функциональность SoapFormatter:

/ создаем объект SoapFormatter

            SoapFormatter formatter = new SoapFormatter();

            // получаем поток, куда будем записывать сериализованный объект

            using (FileStream fs = new FileStream("people.soap", FileMode.OpenOrCreate))

            {

                formatter.Serialize(fs, people);

                Console.WriteLine("Объект сериализован");

            }

            // десериализация

            using (FileStream fs = new FileStream("people.soap", FileMode.OpenOrCreate))

            {

                Person[] newPeople = (Person[])formatter.Deserialize(fs);

                Console.WriteLine("Объект десериализован");

                foreach (Person p in newPeople)

                {

                    Console.WriteLine("Имя: {0} --- Возраст: {1}", p.Name, p.Age);

                }

            }

            Console.ReadLine();

**Принцип использования SoapFormatter** похож на рассмотренную в прошлой теме 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: Person[] newPeople = (Person[])formatter.Deserialize(fs)

После сериализации все данные будут сохранены в файл *people.soap*, который будет иметь следующее содержание:

**Для сериализации объектов в файлы xml** используется класс XmlSerializer. Он стоит несколько особняком от других ранее рассмотренных классов сериализаций, поэтому работа с ним будет немного отличаться.

Во-первых, XmlSerializer предполагает некоторые ограничения. Например, класс, подлежащий сериализации, должен иметь стандартный конструктор без параметров. Также сериализации подлежат только открытые члены. Если в классе есть поля или свойства с модификатором private, то при сериализации они будут игнорироваться.

// объект для сериализации

            Person person = new Person("Tom", 29);

            Console.WriteLine("Объект создан");

            // передаем в конструктор тип класса

            XmlSerializer formatter = new XmlSerializer(typeof(Person));

            // получаем поток, куда будем записывать сериализованный объект

            using (FileStream fs = new FileStream("persons.xml", FileMode.OpenOrCreate))

            {

               formatter.Serialize(fs, person);

                Console.WriteLine("Объект сериализован");

            }

            // десериализация

            using (FileStream fs = new FileStream("persons.xml", FileMode.OpenOrCreate))

            {

                Person newPerson = (Person)formatter.Deserialize(fs);

                Console.WriteLine("Объект десериализован");

                Console.WriteLine("Имя: {0} --- Возраст: {1}", newPerson.Name, newPerson.Age);

            }

            Console.ReadLine();

**Для сериализации объектов в формат JSON в пространстве** System.Runtime.Serialization.Json определен класс DataContractJsonSerializer. Чтобы задействовать этот класс, в проект надо добавить сборку *System.Runtime.Serialization.dll*. Для записи объектов в json-файл в этом классе имеется метод WriteObject(), а для чтения ранее сериализованных объектов - метод ReadObject(). Рассмотрим их применение.

        {            // объект для сериализации

            Person person1 = new Person("Tom", 29);

            Person person2 = new Person("Bill", 25);

            Person[] people = new Person[] { person1, person2 };

            DataContractJsonSerializer jsonFormatter = new DataContractJsonSerializer(typeof(Person[]));

            using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))

            {

                jsonFormatter.WriteObject(fs, people);

            }

            using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))

            {

                Person[] newpeople = (Person[])jsonFormatter.ReadObject(fs);

                foreach (Person p in newpeople)

                {

                    Console.WriteLine("Имя: {0} --- Возраст: {1}", p.Name, p.Age);

                }

            }

            Console.ReadLine();

        }

    }

}  
Чтобы пометить класс как сериализуемый, к нему применяется атрибут DataContract, а ко всем его сериализуемым свойствам - атрибут DataMember.

Метод WriteObject() принимает два параметра: файловый поток FileStream и объект, который надо сериализовать - в данном случае массив объектов Person. А метод ReadObject() принимает в качестве параметра файловый поток, который представляет файл в формате json.

