**1. Что такое .Net Framework и из чего он состоит?**

.**NET Framework** - это программная платформа, разработанная компанией Microsoft, которая предоставляет среду выполнения и библиотеки для разработки и запуска различных типов приложений, основанных на технологии Microsoft.

NET Framework состоит из следующих основных компонентов:

1. **Common Language Runtime (CLR)**: CLR - это виртуальная машина, которая управляет выполнением кода, написанного на языках, совместимых с .NET. CLR обеспечивает множество функций, таких как управление памятью, безопасность типов, сборка мусора и т.д.
2. **Framework Class Library (FCL)**: FCL - это библиотека классов, предоставляющая широкий набор функций и типов для разработчиков. Она включает в себя классы для работы с файлами, сетью, базами данных, пользовательским интерфейсом и многими другими аспектами разработки приложений.

**2. Что такое CLR, FCL/BCL, CLI, IL?**

1. **CLR (Common Language Runtime)**:
   * CLR - это виртуальная машина, которая управляет выполнением кода, написанного на языках, совместимых с .NET (например, C#, VB.NET, F#). Она обеспечивает среду выполнения для приложений .NET и предоставляет такие функции, как управление памятью, безопасность типов, сборка мусора и другие.
   * CLR выполняет компиляцию кода на уровне байт-кода в момент его выполнения и преобразует его в нативный код для конкретной архитектуры процессора.
2. **FCL (Framework Class Library)**:
   * FCL - это библиотека классов, предоставляемая платформой .NET, которая содержит широкий набор классов, методов и типов, используемых для разработки приложений. Она включает в себя функциональность для работы с файлами, сетью, базами данных, пользовательским интерфейсом и многими другими задачами.
   * FCL предоставляет согласованный и стандартизированный набор API для разработчиков приложений .NET, что упрощает разработку и обеспечивает переносимость кода между различными приложениями и платформами.
3. **CLI (Common Language Infrastructure)**:
   * CLI - это стандарт, определенный ECMA (European Computer Manufacturers Association), который описывает общую инфраструктуру для выполнения и разработки кода на языках, совместимых с .NET.
   * CLI включает в себя спецификации для байт-кода (IL), метаданных, системы типов и других элементов, необходимых для создания многоплатформенных и многопарадигмальных приложений.
4. **IL (Intermediate Language)**:
   * IL - это промежуточный язык, на котором компилируется исходный код на языках, совместимых с .NET, перед его выполнением. IL - это ассемблероподобный язык, понятный CLR.
   * Когда приложение .NET компилируется, исходный код преобразуется в IL. Затем CLR выполняет JIT-компиляцию (Just-In-Time compilation), чтобы преобразовать IL в нативный код для конкретной платформы в момент выполнения приложения.

**3. Пояснить работу JIT-компилятора?**

**JIT (Just-In-Time) компилятор** - это часть Common Language Runtime (CLR), виртуальной машины, используемой в платформе .NET. JIT-компилятор выполняет важную роль в процессе выполнения приложений .NET. Его работа заключается в компиляции промежуточного кода (IL, Intermediate Language), который создается в результате компиляции исходного кода на языках, совместимых с .NET (например, C#, VB.NET), в нативный код для конкретной архитектуры процессора в момент выполнения приложения.

**4. Что такое CTS (Common Type System)?**

**CTS (Common Type System)** - это один из ключевых компонентов платформы .NET, который определяет систему типов, которую используют все языки, совместимые с .NET, для представления данных и объектов. CTS обеспечивает единообразное и согласованное представление типов данных, независимо от того, на каком языке программирования был написан код. Все языки программирования, совместимые с платформой .NET, должны соответствовать CTS, чтобы обеспечивать совместимость и возможность взаимодействия между различными языками. Это позволяет разработчикам выбирать наиболее подходящий язык для своих задач и интегрировать компоненты, написанные на разных языках, в единое приложение.

