Плывите сосиски (сосите плывиски))))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Вопросы

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

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.

**Операции преобразования типа**

* преобразует объект исходного класса в другой тип
* явная и неявна форма - будет ли этот алгоритм выполняться неявно или необходимо будет явным образом указывать необходимость соответствующего преобразования.

**implicit** operator *тип* ( *параметр* ) // неявное преобразование - преобразование вызывается автоматически.

**explicit** operator *тип* ( *параметр* ) // явное преобразование - преобразование вызывается в том случае, когда выполняется привидение типов

*тип* - тип, в который выполняется преобразование.

*параметр* - тип, который преобразуется.

Преобразуемые типы не должны быть связаны отношениями наследования.

Ключевые слова implicit и explicit в сигнатуру не включаются.

**Ограничения на операторах преобразования:**

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

class Counter

{

public static implicit operator Counter(int x)

{

return new Counter { Seconds = x };

}

public static explicit operator int(Counter counter)

{

return counter.Seconds;

}

}

Так как 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.

# ===22. Вложенные типы. Вложенные объекты

Тип, определенный внутри типа называется **вложенным типом:**

class Person {

public class Date {

public Date() { }

}

}

Person.Date today = new Person.Date();

**По умолчанию являются private.** Однако можно указать модификатор доступа, определяющий доступность вложенного типа:

* Вложенные типы класса могут иметь модификаторы public, protected, internal, protected internal, private или private protected
* Вложенные типы структуры могут иметь модификаторы public, internal или private.

Вложенный тип может получить доступ к внешнему 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.

**Вложенные объекты**

public class Date {

public Date() { }

}

class Person {

Date birthday;

}

Person anna = new Person();

anna.birthday = null;

Вложение или включение классов **модель включения-делегирования.**

**#region -** сворачивание и разворачивания блоков кода.

class Person {

#region Peson filed

public Date birthday;

#endregion

}

# ===23. Правила наследования C#.

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

Роль наследования:

* формирует иерархию
* поощряет повторное использование кода

Правила наследования:

* В C# наследование всегда подразумевается открытым
  + class Student : Person
* Запрещено множественное наследование классов (но не интерфейсов)
* Если класс объявлен с модификатором **sealed**, то от этого класса **нельзя наследовать** и создавать производные классы
* наследуются все свойства, методы, поля и т.д., которые есть в базовом классе
* Производному классу доступны public, internal, protected и protected internal члены базового класса (private – недоступны)
* не наследуются конструкторы базового класса
* тип доступа к производному классу должен быть таким же, как и у базового класса или более строгим
* Ссылке на объект базового класса можно присвоить объект производного класса (но вызываются для него только методы и свойства, определенные в базовом классе.)
* С помощью ключевого слова **base** мы можем обратиться к базовому классу:

class Person {

public string Name { get; set; }

public Person(string name) { Name = name; }

public void Display() { Console.WriteLine(Name); }

}

class Employee : Person {

public string Company { get; set; }

public Employee(string name, string company) : base(name)

{ Company = company; }

}

# ===24. Сокрытие имен при наследовании. Обращение к скрытым членам

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

class MyClass {

public int x = 10, y = 5, z = 6;

}

class ClassA : MyClass {

//скрываем члены класса MyClass

public new int x = 12, y = -2, z = -5;

}

**Применение ключевого слова base для доступа к скрытому имени**

Имеется ещё одна форма ключевого слова **base**, которая действует подобно ключевому слову this, за исключением того, что она всегда ссылается на базовый класс в том производном классе, в котором она используется: **base.член**, где член может обозначать метод или поле экземпляра. Эта форма ключевого слова base чаще всего применяется в тех случаях, когда под именами членов производного класса скрываются члены базового класса с теми же именами. С помощью base могут вызываться скрытые методы.

class MyClass {

public int x;

}

class ClassA : MyClass {

new int x = 10;

public void method(int i1, int i2) {

//Координата х из базового класса MyClass

base.x=i1;

Console.Writelene(base.x);

//Координата х из класса ClassA

x = 12;

Console.Writelene(x);

}

}

# ===25. Использование операций is и 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.

Оператор is никогда не генерирует исключение.

String derivedObj = "Dummy"; // derivedObj is Object - true

Object baseObjl = new Object (); // baseObjl is String - false

Object baseObj2 = derivedObj; // baseObj2 is String - true

int j = 123;

object boxed = j; // boxed is int - true; boxed is ValueType - true

object obj = new Object () ; // obj is int - false

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

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

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. Операнд as <тип>

Операция as выполняется:

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

class A {}

class B : A {}

public static void Main() {

A obj1 = new A();

B obj2 = obj1 as B; //null

B obj3 = new B();

A obj4 = obj3 as A;

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

**Полиморфизм** - ключевой аспект объектно-ориентированного программирования, способность к изменению функций, унаследованных от базового класса.