### 50. Сериализация контрактов данных. интерфейс ISerializable.

Любой класс, который может быть сериализован, должен быть помечен атрибутом [SerializableAttribute](https://msdn.microsoft.com/ru-ru/library/system.serializableattribute(v=vs.110).aspx). Если класс должен управлять своим процессом сериализации, можно реализовать ISerializable интерфейса. [Formatter](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.formatter(v=vs.110).aspx) Вызовы [GetObjectData](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.iserializable.getobjectdata(v=vs.110).aspx) во время сериализации и заполняет предоставленный [SerializationInfo](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.serializationinfo(v=vs.110).aspx) со всеми данными, необходимыми для представления объекта. [Formatter](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.formatter(v=vs.110).aspx) Создает [SerializationInfo](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.serializationinfo(v=vs.110).aspx) с типом объекта в графе. Объекты, которые необходимо отправить прокси для самих можно использовать [FullTypeName](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.serializationinfo.fulltypename(v=vs.110).aspx) и [AssemblyName](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.serializationinfo.assemblyname(v=vs.110).aspx) методы [SerializationInfo](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.serializationinfo(v=vs.110).aspx) Изменение передаваемых данных.

В случае наследования класса является возможность сериализации класса, производного от базового класса, реализующего ISerializable. В этом случае производный класс должен вызывать реализацию базового класса [GetObjectData](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.iserializable.getobjectdata(v=vs.110).aspx) внутри своей реализации [GetObjectData](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.iserializable.getobjectdata(v=vs.110).aspx). В противном случае данные из базового класса не будут сериализованы.

ISerializable Интерфейс заключает в себе конструктор с помощью конструктора подпись ([SerializationInfo](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.serializationinfo(v=vs.110).aspx) сведения [StreamingContext](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.streamingcontext(v=vs.110).aspx) контекста). Во время десериализации текущий конструктор вызывается только после данных в [SerializationInfo](https://msdn.microsoft.com/ru-ru/library/system.runtime.serialization.serializationinfo(v=vs.110).aspx) десериализован с помощью модуля форматирования. Как правило этот конструктор нужно объявлять как protected, если класс не не объявлен как ненаследуемый (sealed).

Порядок, в котором объекты десериализуются, не может быть гарантирован. Например если один тип ссылается на тип, который не был десериализован, возникнет исключение. При создании типов, имеющих такие зависимости, можно устранить проблему, реализовав **IDeserializationCallback** интерфейс и **OnDeserialization** метод.

### 51. Атрибуты. Создание собственного атрибута.

В .NET атрибуты представляют собой типы классов, которые расширяют абстрактный базовый **класс System.Attribute**. В поставляемых в .NET пространствах имен доступно множество предопределенных атрибутов, которые полезно применять в своих приложениях. Более того, можно также создавать собственные атрибуты и тем самым дополнительно уточнять поведение своих типов, создавая для атрибута новый тип, унаследованный от 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. Например, атрибут может определять примечание, описывающее элемент, к которому присоединяется атрибут. Например:

**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.). В одних случаях подобное поведение оказывается именно тем, что нужно, но в других, однако, может потребоваться создать вместо этого специальный атрибут, действие которого бы распространялось только на избранные элементы кода.

Ниже представлен пример создания пользовательских атрибутов:

**using** System;

**using** System.Reflection;

**namespace** ConsoleApplication1

{

*// Создаем атрибут*

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Field)]

**public** sealed **class** UInfoAttribute : System.Attribute

{

**public** string Desc;

**public** **UInfoAttribute**() { }

**public** **UInfoAttribute**(string str)

{

Desc = str;

}

}

**class** Program

{

[UInfo(Desc = "Главный метод программы")]

**static** **void** **Main**()

{

Type t = typeof(UInfoAttribute);

object[] obj = t.GetCustomAttributes(**false**);

foreach (object o in obj)

Console.WriteLine(o);

Console.ReadLine();

}

}

}

### 52. Процесс. Домен приложений. Поток выполнения.

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 **Modules**: получает доступ к коллекции ProcessModuleCollection, которая хранит набор модулей (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
* Свойство **VirtualMemorySize64**: возвращает объем памяти, который выделен для данного процесса
* Метод **CloseMainWindow()**: закрывает окно процесса, который имеет графический интерфейс
* Метод **GetProcesses()**: возвращающий массив всех запущенных процессов
* Метод **GetProcessesByName()**: возвращает процессы по его имени. Так как можно запустить несколько копий одного приложения, то возвращает массив
* Метод **Kill()**: останавливает процесс
* Метод **Start()**: запускает новый процесс

