Оглавление

1. Состав. NET Framework. Структура среды выполнения CLR. 2
2. Структура управляемого модуля - portable executable (PE). Понятие и исполнение сборки. CIL. 3
3. CTS (Common Type System). Типы данных C#. Ссылочные и типы значений. 4
4. Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, null-объединение 5
5. Тип данных String: операции, литералы, пустые и нулевые строки, форматированный вывод. 5
6. Неявная типизация – назначение и использование. 6
7. Массивы C# одномерные, прямоугольные и ступенчатые. 7
8. Понятие кортежей. Свойства, создание 7
9. Принципы объектно-ориентированного программирования. 8
10. Класс. Элементы класса. Свойства и индексаторы. 8
11. Класс. Константы. Поля только для чтения. Инициализаторы класса. 9
12. Спецификаторы доступа C#. Видимость типов. Доступ к членам типов. 10
13. Класс. Конструкторы и их свойства. Деструкторы 10
14. Класс и методы System.Object. 11
15. Статические методы и статические конструкторы класса. 12
16. Статические классы. Методы расширения и правила их определения. 12
17. Анонимные типы. 13
18. Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы. 13
19. Перегрузка методов и операторов. Правила перегрузки операторов. 14
20. Операции преобразования типа. Явная и неявная форма. Ограничения. 15
21. Вложенные типы. Вложенные объекты 16
22. Правила наследования C#. 16
23. Сокрытие имен при наследовании. Обращение к срытым членам 18
24. Использование операций is и as 18
25. Полиморфизм. Виртуальные методы, свойства и индексаторы. Правила переопределения. 18
26. Понятие раннего и позднего связывания. 19
27. Абстрактные классы и методы. Бесплодные классы. 20
28. Структур в C#. 20
29. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов. 21
30. Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы. 21
31. Ковариантность интерфейсов. Контравариантность интерфейсов 22
32. Стандартные интерфейсы .NET. Назначение и применение. 22
33. Исключительные ситуации. Генерация и повторная генерация исключений. 22
34. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений 23

35. Обобщения (generics). Свойства обобщений. 23
36. Концепция ограничений обобщений. Статические члены обобщений. 24
37. Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты. 25
38. Анонимные функции. Лямбда-выражения. 25
39. Обобщённые делегаты .NET. Action, Func, Predicate 26
40. События и делегаты. 26
41. Стандартные коллекции .NET. Типы коллекций. 27
42. Стандартные интерфесы коллекций. 28
43. IEnumerable и IEnumerator 28
44. LINQ to Objects. Синтаксис. Форма. Возврат результата. Грамматика выражений запросов. Отложенные и неотлаженные операции. 29
45. LINQ to Objects. Операции Where, Select, Take, OrderB, Join, GroupBy. 30
46. Рефлексия. System Type. 30
47. Классы для работы с файловой системой. 31
48. Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы. 32
49. Классы адаптеры потоков. 33
50. Сериализация. Форматы сериализации. 33
51. Сериализация контрактов данных. интерфейс ISerializable 34
52. Атрибуты. Создание собственного атрибута 34
53. Процесс. Домен приложений. Поток выполнения. 35
54. Создание потоков , классы приоритетов. Состояния потоков 35
55. Синхронизация потоков. Lock. Monitor. Mutex. Semaphore 36
56. Библиотека параллельных задач TPL. Класс Task. Состояние задачи. 37
57. Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения. 37
58. Параллелизм при императивной обработке данных. Класс Parallel 38
59. Асинхронные методы. async и await 39
60. Проектирование отношений. Агрегация, композиция и ассоциация 39
61. Антипаттерны проектирования. Рефакторинг. Методы рефакторинга. 39
62. Чистый код. Требования к именам, функциям, форматированию. 40
63. Чистый код. Требования к классам и объектам. 40

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

.Net Framework – платформа для разработки и выполнения приложений

Состоит из: Компиляторы, CLR, библиотеки классов

CLR – виртуальная машина, обеспечивает выполнение сборки

Структура среды выполнения CLR:

1. JIT-компилятор (Just-In-Time Compiler):**

- Преобразует промежуточный байт-код в машинный код во время выполнения программы.

2. Garbage Collector (Сборщик мусора):**

- Отслеживает и удаляет объекты, которые больше не используются, освобождая ресурсы и предотвращая утечки памяти.

3. Исполняющая среда (Execution Engine):**

- Отвечает за выполнение инструкций в машинном коде, полученном от JIT-компилятора.

4. Common Type System (Общая система типов):**

- Определяет основные типы данных и операции, которые могут выполняться над этими типами в различных языках .NET.

5. Система управления исключениями:**

- Обеспечивает обработку исключений в .NET-приложениях.

8. Base Class Library (Базовая библиотека классов):**

- Содержит основные классы и функциональность, используемые всеми приложениями .NET.

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

заголовок PE (32/64),

заголовок CLR (версия CLR, точки входа модуля, размеры и месторасположение ресурсов и метаданных),

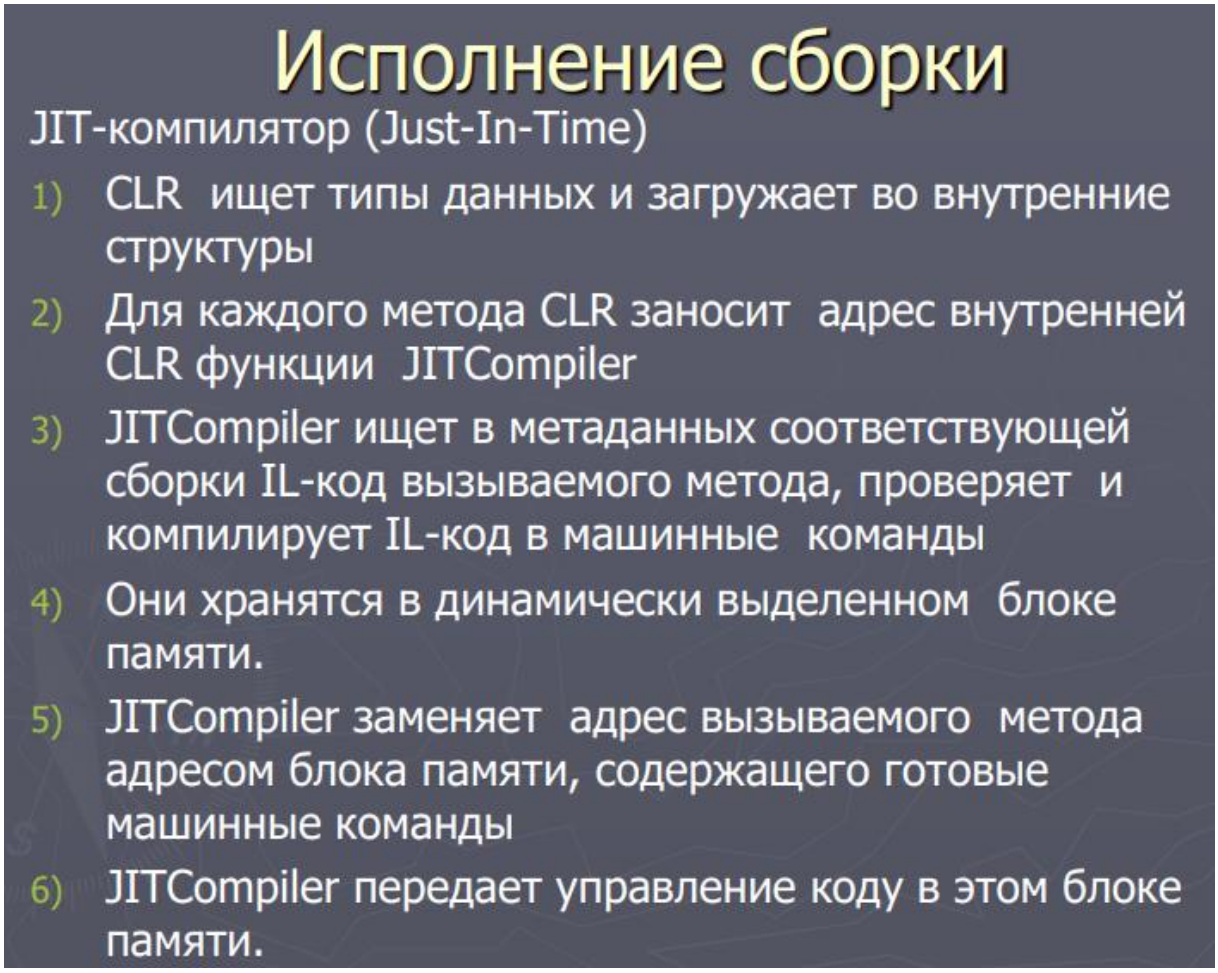
метаданные (специальные таблицы, содержащие исходный код типов и членов данных);

код IL (код который CLR компилирует в команды процессора).