Простыми словами, полиморфизм – это различная реализация однотипных действий.

**Виртуальный метод** – это метод, который МОЖЕТ быть переопределен в классе-наследнике. Такой метод может иметь стандартную реализацию в базовом классе. Такие методы помечаются модификатором **virtual.**

Виртуальными могут быть: методы, свойства, индексаторы.

**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).

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

**Полиморфный интерфейс в базовом классе** - набор членов класса, которые могут быть переопределены в классе-наследнике.

virtual public void A\_method() { }

Переопределение виртуального метода в производном классе:

override public void A\_method() { }

Также можно запретить переопределение методов и свойств. В этом случае их надо объявлять с модификатором **sealed.** При создании методов с модификатором sealed надо учитывать, что sealed применяется в паре с override, то есть только в переопределяемых методах.

**Правила переопределения**

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

# ===27. Понятие раннего и позднего связывания.

Перегрузка методов обеспечивает статический полиморфизм, а виртуальный метод – динамический.

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

**Раннее связывание** – адрес функции назначается во время компиляции, и именно этот адрес используется при вызове функции.

**Достоинства:** высокое быстродействие получаемых выполнимых программ.

**Недостатки:** снижение гибкости программ.

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

**Достоинства:** высокая гибкость выполняемой программы, возможность реакции на события.

**Недостатки:** относительно низкое быстродействие программы.

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.

Виртуальные функции предоставляют механизм позднего или динамического связывания.

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

**Бесплодные (запечатанные) классы** - класс, от которого наследовать запрещается.

sealed class A: AAA

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

abstract class MyClass {}

**Абстрактные классы:**

* Служат только для порождения потомков, предоставляют базовый функционал для классов-наследников.
* Задают интерфейс для всей иерархии
* Могут содержать и полностью определенные методы, переменные, конструкторы, свойства
* Создавать объект абстрактного класса нельзя!!!!!!! (ссылку можно)

public abstract class Person {

public abstract void work();

}

public class Employee : Person {

public override void work() { //... }

}

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.

**Абстрактный метод** определяет полиморфный интерфейс, не имеют никакой реализации.

**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.

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

**Свойства abstract методов**

* абстрактные методы автоматически виртуальные (virtual не ставится)
* абстрактные методы не используются со static

public abstract class Person

{

public abstract void work();

}

private static void Main(string[] args)

{

Person anna = new Person(); //ошибка

}

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

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

}

# ===29. Структуры в C#.

Кроме базовых элементарных типов данных и перечислений в C# имеется и составной тип данных, который называется структурой.

**Структура** – это составной объект (пользовательский тип данных), совокупность логически связанных данных различного типа, объединенных под одним идентификатором.

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.

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.

Также можно 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;

Кроме обычных методов структура может содержать специальный метод - **конструктор**, который выполняет некую начальную инициализацию объекта, например, присваивает всем полям некоторые значения по умолчанию. Нельзя определить конструктор, используемый по умолчанию (конструктор без параметров). Он определяется для всех структур автоматически и не подлежит изменению В принципе для структуры не обязательно 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;

**Перечисление** – тип данных, чьё множество значений представляет собой ограниченный список идентификаторов.

**Тип перечисления** - целочисленный тип (byte, int, short, long). По умолчанию используется тип int:

enum MathOperation { Add , Subtract, Multiply, Divide }

Каждому элементу перечисления присваивается целочисленное значение, 0, 1 т.д. Можно определять явным образом:

enum MathOperation :short

{

Add = 4,

Subtract, //5

Multiply =9 , //9

Divide //10

}

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

**Интерфейсы** позволяют определить некоторый функционал, не имеющий конкретной реализации. Задается набор абстрактных методов, свойств, событий и индексаторов, которые должны быть реализованы в производных классах.

interface **I**имя\_интерфейса [ : предки]

Тело интерфейса [; ]

Интерфейс **не может содержать**: константы, поля, операции, конструкции, деструкторы, типы, любые статические элементы.

Интерфейс **может содержать:** абстрактные методы, обобщения свойства и индексаторы, события.

**Public по умолчанию.**

interface IDo {

void Go();

int Jump(int a);

int Energy {get; } //шаблон свойства

}

Назначение: задание общих характеристик объектов различных иерархий - навязывание контракта.

У интерфейса методы и свойства не имеют реализации, в этом они сближаются с абстрактными методами абстрактных классов. 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.

Интерфейс или класс может наследовать свойства нескольких интерфейсов, в этом случае предки перечисляются через запятую.

interface IDo {

void Go();

int Jump(int a);

int Energy {get; } //шаблон свойства

}

interface IKnow {

Void Count();

int Math();

}

interface IPosebel : IDo, IKnow { }