Получим все запущенные процессы:

foreach(Process process in Process.GetProcesses())

{

    // выводим id и имя процесса

    Console.WriteLine("ID: {0}  Name: {1}", process.Id, process.ProcessName);

}

**Потоки процесса**

Свойство Threads представляет коллекцию потоков процесса - объект ProcessThreadCollection, каждый поток в которой является объектом **ProcessThread**. В данном классе можно выделить следующие свойства:

* **CurrentPriority**: возвращает текущий приоритет потока
* **Id**: идентификатор потока
* **IdealProcessor**: позволяет установить процессор для обработки потока
* **PriorityLevel**: уровень приоритета потока
* **StartAddress**: адрес в памяти функции, запустившей поток
* **StartTime**: время запуска потока

Запуск нового процесса

С помощью статического метода **Process.Start()** можно запустить новый процесс.

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.

Для управления домена платформа .NET предоставляет класс **AppDomain**. Рассмотрим некоторые основные методы и свойства данного класса:

* 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 **CreateDomain()**: статический метод, позволяющий создать новый домен приложения в рамках текущего процесса

Для создания нового потока используется делегат **ThreadStart**, который получает в качестве параметра метод, который мы определил выше. И чтобы запустить поток, вызывается метод **Start**.

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

Используя класс Thread, мы можем выделить в приложении несколько потоков, которые будут выполняться одновременно.

Во-первых, для запуска нового потока нам надо определить задачу в приложении, которую будет выполнять данный поток. Для этого мы можем добавить новый метод, производящий какие-либо действия.

Для создания нового потока используется делегат **ThreadStart**, который получает в качестве параметра метод, который мы определил выше.

И чтобы запустить поток, вызывается метод **Start**.

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**

Операционные системы не требуются для предоставления приоритета потока.

По умолчанию потоку задается значение Normal. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

***Статус потока***

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

В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение Unstarted. Запустив поток, мы изменим его статус на Running. Вызвав метод Sleep, статус изменится на WaitSleepJoin. А применяя метод Abort, мы тем самым переведем поток в состояние AbortRequested, а затем Aborted, после чего поток окончательно завершится.

### 54. Синхронизация потоков. Lock. Monitor. Мьютекс. Семафор

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. И мы можем переделать предыдущий пример следующим образом:

class Program

{

    static int x=0;

    static object locker = new object();

    static void Main(string[] args)

    {

        for (int i = 0; i < 5; i++)

        {

            Thread myThread = new Thread(Count);

            myThread.Name = "Поток " + i.ToString();

            myThread.Start();

        }

        Console.ReadLine();

    }

    public static void Count()

    {

        lock (locker)

        {

            x = 1;

            for (int i = 1; i < 9; i++)

            {

                Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

                x++;

                Thread.Sleep(100);

            }

        }

    }

}

Наряду с оператором lock для синхронизации потоков мы можем использовать мониторы, представленные классом **System.Threading.Monitor**. Фактически конструкция оператора lock из прошлой темы инкапсулирует в себе синтаксис использования мониторов. А рассмотренный в прошлой теме пример будет эквивалентен следующему коду:

class Program

{

    static int x=0;

    static object locker = new object();

    static void Main(string[] args)

    {

        for (int i = 0; i < 5; i++)

        {

            Thread myThread = new Thread(Count);

            myThread.Name = "Поток " + i.ToString();

            myThread.Start();

        }

        Console.ReadLine();

    }

    public static void Count()

    {

        try

        {

            Monitor.Enter(locker);

            x = 1;

            for (int i = 1; i < 9; i++)

            {

                Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

                x++;

                Thread.Sleep(100);

            }

        }

        finally

        {

            Monitor.Exit(locker);

        }

    }

}

Метод Monitor.Enter блокирует объект locker так же, как это делает оператор lock. А в блоке try...finally с помощью метода Monitor.Exit происходит освобождение объекта locker, и он становится доступным для других потоков.

Еще один инструмент управления синхронизацией потоков представляет класс **Mutex**, также находящийся в пространстве имен System.Threading. Данный класс является классом-оболочкой над соответствующим объектом ОС Windows "мьютекс". \ Сначала создаем объект мьютекса: Mutex mutexObj = new Mutex().