- *Сборка* (assembly) — 1) это абстрактное понятие, для логической группировки одного или нескольких управляемых модулей или файлов ресурсов.
- 2) дискретная единица многократно используемого кода внутри CLR

CIL или же IL – промежуточный язык в котором компилируется исходный код

JIT-компиляция – процесс преобразования IL языка в машинный код во время выполнения программы



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

CTS – спецификация типов, которые должны поддерживаться всеми языками ориентированными на CLR

Основные и наиболее базовые типы данных, которые используются для хранения и обработки простых значений. String, bool, char, double, float, int byte, sbyte, long, ulong, short, ushort, uint

В переменных ссылочных типов хранятся ссылки на их данные (объекты). Тип object Тип string Классы (class) Интерфейсы (interface) Делегаты (delegate)

Целочисленные типы (byte, sbyte, short, ushort, int, uint, long, ulong) Типы с плавающей запятой (float, double) Тип decimal Тип bool Тип char Перечисления enum Структуры (struct)

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

Упаковка и распаковка это процессы, которые связаны с преобразование значимых типов в ссылочные и наоборот.

int intValue = 42;

object boxedValue = intValue; // Упаковка

object boxedValue = 42;

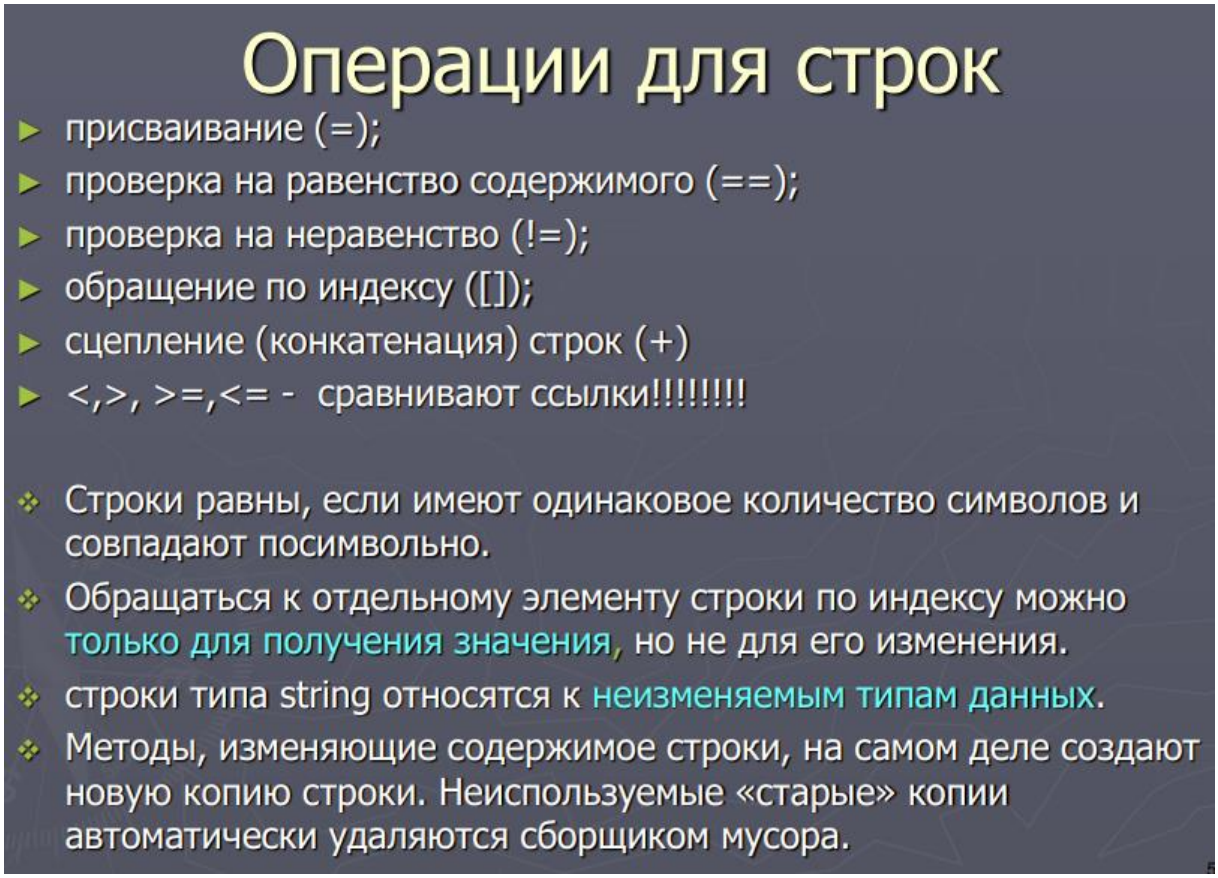
int intValue = (int)boxedValue; // Распаковка

Null показывает что переменная инициализирована но просто пустая или ее значение еще не определено, два значения.

Int? A = null;//нельзя null для int однако тут Nullable

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

String name = “Hello” - строковый литерал

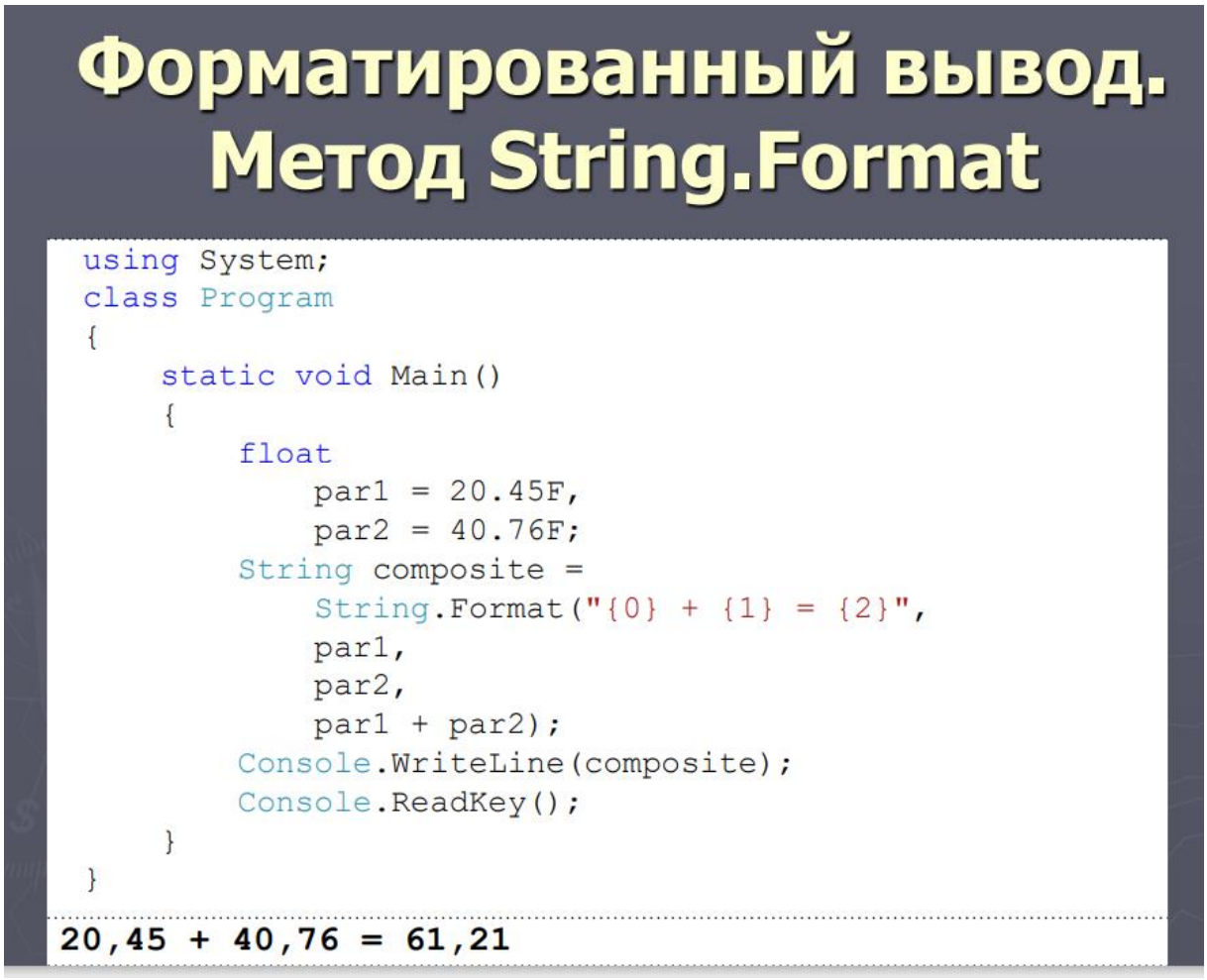


Отличие нулевой и пустой строки:

Пустая строка ("") может быть использована в качестве аргумента в методах

Пустая строка ("") является допустимой ссылкой на объект типа string.

Пустую строку можно проверить на равенство с помощью оператора ==



6. Неявная типизация – назначение и использование.

Неявная типизация в C# относится к возможности компилятора автоматически определять тип переменной на основе ее значения или контекста, в котором она используется. Это позволяет программистам писать более чистый и более лаконичный код, не явно указывая типы переменных.