**5. Какие аспекты поведения определяет тип System.Object?**

Тип System.Object в языке C# и других языках, совместимых с платформой .NET, является корневым типом для всех других типов данных в .NET Framework и .NET Core. Он определяет некоторые основные аспекты поведения, которые применимы к всем типам, такие как:

1. **Уникальность идентификатора объекта**: Каждый объект в .NET имеет уникальный идентификатор (ID), который можно получить с помощью метода GetHashCode(). Этот идентификатор не изменяется в течение жизни объекта и используется, например, в хеш-таблицах для оптимизации поиска.
2. **Методы для сравнения объектов**: Объекты типа System.Object предоставляют два метода для сравнения: Equals() и ReferenceEquals(). Метод Equals() сравнивает содержимое объектов, в то время как метод ReferenceEquals() сравнивает ссылки на объекты.
3. **Методы для работы с хеш-кодами**: Объекты типа System.Object предоставляют методы GetHashCode(), который возвращает хеш-код объекта, и SetHashCode(), который позволяет задать хеш-код явным образом.
4. **Методы для преобразования в строку**: Объекты типа System.Object имеют метод ToString(), который возвращает строковое представление объекта. Этот метод можно переопределить в производных типах для предоставления более информативной строки.
5. **Метод для определения типа объекта**: Метод GetType() возвращает объект Type, представляющий тип объекта. Этот метод позволяет получить информацию о типе объекта во время выполнения (runtime).
6. **Методы для управления блоками финализации и освобождения ресурсов**: System.Object предоставляет методы Finalize() (финализация) и MemberwiseClone() (клонирование членов). Метод Finalize() может быть переопределен в производных типах для выполнения операций при завершении объекта.
7. **Методы для сериализации и десериализации**: System.Object предоставляет методы для поддержки сериализации и десериализации объектов, такие как GetObjectData() и SetObjectData(). Эти методы позволяют сохранять и восстанавливать состояние объекта.
8. **Методы для работы с блокировкой**: System.Object также предоставляет методы для работы с мониторами и блокировкой, такие как Monitor.Enter() и Monitor.Exit(), которые позволяют управлять доступом к ресурсам из нескольких потоков.

**7. Что такое «сборка»? Из чего состоит сборка .NET?**

***Сборка***(assembly) — это абстрактное понятие, для логической группировки одного или нескольких управляемых модулей или файлов ресурсов.

**8. Какие виды сборок существуют?**

**Сборка со строгими именами** (Strongly Named Assembly) и **сборка с нестрогими именами** (Weakly Named Assembly) отличаются следующим образом:

1. \*\*Уникальность идентификации:\*\*

- Сборка со строгими именами идентифицируется уникальным именем, включая версию, культуру и PublicKeyToken.

- Сборка с нестрогими именами идентифицируется только базовым именем.

2. \*\*Безопасность:\*\*

- Сборка со строгими именами обеспечивает безопасность и подлинность сборки, так как она подписывается цифровым ключом.

- Сборка с нестрогими именами не обязана быть подписанной и не обеспечивает такой же уровень безопасности.

3. \*\*Совместимость:\*\*

- Сборка со строгими именами обеспечивает контроль над версиями и совместимостью, что полезно в средах с разными версиями сборок.

- Сборка с нестрогими именами может вызвать проблемы совместимости и конфликты версий.

4. \*\*Использование:\*\*

- Сборки со строгими именами обычно используются в библиотеках и компонентах, предназначенных для повторного использования.

- Сборки с нестрогими именами могут быть удобными для простых проектов или маленьких приложений.

**Выбор между сборкой со строгими именами и нестрогими зависит от требований проекта к безопасности, управлению версиями и совместимости.**

**9. Что такое assembly manifest?**

**Assembly Manifest** (манифест сборки) - это часть метаданных, которая содержит важную информацию о сборке в платформе .NET. Манифест является неотъемлемой частью каждой сборки и содержит различные детали о сборке, включая следующие:

1. **Имя и версия сборки**: Манифест содержит информацию об имени сборки (assembly name) и ее версии. Это позволяет уникально идентифицировать сборку и управлять ее версионированием.
2. **Сведения о культуре**: Манифест может указывать информацию о культуре, связанной с сборкой. Это может быть полезно для локализации и поддержки разных языков и региональных настроек.
3. **Сведения о подписи (Strong Name)**: Манифест может содержать информацию о цифровой подписи сборки. Цифровая подпись обеспечивает безопасность и подтверждает подлинность сборки, что важно для обеспечения ее целостности и безопасности.
4. **Список всех файлов сборки**: Манифест включает список всех файлов, входящих в сборку, включая исполняемый код (assembly manifest, DLL), ресурсы, описание метаданных типов и т. д.
5. **Список зависимостей**: Манифест может содержать информацию о других сборках, которые требуются для корректной работы текущей сборки. Это включает в себя ссылки на другие сборки и их версии.
6. **Сведения о разрешении кода (Code Access Security)**: Манифест может определять требования к безопасности кода и уровни доверия, которые требуются для выполнения кода из сборки.
7. **Дополнительные метаданные**: Манифест может также содержать дополнительные метаданные, такие как описание сборки, краткое описание, автор, копирайты и т. д.