Основную работу по синхронизации выполняют методы **WaitOne()** и **ReleaseMutex()**. Метод mutexObj.WaitOne() приостанавливает выполнение потока до тех пор, пока не будет получен мьютекс mutexObj.

После выполнения всех действий, когда мьютекс больше не нужен, поток освобождает его с помощью метода mutexObj.ReleaseMutex()

Таким образом, когда выполнение дойдет до вызова mutexObj.WaitOne(), поток будет ожидать, пока не освободится мьютекс. И после его получения продолжит выполнять свою работу.

Мы использовали мьютекс для синхронизации потоков. Однако замечательная черта мьютексов состоит также в том, что они могут также применяться не только внутри одного процесса, но и между процессами.

Еще один инструмент, который предлагает нам платформа .NET для управления синхронизацией, представляют **семафоры**. Семафоры позволяют ограничить доступ определенным количеством объектов.

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

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

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

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**. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

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. Иными словами, TPL открывает возможности для автоматического масштабирования приложений с целью эффективного использования ряда доступных процессоров.

Благодаря этим двум особенностям библиотеки TPL она рекомендуется в большинстве случаев к применению для организации многопоточной обработки.

Еще одним средством параллельного программирования, внедренным в версию 4.0 среды .NET Framework, является **параллельный язык интегрированных запросов (PLINQ)**. Язык PLINQ дает возможность составлять запросы, для обработки которых автоматически используется несколько процессоров, а также принцип параллелизма, когда это уместно.

Как станет ясно из дальнейшего, запросить параллельную обработку запроса очень просто. Следовательно, с помощью PLINQ можно без особого труда внедрить параллелизм в запрос.

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

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();

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!"));

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

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

**Свойства класса Task**

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

**Создание продолжения задачи**

Одной из новаторских и очень удобных особенностей библиотеки TPL является возможность создавать продолжение задачи. Продолжение — это одна задача, которая автоматически начинается после завершения другой задачи. Создать продолжение можно, в частности, с помощью метода **ContinueWith()**, определенного в классе Task. Ниже приведена простейшая форма его объявления:

*public Task ContinueWith(Action<Task> действие\_продолжения)*

где действие\_продолжения обозначает задачу, которая будет запущена на исполнение по завершении вызывающей задачи. У делегата Action имеется единственный параметр типа Task. Следовательно, вариант делегата Action, применяемого в данном методе, выглядит следующим образом:

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

В данном случае обобщенный параметр T обозначает класс Task.

**Возврат значения из задачи**

Задача может возвращать значение. Это очень удобно по двум причинам. Во-первых, это означает, что с помощью задачи можно вычислить некоторый результат. Подобным образом поддерживаются параллельные вычисления. И во-вторых, вызывающий процесс окажется блокированным до тех пор, пока не будет получен результат. Это означает, что для организации ожидания результата не требуется никакой особой синхронизации.

Для того чтобы возвратить результат из задачи, достаточно создать эту задачу, используя обобщенную форму Task<TResult> класса Task. Ниже приведены два конструктора этой формы класса Task:

**public** **Task**(Func<TResult> функция)

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

где функция обозначает выполняемый делегат. Обратите внимание на то, что он должен быть типа Func, а не Action. 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 типа Object, передаваемый как состояние. Имеются также другие конструкторы данного класса.

Как и следовало ожидать, имеются также другие варианты метода StartNew(), доступные в обобщенной форме класса TaskFactory<TResult> и поддерживающие возврат результата из задачи. Ниже приведены те варианты данного метода, которые применяются параллельно с только что рассмотренными конструкторами класса Task:

public Task<**TResult**> StartNew(Func<**TResult**> функция)

public Task<**TResult**> StartNew(Func<**Object,** TResult> функция, Object состояние

Признак отмены является экземпляром объекта типа CancellationToken, т.е. структуры, определенной в пространстве имен System.Threading. В структуре CancellationToken определено несколько свойств и методов, но мы воспользуемся двумя из них. Во-первых, это доступное только для чтения свойство **IsCancellationRequested**. Оно возвращает логическое значение true, если отмена задачи была запрошена для вызывающего признака, а иначе — логическое значение false. И во-вторых, это метод ThrowIfCancellationRequested().