В C# есть несколько сценариев, где используется неявная типизация:

1. **Оператор `var`:****

   var number = 42; // Компилятор определит тип как int

   var name = "John"; // Компилятор определит тип как string

2. **Динамический тип `dynamic`:****

Использование ключевого слова `dynamic` позволяет отложить определение типа переменной до времени выполнения.

```
dynamic variable = 10;

variable = "Hello";
```

Преимущества неявной типизации включают улучшение читаемости кода, уменьшение объема ненужной информации, более гибкую работу с анонимными типами и динамическими объектами. Однако, следует использовать неявную типизацию разумно, учитывая читаемость кода и предотвращая потенциальные проблемы.

Неявная типизация в программировании предоставляет возможность объявления переменных без явного указания их типов, оставляя компилятору определение типа переменной на основе контекста. Это может значительно улучшить читаемость кода, сделать его более лаконичным и уменьшить вероятность ошибок, связанных с несоответствием типов.

7. Массивы C# одномерные, прямоугольные и ступенчатые.

int[,] matrix = new int[,] – двумерный

int[,] matrix = new int[3, 3] – двумерный

int[] numbers = new int[5];

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

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

Двумерный массив - это массив, в котором элементы организованы в виде таблицы с двумя измерениями: строки и столбцы. Это массив массивов, то есть массив, элементами которого являются другие массивы.

```
int[][] jaggedArray = new int[3][];

jaggedArray[0] = new int[] { 1, 2, 3 };

jaggedArray[1] = new int[] { 4, 5 };

jaggedArray[2] = new int[] { 6, 7, 8, 9 };
```

8. Понятие кортежей. Свойства, создание

Кортеж – структура данных для хранения различных типов данных

var tuple = (1, "Hello", 3.14, true);

Доступ к элементам кортежа осуществляется по индексу:

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

Console.WriteLine(tuple.Item2); // Вывод: Hello

Именованные кортежи позволяют явно указать имена элементов кортежа, что делает код более читаемым.

var namedTuple = (id: 1, message: "Hello", pi: 3.14, flag: true);

Доступ к элементам кортежа осуществляется по именам:

Console.WriteLine(namedTuple.id); // Вывод: 1

Console.WriteLine(namedTuple.message);// Вывод: Hello

9. Принципы объектно-ориентированного программирования.

Наследование: Наследование позволяет создавать новые классы на основе существующих классов, позволяя наследникам наследовать свойства и методы родительского класса. Это позволяет повторно использовать код, улучшает расширяемость и обеспечивает иерархию классов с общими характеристиками.

Инкапсуляция: Инкапсуляция означает объединение данных и методов, работающих с этими данными, в единый объект. При инкапсуляции данные скрыты от прямого доступа извне, и доступ к ним осуществляется через методы объекта. Это обеспечивает контроль над доступом к данным и обеспечивает безопасность и целостность объекта.

Полиморфизм: Полиморфизм позволяет объектам одного типа проявлять различное поведение в зависимости от контекста. Это означает, что один и тот же метод может иметь различную реализацию в разных классах. Полиморфизм позволяет программисту работать с объектами разных классов с использованием общего интерфейса, что упрощает разработку гибкого и расширяемого кода.

Абстракция: Позволяет выделять какие нибудь компоненты необходимые для решения задач и скрыть их реализацию.

10. Класс. Элементы класса. Свойства и индексаторы.

Класс представляет собой шаблон или описание, определяющее структуру и поведение объектов.

Он определяет атрибуты (поля) и методы (функции), которые объекты класса могут иметь.

Элементы класса:

1. Поля класса: Переменные, которые хранят данные объекта. В приведенном выше примере `Name` - это поле.

2.Методы класса: Функциональные части класса, которые могут выполнять операции с данными объекта или выполнять другие действия. `PrintInfo` - это метод.

3.Свойства: Свойства обеспечивают способ доступа к полям класса, предоставляя гибкость в управлении чтением и записью данных.

4.Индексаторы: Индексаторы позволяют обращаться к объекту как к массиву. Они обеспечивают индексированный доступ к данным класса.

Автоматические свойства:

В C# существует сокращенный синтаксис для объявления свойств, называемый автоматическими свойствами. Если вам не требуется добавлять дополнительную логику в методы `get` и `set`, вы можете использовать следующий синтаксис:

```
class Person{

   // Автоматическое свойство для имени (Name)

   public string Name { get; set; }



   // Автоматическое свойство для возраста (Age)

   public int Age { get; set; }}
```

Индексаторы в C# представляют собой специальный вид свойств, которые позволяют обращаться к объекту с использованием синтаксиса массива. Индексаторы могут быть полезными, когда вы хотите предоставить доступ к элементам объекта по индексу, подобно тому, как это делается с массивами.

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

Класс представляет собой шаблон или описание, определяющее структуру и поведение объектов.

Он определяет атрибуты (поля) и методы (функции), которые объекты класса могут иметь.

В C#, константы - это поля, значения которых нельзя изменить после их определения. Они объявляются с использованием ключевого слова `const` и обычно представляют собой значения, которые известны на этапе компиляции:

public const double Pi = 3.14159;

Константы используются для хранения значений, которые не должны изменяться в течение времени выполнения программы, и их значения подставляются непосредственно в места использования во время компиляции.

Поля только для чтения объявляются с использованием ключевого слова `readonly`. Readonly определяется только внутри конструктора или класса и не может изменять из вне

Инициализатор класса:

```
class Person
{
   public string Name { get; set; }
   public int Age { get; set; }
   public Person(string name, int age)
   {
```

```
        Name = name;
        Age = age;
    }
}
Person person = new Person("Alice", 30);
```

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

В языке C# применяются следующие модификаторы доступа:

- **private**: закрытый или приватный компонент класса или структуры. Приватный компонент доступен только в рамках своего класса или структуры.
- **private protected**: компонент класса доступен из любого места в своем классе или в производных классах, которые определены в той же сборке.
- **file**: добавлен в версии C# 11 и применяется к типам, например, классам и структурам. Класс или структура с такми модификатором доступны только из текущего файла кода.
- **protected**: такой компонент класса доступен из любого места в своем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
- **internal**: компоненты класса или структуры доступен из любого места кода в той же сборке, однако он недоступен для других программ и сборок.
- **protected internal**: совмещает функционал двух модификаторов protected и internal. Такой компонент класса доступен из любого места в текущей сборке и из производных классов, которые могут располагаться в других сборках.
- Public: доступ не ограничен

13. Класс. Конструкторы и их свойства. Деструкторы

Конструктор (constructor) - это специальный метод в объектно-ориентированных языках программирования, который используется для инициализации объектов класса.

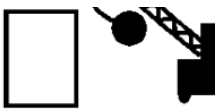
Конструкторы: по умолчанию, копирующий, с параметрами, без параметров, статический

по умолчанию: Это конструктор, который не принимает параметров.

Если в классе не определен конструктор, то автоматически создается конструктор по умолчанию, который не выполняет никаких дополнительных действий.

Если в классе определен хотя бы один конструктор, конструктор по умолчанию не будет создан автоматически.

Деструкторы



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

~имя_класса () { // код деструктора }

нельзя узнать, когда именно вызовется деструктор

Если программа завершиться до того, как произойдет "сборка мусора", деструктор может быть вообще не вызван

```
~Student()
        {
            Console.WriteLine("Объект уничтожен");
        }
```

Свойства деструктора

- Класс может иметь только один деструктор.
- Деструкторы не могут быть унаследованы или перегружены.
- Деструкторы невозможно вызвать. Они запускаются автоматически.
- Деструктор не принимает модификаторы и не имеет параметров.

14. Класс и методы System.Object.

Открытие методы System.Object:

Equals(Object) Определяет, равен ли указанный объект текущему объекту.

Equals(Object, Object) Определяет, считаются ли равными указанные экземпляры объектов.

GetHashCode() Служит хэш-функцией по умолчанию.

GetType() Возвращает объект Type для текущего экземпляра.

ReferenceEquals(Object, Object) Определяет, совпадают ли указанные экземпляры Object.

ToString() Возвращает строку, представляющую текущий объект.

Закрытые методы System.Object:

Finalize() Позволяет объекту попытаться освободить ресурсы и выполнить другие операции очистки, перед тем как он будет уничтожен во время сборки мусора.

15. Статические методы и статические конструкторы класса.

Статические методы в C# принадлежат классу, а не экземпляру объекта. Они вызываются через имя класса, а не через экземпляр объекта. Статические методы могут использоваться, например, для выполнения операций, не зависящих от конкретного состояния объекта.

Статический конструктор в C# используется для инициализации статических членов класса и выполняется только один раз при первом обращении к классу или статическим членам класса. Статические конструкторы вызываются автоматически перед любыми статическими методами или статическими полями в классе.

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

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

Статический класс отличается от обычного класса в нескольких аспектах:

1. Нельзя создавать экземпляры: Нельзя создать объект статического класса с помощью оператора `new`. Все его члены и методы могут вызываться непосредственно через имя класса, без создания объекта.

2. Нельзя наследовать от статического класса: Статический класс не может быть унаследован другими классами, и он не может наследовать от других классов, кроме статических классов.

3. Содержит только статические члены: Внутри статического класса все его члены (поля, свойства, методы) также должны быть статическими. Это означает, что они принадлежат типу, а не экземпляру.

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

```
public static class StringExtensions
{
    public static string AddExclamation(this string input)
    {
        return input + "!";
    }
```

```
}
```

В данном случае, метод AddExclamation расширяет функциональность класса string, добавляя метод AddExclamation. Он принимает строку input и возвращает эту строку с добавленным восклицательным знаком.

17. Анонимные типы.

Анонимные типы (Anonymous Types) в C# представляют собой специальный вид типов данных, которые позволяют создавать объекты с набором свойств без явного определения типа.

Создание анонимного типа происходит с использованием ключевого слова `new` и инициализатора объекта:

```
var person = new { Name = "John", Age = 30, IsStudent = false };
```

В данном примере создается анонимный тип с тремя свойствами: `Name`, `Age`, и `IsStudent`. Тип данных каждого свойства определяется автоматически на основе выражений инициализации.

Анонимные типы предоставляют удобный синтаксис для создания объектов с ограниченным временем жизни и использования в сценариях, где не требуется явное определение типа.

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

ref и out - это ключевые слова в C#, которые используются для передачи параметров в методы.

ref используется для передачи переменной в метод с возможностью изменения этой переменной внутри метода.

out также используется для передачи переменной в метод, но в отличие от ref, переменная не требует инициализации до передачи в метод. Метод с параметром out обязан присвоить значение перед выходом из метода.

```
public void ModifyValue(ref int value)
{
    value = value * 2;
}
public void GetValues(out int a, out int b)
{
    a = 10;
    b = 20;
```

```
}
```

`params` (Variable Number of Parameters):

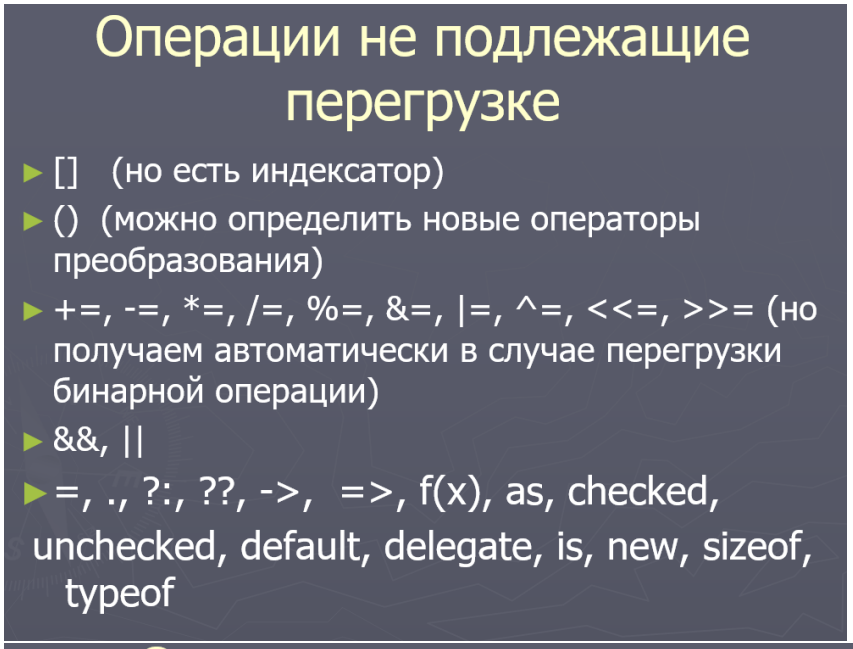
- Используется для передачи переменного числа параметров в метод. Все параметры должны иметь одинаковый тип.

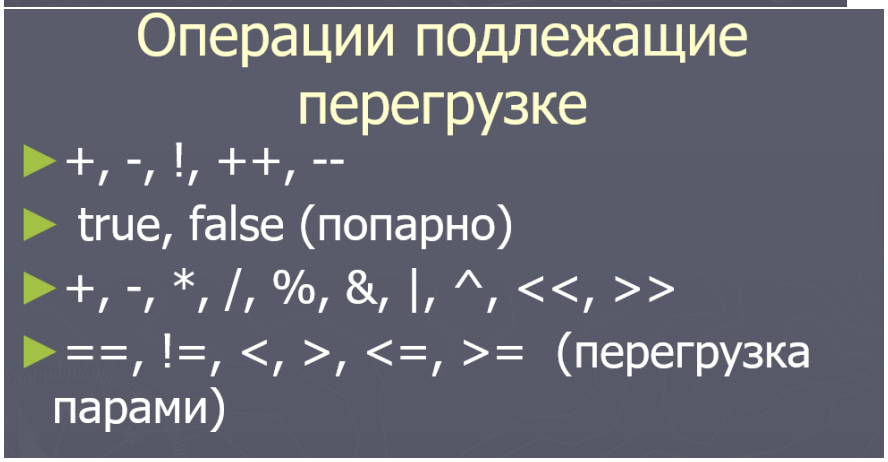
```
int Sum(params int[] numbers)
{
    int result = 0;
    foreach (int num in numbers)
    {
        result += num;
    }
    return result;
}
```

19. Перегрузка методов и операторов. Правила перегрузки операторов.

Назначение перегрузки операторов в языке программирования заключается в том, чтобы позволить программистам определять собственное поведение операторов для пользовательских типов данных.

Ключевое слово operator в языке программирования C# используется для определения и перегрузки операторов в пользовательских типах данных.





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

В C# предусмотрены два типа операций преобразования:

1**. Явное преобразование:**

Явное преобразование типов (Explicit Conversion): Происходит при помощи операторов приведения типов (cast), и требует указания явного преобразования в коде.

double myDouble = 10.5;

int myInt = (int)myDouble;

2. Неявное преобразование (Implicit Casting):

Неявное преобразование типов (Implicit Conversion): Происходит автоматически компилятором без необходимости указания явного преобразования в коде.

int myInt = 10;

double myDouble = myInt; // Неявное преобразование int в double

```
string strValue = "10";int intValue = Convert.ToInt32(strValue); класс
string strValue = "10";int intValue = int.Parse(strValue); метод
```

21. Вложенные типы. Вложенные объекты

Вложенные типы:

В языке программирования C#, вложенные типы (также известные как вложенные классы или вложенные структуры) представляют собой типы, объявленные внутри другого типа. Это могут быть классы, структуры, интерфейсы или перечисления, объявленные внутри другого класса или структуры.

Вложенные объекты:

В терминах программирования "вложенные объекты" обычно означают объекты, которые являются членами других объектов. В контексте C# это может включать в себя вложенные типы, но также может относиться к объектам, которые являются членами класса или структуры.

22. Правила наследования C#.

Наследование – это механизм получения нового класса на основе уже существующего

Роль наследования

- формирует иерархию
- поощряет повторное использование кода

Правила наследования:

1) В C# наследование всегда подразумевается открытым

```
class Student : Person
```

2) Запрещено множественное наследование классов (но не интерфейсов)

3) наследуются все свойства, методы, поля и т.д., которые есть в базовом классе

4) Производному классу доступны public, internal, protected и protected internal члены базового класса (private – недоступны)

5) не наследуются конструкторы базового класса

6) тип доступа к производному классу должен быть таким же, как и у базового класса или более строгим

```
internal class Машина { }

public class Грузовик :Машина{ }
```

7) Ссылке на объект базового класса можно присвоить объект производного класса (но вызываются для него только методы и свойства, определенные в базовом классе.)

```
class Person
        {
            public void buy() { }
        }
class Student : Person
        {
            public void study() { }
        }
```

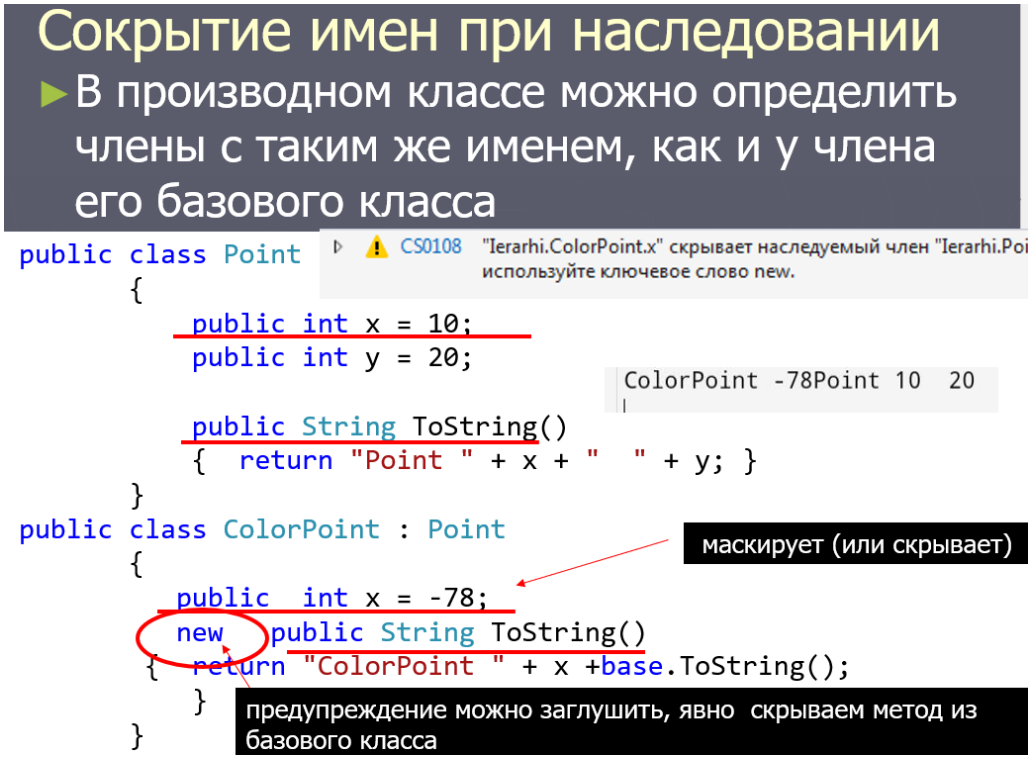
```
Person anna = new Person();
Person uman = new Student();

 anna.buy();
 uman.buy();
 uman.study();
```

23. Сокрытие имен при наследовании. Обращение к срытым членам

Сокрытие имен при наследовании (Hiding Members):

В языке C#, сокрытие имен (hiding names) при наследовании происходит, когда в производном классе создается новый член (метод или свойство) с тем же именем, что и у члена базового класса. В результате этого члена базового класса становятся недоступными без явного указания базового класса.