CancellationToken cancelTok = (CancellationToken)ct;

cancelTok.ThrowIfCancellationRequested();

Console.WriteLine("MyTask() №{0} запущен",Task.CurrentId);

**for** (**int** count = 0; count < 10; count++)

{

**if** (!cancelTok.IsCancellationRequested)

{

Thread.Sleep(500);

Console.WriteLine("В методе MyTask №{0} подсчет равен {1}", Task.CurrentId, count);

}

}

### 57. 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.

Одним из методов, позволяющих параллельное выполнение задач, является метод Invoke:

static void Main(string[] args)

{

    Parallel.Invoke(Display,

        () => {

                    Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

                    Thread.Sleep(3000);

                },

        () => Factorial(5));

    Console.ReadLine();

}

static void Display()

{

    Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

    Thread.Sleep(3000);

}

static void Factorial(int x)

{

    int result = 1;

    for (int i = 1; i <= x; i++)

    {

        result \*= i;

    }

    Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

    Thread.Sleep(3000);

    Console.WriteLine("Результат {0}", result);

}

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, но в данном случае мы определяем выполнение трех методов. Опять же как и в случае с классом Task мы можем передать либо название метода, либо лямбда-выражение.

И таким образом, при наличии нескольких ядер на целевой машине данные методы будут выполняться параллельно на различных ядрах.

**Parallel.For**

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.

**Parallel.ForEach**

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.

Здесь метод Factorial, обрабатывающий каждую итерацию, принимает дополнительный параметр - объект **ParallelLoopState**. И если счетчик в цикле достигнет значения 5, вызывается метод Break. Благодаря чему система осуществит выход и прекратит выполнение метода Parallel.For при первом удобном случае.

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

### 58. Асинхронные методы. async и await

Операторы async и await используются вместе для создания асинхронного метода. Такой метод, определенный с помощью модификатора async и содержащий одно или несколько выражений await, называется **асинхронным методом**.

Ключевое слово **async** указывает, что метод или лямбда-выражение являются асинхронными. А оператор **await** применяется к задаче в асинхронных методах, чтобы приостановить выполнение метода до тех пор, пока эта задача не завершится. При этом выполнение потока, в котором был вызван асинхронный метод, не прерывается.

Посмотрим на примере:

|  |
| --- |
| class Program  {      static void Main(string[] args)      {          DisplayResultAsync();          Console.ReadLine();      }        static async void DisplayResultAsync()      {          int num = 5;            int result = await FactorialAsync(num);          Thread.Sleep(3000);          Console.WriteLine("Факториал числа {0} равен {1}", num, result);      }        static Task<int> FactorialAsync(int x)      {          int result = 1;            return Task.Run(() =>          {              for (int i = 1; i <= x; i++)              {                  result \*= i;              }              return result;          });      }  } |

Итак, здесь определен метод, возвращающий объект Task - задачу, которая выполняется асинхронно. И так так мы получаем факториал целого числа, то объект задачи типизирован типом int.

Сам факториал мы получаем в асинхронном методе DisplayResultAsync. Асинхронным он является потому, что он объявлен с модификатором async и содержит использование ключевого слова await.

По негласным правилам в названии асинхроннных методов принято использовать суффикс *Async*.

* 1. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений
  2. Проектирование отношений. Агрегация, композиция и ассоциация

Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа.

Композиция определяет отношение **HAS A**, то есть отношение "имеет".

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

* 1. Стандартные интерфесы коллекций.

Основой для создания всех коллекций является реализация интерфейсов **IEnumerator** и **IEnumerable** (и их обобщенных двойников **IEnumerator<T>** и **IEnumerable<T>**). Интерфейс IEnumerator представляет перечислитель, с помощью которого становится возможен последовательный перебор коллекции, например, в цикле **foreach**. А интерфейс IEnumerable через свой метод GetEnumerator предоставляет перечислитель всем классам, реализующим данный интерфейс. Поэтому интерфейс IEnumerable (IEnumerable<T>) является базовым для всех коллекций.