Обращение к скрытым членам - оператор `base` используется для вызова членов базового класса внутри производного класса.

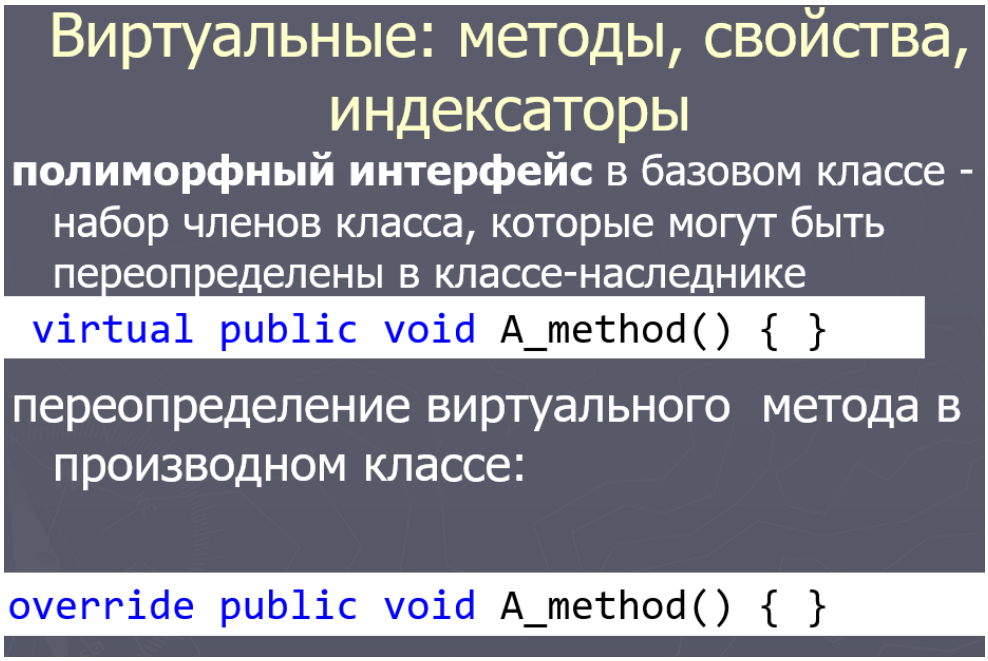
24. Использование операций is и as

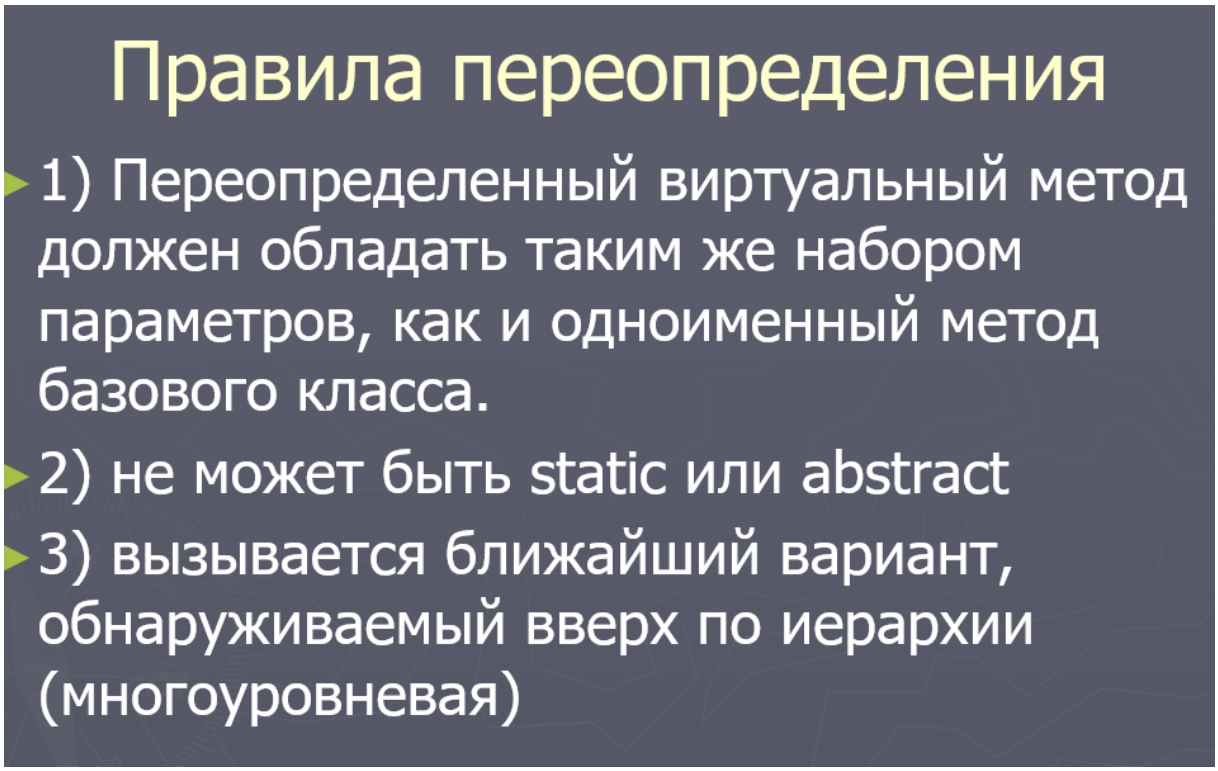
as и is - это операторы в языке C#, используемые для проверки и приведения типов. Оператор as используется для попытки явного приведения типа объекта к другому типу. Если приведение типа возможно, то оператор as возвращает объект с приведенным типом. Если приведение типа невозможно, то оператор as возвращает null, а не генерирует исключение. Например: var result = obj as MyClass; Оператор is используется для проверки, является ли объект экземпляром определенного типа. Он возвращает булевое значение true, если объект является экземпляром указанного типа, и false в противном случае. Например: if (obj is MyClass) { ... }

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

Полиморфизм: Полиморфизм позволяет объектам одного типа проявлять различное поведение в зависимости от контекста. Это означает, что один и тот же метод может иметь различную реализацию в разных классах. Полиморфизм позволяет программисту работать с объектами

разных классов с использованием общего интерфейса, что упрощает разработку гибкого и расширяемого кода.





26. Понятие раннего и позднего связывания.

Позднее связывание (dynamic binding) - это процесс определения вызываемого метода или свойства во время выполнения программы. В позднем связывании конкретный метод или свойство определяется по типу объекта, с которым происходит взаимодействие, во время выполнения программы.

Раннее связывание (early binding) - это процесс определения вызываемого метода или свойства во время компиляции программы. В раннем связывании конкретный метод или свойство определяются по типу переменной или выражения на этапе компиляции.

27. Абстрактные классы и методы. Бесплодные классы.

Ключевое слово abstract указывает на абстрактный класс. Абстрактные классы не могут быть инстанциированы, а только использованы в качестве базового класса для других классов. Пример: public abstract class MyAbstractClass { }.

Бесплодные классы, классы от которых запрещено наследоваться, ключевое слово sealed

Абстрактные методы:

Абстрактные методы не содержат тела и должны быть реализованы в производных классах. Класс, содержащий абстрактные методы, также должен быть абстрактным.

28. Структур в C#.

Структуры (structures) в C# - это типы данных, похожие на классы, которые могут содержать поля (variables) и методы (methods), но обладают рядом особенностей:

Значимый тип данных: Структуры являются значимыми типами данных (value types).

Подходят для хранения небольших объемов данных: Структуры часто используются для хранения небольших объемов данных

Не поддерживают наследование: В отличие от классов, структуры не поддерживают наследование друг от друга. Они не могут быть базовыми или производными друг от друга.

```
public struct Point
{
    public int X;
    public int Y;
    // Конструктор структуры
    public Point(int x, int y)
    {X = x;
     Y = y;}
    // Метод для изменения координат
    public void Move(int deltaX, int deltaY)
    {X += deltaX;
      Y += deltaY;}}
```

29. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов.

Интерфейс в C# - Интерфейс представляет ссылочный тип, который может определять некоторый функционал - набор методов и свойств без реализации. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

```
public interface IShape
{
    double Area { get; }  // Свойство для получения площади фигуры
}
```

Пример реализации интерфейса в классе:

```
// Пример класса, реализующего интерфейс
public class Circle : IShape
{
    public double Radius { get; set; }

    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Area
    {
        get { return Math.PI * Radius * Radius; }
    }
}
```

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

Неявная реализация интерфейса (Implicit Interface Implementation): Это стандартный подход к реализации интерфейсов, где методы интерфейса реализуются в классе без явного указания имени интерфейса. Это делается путем простого перечисления методов, определенных в интерфейсе, внутри класса.

Явная реализация интерфейса (Explicit Interface Implementation): Это способ реализации интерфейса, при котором методы интерфейса реализуются явно, указывая имя интерфейса перед именем метода. Это позволяет иметь две версии метода: одну для внутреннего использования классом и другую для использования интерфейсом.

31. Ковариантность интерфейсов. Контравариантность интерфейсов

Ковариантность и контравариантность в C# касаются не только интерфейсов, но также делегатов и обобщенных типов.

Ковариантность интерфейсов: если интерфейс определяет метод с возвращаемым типом, можно использовать в методе более производный тип, чем указано в интерфейсе.

Контравариантность интерфейсов: если интерфейс определяет метод с параметром, можно использовать в методе более базовый тип, чем указано в интерфейсе.

32. Стандартные интерфейсы .NET. Назначение и применение.

Интерфейсы служат важным инструментом для создания абстракции и обеспечения гибкости в приложениях. От их можно наследоваться и переопределять методы, которые находятся в интерфейсах.

ICloneable (System.ICloneable): Этот интерфейс определяет метод Clone(), который позволяет создать копию объекта.

IComparable (System.IComparable): Этот интерфейс определяет метод CompareTo(), который позволяет сравнить текущий объект с другим объектом того же типа.

IComparer (System.Collections.IComparer): Этот интерфейс определяет метод Compare(), который позволяет сравнивать два объекта для определения их относительного порядка.

IEnumerable (System.Collections.IEnumerable): Этот интерфейс определяет метод GetEnumerator(), который позволяет перебирать элементы коллекции последовательно.

33. Исключительные ситуации. Генерация и повторная генерация исключений.

Обычно система сама генерирует исключения при определенных ситуациях, например, при делении числа на ноль. Но язык C# также позволяет генерировать исключения вручную с помощью оператора throw. То есть с

помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения. Повторную генерацию исключения можно выполнить, перехватив исключение и затем снова вызвав throw.

Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция ***try...catch...finally***.

Try проверяет условие catch выводит ошибку finally выводит сообщение после блока catch throw выбрасывает ошибку.

34. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений

Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция ***try...catch...finally***.

Try проверяет условие catch выводит ошибку finally выводит сообщение после блока catch throw выбрасывает ошибку.

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

Для отлова любого возможного исключения в C# нужно использовать синтаксис: Блок catch с параметрами Exeption, отлавливающий исключения типа System.Exception. Также можно просто использовать catch{}, не указывая тип исключения.

Методы содержаться в классе Exception:

InnerException: хранит информацию об исключении, которое послужило причиной текущего исключения

Message: хранит сообщение об исключении, текст ошибки

Source: хранит имя объекта или сборки, которое вызвало исключение

StackTrace: возвращает строковое представление стека вызовов, которые привели к возникновению исключения

35. Обобщения (generics). Свойства обобщений.

В C# обобщение (generic) представляет собой механизм языка, который позволяет создавать обобщенные классы, интерфейсы и методы.
Обобщенные типы данных позволяют писать код, который может работать с различными типами данных, без необходимости повторного написания кода для каждого конкретного типа.

Из особенностей обобщения:

1) Ограничения на параметр

Для наложения определенного ограничения на параметр типа в обобщенном коде в C# можно использовать ключевое слово where. Ограничения позволяют указать, какие типы данных могут быть использованы в качестве аргументов для параметра типа. Чтобы наложить несколько ограничений на параметр типа в C#, вы можете перечислить их, используя запятую.

```
class Test<T> where T : class { }
```

```
class Test<T> where T : struct { }
```

2) Улучшенная производительность и безопасность типов

Обобщения позволяют избегать необходимости использования явных приведений типов и обеспечивают безопасность типов во время компиляции, уменьшая вероятность возникновения ошибок времени выполнения.

36. Концепция ограничений обобщений. Статические члены обобщений.

В C# обобщение (generic) представляет собой механизм языка, который позволяет создавать обобщенные классы, интерфейсы и методы. Обобщенные типы данных позволяют писать код, который может работать с различными типами данных, без необходимости повторного написания кода для каждого конкретного типа.

Для наложения определенного ограничения на параметр типа в обобщенном коде в C# можно использовать ключевое слово where. Ограничения позволяют указать, какие типы данных могут быть использованы в качестве аргументов для параметра типа. Чтобы наложить несколько ограничений на параметр типа в C#, вы можете перечислить их, используя запятую.

```
class Test<T> where T : class { }
```

```
class Test<T> where T : struct { }
```

В обобщенных классах в C# можно использовать статические переменные так же, как и в обычных классах. И она является общей для всех экземпляров класса. Это делает статические члены доступными без создания экземпляра класса и позволяет им обеспечивать общую функциональность для всех экземпляров.

```
Privet static int count;
```

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

Делегат — это тип данных, они используются для передачи методов в качестве аргументов другим методам.

Одним из вариантов использования делегатов является использование их с событиями event. Делегаты предназначены для оповещения других частей программы о том, что произошло какое-то событие. Событие обычно определяется в классе, а другие части программы могут подписаться на это событие, чтобы получать уведомления о его возникновении.

```
// Объявление делегата, который используется для события
public delegate void MyEventHandler(string message);

public class EventPublisher
{
    // Определение события с использованием делегата
    public event MyEventHandler SomeEvent;

    // Метод, который вызывает событие
    public void RaiseEvent(string message)
    {
        // Проверка, есть ли подписчики на событие, перед вызовом
        SomeEvent?.Invoke(message);
    }
}
```

Обобщенные делегаты в C# - это делегаты, которые могут работать с различными типами данных без явного указания типа данных в момент их объявления.

```
public delegate T MyGenericDelegate<T>(T arg);
```

38. Анонимные функции. Лямбда-выражения.

Анонимные функции в C# позволяют определять методы без явного объявления, то есть без указания имени метода. Они обычно используются там, где требуется передать функцию как аргумент в другой метод или для реализации простых функций без необходимости создания отдельного метода.

Существует несколько видов анонимных функций в C#:

1) Лямбда-выражения представляют упрощенную запись анонимных методов.

(string name, int age, string city) => $"My name is {name}, I'm {age} years old, living in {city}."

2) Анонимный метод

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

Обобщенные делегаты в C# - это делегаты, которые могут работать с различными типами данных без явного указания типа данных в момент их объявления.

```
public delegate T MyGenericDelegate<T>(T arg);
```

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

Func 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.

Predicate представляет метод, который принимает один аргумент определенного типа и возвращает булевское значение (true ил false). Обычно Predicate используется для проверки условий элементов в коллекции.

40. События и делегаты.

Делегат — это тип данных, они используются для передачи методов в качестве аргументов другим методам.

Делегаты предназначены для оповещения других частей программы о том, что произошло какое-то событие. Событие обычно определяется в классе, а другие части программы могут подписаться на это событие, чтобы получать уведомления о его возникновении.

```
// Объявление делегата, который используется для события
public delegate void MyEventHandler(string message);

public class EventPublisher
{
    // Определение события с использованием делегата
    public event MyEventHandler SomeEvent;

    // Метод, который вызывает событие
    public void RaiseEvent(string message)
    {
```

```
        // Проверка, есть ли подписчики на событие, перед вызовом
        SomeEvent?.Invoke(message);
    }
}
```

41. Стандартные коллекции .NET. Типы коллекций.

В среде .NET Framework поддерживаются пять видов 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".

Специальные коллекции Оперируют данными конкретного типа или же делают это каким-то особым образом.

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.

Обобщенные коллекции Обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связные списки, стеки, очереди и словари. Хранят определенный тип указанный

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

Основные типы коллекций:

LinkedList<T>: класс двухсвязанного списка.

HashSet <T>: представляет набор уникальных значений.

Dictionary<TKey, TValue>: класс коллекции, хранящей наборы пар "ключ-значение".

ConcurrentBag <Tkey, TValue>: Представляет потокобезопасную неупорядоченную коллекцию объектов.

Queue<T>: класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел").

Stack<T>: класс стека однотипных объектов.

SortedList<TKey, TValue>: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу.

ObservableCollection<T> - это обобщенная коллекция, которая представляет наблюдаемую коллекцию объектов, оповещающую об изменениях своего состояния.

ArrayList в языке программирования C# представляет динамическую коллекцию объектов переменного размера.

42. Стандартные интерфесы коллекций.

Стандартные интерфейсы коллекции в C# определяют общий функционал для коллекций определенного типа. Они предоставляют стандартные методы и свойства, которые ожидаются для использования с определенным типом коллекции.

Стандартные интерфейсы коллекций включают в себя такие типы как:

IEnumerable<T> - определяет метод GetEnumerator, с помощью которого можно получать элементы любой коллекции

IEnumerator<T> - определяет методы, с помощью которых потом можно получить содержимое коллекции по очереди

ICollection<T> - определяет методы для добавления, удаления и проверки элементов коллекции

IList<T> - предоставляет индексированный доступ к элементам коллекции

IDictionary<TKey, TValue> - определяет методы для работы с парами ключ-значение

43. IEnumerable и IEnumerator

IEnumerable<T>: определяет метод GetEnumerator, с помощью которого можно получать элементы любой коллекции

IEnumerator<T>: определяет методы, с помощью которых потом можно получить содержимое коллекции по очереди

IEnumerator позволяет перебирать элементы коллекции и предоставляет методы для перемещения по коллекции и доступа к текущему элементу.

IEnumerable определяет метод GetEnumerator(), который возвращает объект типа IEnumerator для возможности итерации по коллекции.

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

LINQ (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных.

```
string[] months = { "June", "July", "May", "December", "January", "August", "February", "September",
"November", "April", "October", "March" };
int Eight= 8;
var monthsWithLengthEight = months.Where(month => month.Length == Eight);
Console.WriteLine($"Месяцы с длиной строки равной {Eight}:");
            foreach (var month in monthsWithLengthEight){
              Console.WriteLine(month);}
```

Форма и возврат результата:

LINQ запросы возвращают результат в виде новой последовательности, полученной в результате применения операций к исходной коллекции.

Результатом может быть другая последовательность, объект или значение.

Грамматика выражений запросов:

Синтаксис запросов LINQ включает в себя ключевые слова, которые предоставляют возможность выполнения операций над данными. Основные ключевые слова запросов:

from: Определяет переменную диапазона и источник данных.

where: Фильтрует записи согласно определенному условию.

select: Определяет, какие данные должны быть выбраны из источника.

orderby: Сортирует записи по заданному критерию.

group by: Группирует записи по ключу.

Отложенные и немедленные операции:

LINQ операции делятся на отложенные (ленивые) и немедленные (жадные). Отложенные операции выполняются только при необходимости получения результата (например, Where, Select), а немедленные - немедленно вызывают запрос и выполняют его (например, ToList, ToArray, Count).

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

LINQ (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных.

```
string[] months = { "June", "July", "May", "December", "January", "August", "February", "September",
"November", "April", "October", "March" };
int Eight= 8;
var monthsWithLengthEight = months.Where(month => month.Length == Eight);
Console.WriteLine($"Месяцы с длиной строки равной {Eight}:");
          foreach (var month in monthsWithLengthEight){
            Console.WriteLine(month);}
```

where: Фильтрует записи согласно определенному условию.

select: Определяет, какие данные должны быть выбраны из источника.

orderby: Сортирует записи по заданному критерию.

group by: Группирует записи по ключу.

take: Получает определенное количество элементов из источника.

Join: Объединяет две коллекции по заданному ключу.

46. Рефлексия. System Type.

Рефлексия представляет собой процесс выявления типов во время выполнения приложения. Рефлексия в C# позволяет программам исследовать, анализировать и модифицировать свою структуру и поведение во время выполнения.

Класс System.Type в языке C# представляет метаданные типа данных во время выполнения программы. Он позволяет получать информацию о типе, такую как его имя, сборку, базовые классы, интерфейсы, свойства, методы и другие связанные с типом данные.

- Метод ***GetConstructors()*** возвращает все конструкторы данного типа в виде набора объектов ConstructorInfo
- Метод ***GetEvents()*** возвращает все события данного типа в виде массива объектов EventInfo
- Метод ***GetInterfaces()*** получает все реализуемые данным типом интерфейсы в виде массива объектов Type
- Метод ***GetMethods()*** получает все методы типа в виде массива объектов MethodInfo
- Свойство ***Name*** возвращает имя типа
- Свойство ***IsArray*** возвращает true, если тип является массивом

- Свойство ***IsClass*** возвращает true, если тип представляет класс
- Свойство ***IsEnum*** возвращает true, если тип является перечислением
- Свойство ***IsInterface*** возвращает true, если тип представляет интерфейс

47. Классы для работы с файловой системой.

Класс FileInfo, в отличие от File, является классом экземпляра. Он предоставляет более широкий набор методов и свойств для работы с файлами.

- File.Exists(path): Проверяет, существует ли файл по указанному пути.
- File.Copy(sourcePath, destinationPath): Копирует файл с исходного пути на указанный путь.
- File.Delete(path): Удаляет файл по указанному пути.
- File.WriteAllText(path, content): Записывает указанное содержимое в файл.

- fileInfo.CopyTo(destinationPath): Копирует файл в указанный путь.
- fileInfo.Delete(): Удаляет файл.
- fileInfo.OpenRead(): Открывает файл для чтения в виде потока.
- fileInfo.OpenWrite(): Открывает файл для записи в виде потока.
- fileInfo.Length: Возвращает размер файла в байтах.

Класс StreamReader считывает символы из потока байтов в определенной кодировке.

Класс StreamWriter для записи символов в поток в определенной кодировке.

Класс BinaryReader Считывает примитивные типы данных как двоичные значения в заданной кодировке.

Класс BinaryWriter Записывает примитивные типы в двоичный поток и поддерживает запись строк в заданной кодировке.

Класс *Path* предоставляет функциональность для обработки и манипулирования строками путей, включая операции соединения путей, получения имени файла, расширения файла и другие.

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

Конструкция using в C# применяется для работы с файловыми потоками и другими ресурсами, которые требуют явного освобождения после использования. Она помогает упростить и безопасно управлять ресурсами в коде и способствует повышению эффективности и надежности приложений.

```
var data = new DataModel { Name = "Alice", Age = 30 };

// Путь для сохранения файла JSON
string filePath = "data.json";

// Сериализация объекта в JSON
string jsonString = JsonSerializer.Serialize(data);

// Запись JSON в файл
File.WriteAllText(filePath, jsonString);
```

```
string filePath = "data.json";
// Чтение JSON из файла
string jsonString = File.ReadAllText(filePath);

// Десериализация JSON
DataModel data = JsonSerializer.Deserialize<DataModel>(jsonString);

// Использование полученных данных
Console.WriteLine($"Name: {data.Name}, Age: {data.Age}");
```

Потоковые классы:

1) FileStream:

Предоставляет возможность чтения и записи данных из/в файлы. Это базовый класс для работы с файловыми потоками.

2) StreamReader / StreamWriter:

Позволяют читать и записывать текст из/в потоки байтов, обеспечивая функции чтения/записи символов в текстовом формате.

3) BinaryReader / BinaryWriter:

Работают с примитивными типами данных, позволяя читать и записывать данные в двоичном формате.

49. Классы адаптеры потоков.

Классы адаптеры потоков (Stream Adapters) используются для преобразования одного типа потока в другой или для добавления дополнительной функциональности к существующим потокам.

1)BufferedStream:

Этот класс оборачивает другой поток и добавляет ему буферизацию.

2)CryptoStream:

Используется для шифрования или дешифрования данных, которые передаются через поток.

3)GZipStream / DeflateStream:

Классы для сжатия и распаковки данных, которые передаются через потоки. Они могут быть использованы для сжатия/распаковки данных. В пространстве имён System.IO.Compression

4)StreamReader / StreamWriter:

Предоставляют возможность работы с текстовыми данными через потоки байтов.

50. Сериализация. Форматы сериализации.

Сериализация— процесс перевода какой-либо структуры данных в последовательность битов. Обратной к операции сериализации является операция…

Десериализации— восстановление начального состояния структуры данных из битовой последовательности.

В .NET можно использовать следующие форматы:

- **бинарный** представляют собой файлы, содержащие данные в двоичном формате
- **SOAP** это протокол обмена структурированными данными, который использует XML для представления сообщений.
- **xml** — это язык разметки, используемый для хранения и передачи данных. Он состоит из элементов, которые образуют иерархическую структуру. Каждый элемент имеет открывающий и закрывающий тег, между которыми располагается содержимое элемента.

- **JSON** это текстовый формат, использующийся для представления структурированных данных. Он основан на двух структурах данных: объектах (набор пар ключ-значение, заключенных в фигурные скобки) и массивах (упорядоченный список значений, заключенных в квадратные скобки).

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

51. Сериализация контрактов данных. интерфейс ISerializable

Контракт данных - это спецификация, определяющая структуру и правила взаимодействия для передачи данных между различными компонентами или системами.

Сериализация контрактов данных - это процесс преобразования объектов, соответствующих контракту данных, в формат, который может быть передан или сохранен.

Интерфейс ISerializable является частью .NET и используется для контроля процесса сериализации объекта в поток байтов или его десериализации из потока. Он позволяет объектам контролировать свой собственный процесс сериализации и десериализации.

52. Атрибуты. Создание собственного атрибута

[OnDeserialized]	Этот атрибут позволяет указать метод, который будет вызван немедленно после десериализации объекта
[OnDeserializing]	Этот атрибут позволяет указать метод, который будет вызван перед процессом десериализации
[OnSerialized]	Этот атрибут позволяет указать метод, который будет вызван немедленно после того, как объект сериализован
[OnSerializing]	Этот атрибут позволяет указать метод, который будет вызван перед процессом сериализации

Создание собственного атрибута в C# позволяет добавлять дополнительные метаданные к вашему коду. Атрибуты могут использоваться для описания и

аннотации элементов программы, предоставляя информацию для компилятора, рефлексии, сериализации и других сценариев.

53. Процесс. Домен приложений. Поток выполнения.

Процесс - это экземпляр программы, который выполняется в операционной системе.
В .NET процесс представлен классом Process из пространства имен System.Diagnostics.

Домен - это изолированная и управляемая среда выполнения внутри процесса.

Домен создаётся использованием статического метода **AppDomain.CreateDomain()**.

Загрузить сборку в домен можно с помощью метода **AppDomain.Load()**.

С помощью метода **AppDomain.Unload()** можно производить избирательную выгрузку определенного домена приложения из обслуживающего процесса.

Поток (Thread) - последовательность выполняемых команд процессора.

Создать и настроить поток можно используя класс Thread из пространства имен System.Threading.

Чтобы запустить поток, вызывается метод **Start**.

Метод **Sleep** позволяет приостановить поток на промежуток времени, указанный как параметр данного метода.

54. Создание потоков , классы приоритетов. Состояния потоков

Поток (Thread) - последовательность выполняемых команд процессора.

Создать и настроить поток можно используя класс Thread из пространства имен System.Threading.

Чтобы запустить поток, вызывается метод **Start**.

Метод **Sleep** позволяет приостановить поток на промежуток времени, указанный как параметр данного метода.

Состояния потока:

1)Выполнение;

2)Ожидание - ожидания конца ввода-вывода;

3)Готовность - готов работать, но компьютер занят другим

Классы приоритетов:

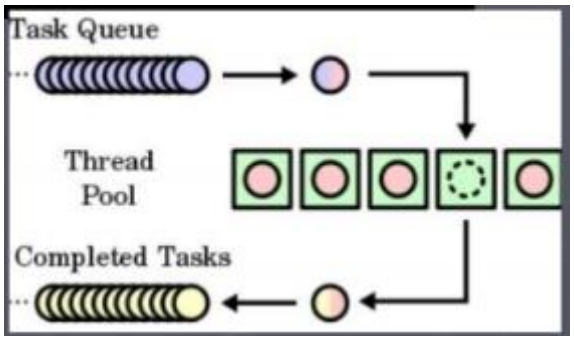
- Lowest;
- BelowNormal;
- Normal (по умолч.);
- AboveNormal;
- Highest;
- Real time.

55. Синхронизация потоков. Lock. Monitor. Mutex. Semaphore

Есть несколько основных средств синхронизации потока:

1) Неблокирующие средства синхронизации - это механизмы, используемые в многопоточных или распределенных системах для обеспечения согласованности и безопасности данных без блокировки или ожидания других потоков.

2) Mutex это средство синхронизации, используемое для контроля доступа к общим ресурсам в многопоточной среде.



WaitOne() – вход в критическую секцию

ReleaseMutex() – выход из нее (выход может быть произведен только в том же потоке выполнения, что и вход)

3) Semaphore, еще один инструмент, для управления синхронизацией, позволяющий войти в заданный участок кода не более чем N потокам (N – емкость семафора).

* класс System.Threading.Semaphore – между процессами

* класс SemaphoreSlim – в рамках одного процесса

* Wait() – получ. блокировки

* Release() – снятие блокировки

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

Библиотека параллельных задач TPL позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах, если на целевом компьютере имеется несколько ядер.

Класс Task упрощает написание кода для создания новых потоков.

```
1) Task task = new Task(() = > Console.WriteLine(“Hello!”));                    //3
task.Start()
2) Task task = Task.Factory.StartNew(() => Console.WriteLine(“Hello!”);                  //1
3) Task task = Task.Run(() => Console.WriteLine(“Hello!”);                    //2
```

В TPL (или при использовании async/await в C#), задача может находиться в одном из следующих состояний:

1)WaitingToRun (Ожидание запуска): Задача ожидает доступа к потоку для выполнения, но еще не начала исполнение.

2)Running (Выполнение): Задача находится в процессе выполнения на выделенном потоке.

3)Waiting (Ожидание): Задача временно приостановлена из-за ожидания завершения какой-то операции (например, ожидание завершения ввода/вывода, семафора и т.д.).

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

Класс Task упрощает написание кода для создания новых потоков.

```
1) Task task = new Task(() = > Console.WriteLine(“Hello!”));                    //3
task.Start()
```

2) Task task = Task.Factory.StartNew(() => Console.WriteLine(“Hello!”); //1

3) Task task = Task.Run(() => Console.WriteLine(“Hello!”); //2

Возврат результата:

```
int sum(int a,int b,int c){ return a + b + c;}
Task<int> result = new Task<int>(() => sum(rand_num1.Result, rand_num2.Result,
rand_num3.Result));
result.Start();
WriteLine($"Результат вычислений задачи {result.Id}:{result.Result} ");
```

Продолжения:

Задачи продолжения (continuation tasks) позволяют выполнить определенные действия после завершения другой задачи. Задача продолжения задается с помощью метода **ContinueWith**, который в качестве параметра принимает делегат **Action<Task>.**

```
Task<int> task1 = new Task<int>(() => Sum(4, 5));
Task task2 = task1.ContinueWith(sum => Display(sum.Result));
task1.Start();
```

58. Параллелизм при императивной обработке данных. Класс Parallel

System.Threading.Tasks.Parallel предоставляет удобные средства для параллельного выполнения итераций и операций в .NET. Он предназначен для упрощения разработки параллельных программ и повышения производительности путем распределения работы между несколькими ядрами процессора или потоками.

Одним из методов, позволяющих параллельное выполнение задач, является метод **Invoke.**

Метод **Parallel.For** позволяет выполнять итерации цикла параллельно.

Метод **Parallel.ForEach** осуществляет итерацию по коллекции, реализующей интерфейс IEnumerable, подобно циклу foreach, только осуществляет параллельное выполнение перебора.

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.

59. Асинхронные методы. async и await

Асинхонный метод обладает следующими признаками:

- В заголовке метода используется модификатор **async**
- Метод содержит одно или несколько выражений **await**
- В качестве возвращаемого типа используется один из следующих:
  - void
  - Task
  - Task<T>
  - ValueTask<T>

Асинхронный метод, как и обычный, может использовать любое количество параметров или не использовать их вообще. Однако асинхронный метод не может определять параметры с модификаторами **out** и **ref**.

Также стоит отметить, что слово **async**, которое указывается в определении метода, не делает автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений **await**.

60. Проектирование отношений. Агрегация, композиция и ассоциация

Ассоциация: Это отношение между двумя классами, которое описывает связь или взаимодействие между объектами этих классов. Она может быть однонаправленной или двунаправленной.

Агрегация: Это отношение "часть-целое", где один объект (целое) может содержать другие объекты (части). Части могут существовать независимо от целого.

Композиция: Это более строгий вид агрегации, при котором объекты-части существуют только как части целого и не могут существовать отдельно от него. При уничтожении целого объекта также уничтожаются все его части.

61. Антипаттерны проектирования. Рефакторинг. Методы рефакторинга.

Антипаттерны проектирования: Неблагоприятные подходы или решения в проектировании ПО, ведущие к проблемам, сложностям или низкому качеству кода.

Рефакторинг: Процесс улучшения структуры и читаемости кода путем изменения его внутренней структуры без изменения его внешнего поведения.

Методы рефакторинга: Небольшие, четко определенные изменения в коде, направленные на улучшение его структуры, читаемости и поддерживаемости, такие как выделение метода, объединение методов, извлечение переменной и другие.

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

Чистый код включает в себя ряд требований и принципов, направленных на создание читаемого, понятного и поддерживаемого кода:

Именование: Имена переменных, классов, функций и других элементов должны быть понятными и описательными. Используйте осмысленные имена, которые отражают суть объекта или операции.

Функции: Функции должны выполнять одну конкретную задачу и быть небольшими (следуйте принципу "одна функция - одна задача"). Избегайте длинных и сложных функций.

Форматирование: Поддерживайте последовательность форматирования для повышения читаемости. Используйте отступы, разделение кода на блоки, правильные отступы, пустые строки для логической раздельности.

63. Чистый код. Требования к классам и объектам.

Чистый код включает в себя ряд требований и принципов, направленных на создание читаемого, понятного и поддерживаемого кода:

Понятные имена: Имена переменных, классов и функций должны быть ясными и описательными, чтобы отражать, что они представляют или делают.

Функциональная специализация: Каждая функция или метод должны делать только одну важную вещь, чтобы они были понятными и маленькими

Читаемость и форматирование: Код должен быть удобочитаемым с использованием отступов, пустых строк, группировки логически связанных частей кода для улучшения его понимания.