**10.Что такое GAC?**

**GAC** - это сокращение от "Global Assembly Cache" (глобальное кэширование сборок), и это специальное место в операционной системе Windows, предназначенное для хранения и управления общими (глобальными) сборками в платформе .NET. GAC предоставляет централизованное хранилище для сборок, которые могут использоваться различными приложениями на компьютере.

Обычно в GAC размещаются общие сборки, которые предназначены для использования множеством приложений на одном компьютере. Это позволяет избежать дублирования сборок в разных приложениях и обеспечивает централизованное управление общими ресурсами.

**11.Чем managed code отличается от unmanaged code**

Managed code и unmanaged code отличаются следующим образом:

1. \*\*Управление памятью:\*\*

- Managed code управляется средой выполнения (например, .NET Framework или Java Virtual Machine), которая автоматически управляет памятью, выделением и освобождением ресурсов. Это снижает риски утечек памяти и ошибок.

- Unmanaged code не зависит от среды выполнения и требует ручного управления памятью. Это может быть более гибким, но и более подверженным ошибкам и утечкам памяти.

2. \*\*Безопасность:\*\*

- Managed code обеспечивает дополнительный уровень безопасности благодаря проверкам типов, средствам автоматической проверки границ массивов и другим мерам безопасности, предотвращающим опасные операции.

- Unmanaged code более низкого уровня и может выполнять опасные операции напрямую без проверок, что может создавать уязвимости.

3. \*\*Портируемость:\*\*

- Managed code, как правило, более портируем, так как среда выполнения может абстрагировать приложение от аппаратной платформы.

- Unmanaged code может быть более зависимым от конкретной платформы и аппаратного обеспечения.

4. \*\*Производительность:\*\*

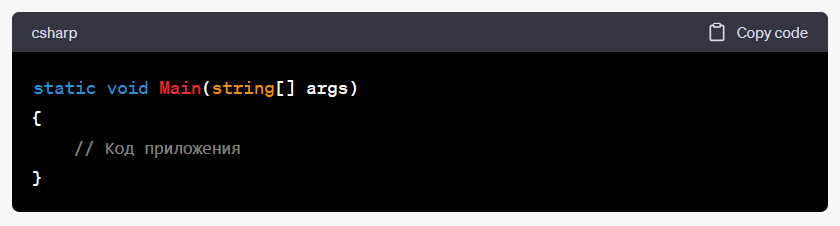
- Unmanaged code может быть более производительным, так как он ближе к аппаратному уровню и не имеет накладных расходов, связанных с управлением средой выполнения.

- Managed code может иметь некоторые накладные расходы из-за автоматического управления памятью и безопасности.

**Выбор между managed code и unmanaged code зависит от конкретных требований проекта, производительности, безопасности и удобства разработки.**

**12.Как и для чего определен метод Main?**

Метод **Main** - это точка входа (entry point) для исполняемого приложения на платформе .NET. В основном, он является статическим методом (то есть принадлежит классу, а не объекту), и его сигнатура (параметры и тип возвращаемого значения) обычно выглядит так:



Давайте разберемся, для чего определен метод **Main** и как он используется:

1. **Точка входа**: Метод **Main** служит как точка входа в исполняемое приложение. Когда вы запускаете приложение, операционная система вызывает метод **Main**, чтобы начать выполнение программы. Это означает, что код внутри метода **Main** будет выполнен первым.
2. **Аргументы командной строки**: Массив **string[] args** представляет аргументы командной строки, переданные приложению при его запуске. Эти аргументы могут быть использованы для передачи информации приложению, например, для указания опций, путей к файлам или других параметров.
3. **Тип возвращаемого значения**: Метод **Main** обычно имеет тип возвращаемого значения **void**, что означает, что он не возвращает результат при завершении. Однако вы также можете определить метод **Main** с типом **int**, чтобы вернуть код возврата при завершении выполнения приложения. Этот код может быть использован для указания результата выполнения (например, успешное завершение - 0, ошибка - другое значение).
4. **Исполнение кода**: Код внутри метода **Main** содержит логику вашего приложения. Здесь вы можете выполнять различные операции, взаимодействовать с пользователем, обрабатывать аргументы командной строки и так далее, в зависимости от целей вашего приложения.
5. **Завершение выполнения**: После завершения выполнения кода в методе **Main**, приложение завершает выполнение и возвращается к операционной системе. Если метод **Main** имеет тип возвращаемого значения **int**, то это значение будет использоваться как код возврата.

Метод **Main** является обязательным для каждого исполняемого приложения на платформе .NET. Этот метод определяет, с чего начнется выполнение вашего приложения, и служит входной точкой для основной логики программы.

**13.Варианты использования директивы using( using Directive ) в C#.**

Директива **using** в C# имеет несколько вариантов использования, и каждый из них выполняет определенную функцию. Вот основные варианты использования директивы **using**:

1. **Импорт пространства имен (Namespace Import)**:



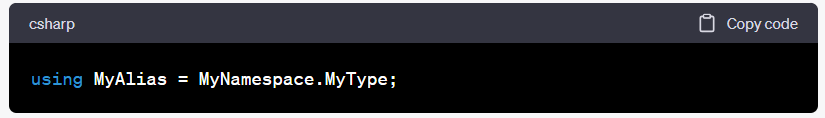
В этом случае, **using** используется для импорта пространства имен **System**, что позволяет использовать типы и члены из этого пространства имен без явного указания имени пространства имен. Например, после такой директивы вы можете использовать **Console.WriteLine(...)** вместо **System.Console.WriteLine(...)**. Это упрощает код и делает его более читаемым.

1. **Импорт нескольких пространств имен (Multiple Namespace Imports)**:



Вы можете использовать несколько директив **using**, чтобы импортировать несколько пространств имен одновременно. Это удобно, если вам нужно использовать типы из разных пространств имен в одном файле кода.

1. **Использование псевдонима типа (Type Alias)**:



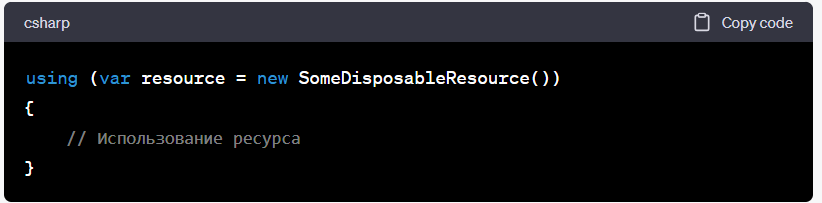
Вы можете использовать директиву **using** для создания псевдонимов для типов. Это полезно, когда у вас есть типы с длинными именами, и вы хотите сделать код более читаемым, создав краткие псевдонимы.

1. **Использование директивы using static (Static Type Import)**:



Этот вариант позволяет импортировать статические члены типа, такие как методы и константы, напрямую в файл кода без необходимости указания имени типа. Например, после такой директивы вы можете использовать **Abs(...)** вместо **Math.Abs(...)**.

1. **Использование директивы using с ресурсами (Resource Cleanup)**:



Этот вариант **using** используется для автоматического управления ресурсами, которые реализуют интерфейс **IDisposable**. Когда блок **using** завершается, он гарантирует вызов метода **Dispose()** на объекте **resource**, что позволяет освободить ресурсы (например, закрыть файл или соединение с базой данных).

Эти различные варианты использования директивы **using** обеспечивают удобство и читаемость кода, а также помогают в управлении пространствами имен, типами и ресурсами в C#.

**14.Как связаны между собой сборки и пространства имен?**

Сборки (assemblies) и пространства имен (namespaces) - это два разных, но взаимосвязанных понятия в платформе .NET. Давайте рассмотрим, как они связаны между собой:

1. **Сборки**:
   * Сборка представляет собой логическую единицу кода, которая может содержать один или несколько типов, ресурсы и другие файлы, необходимые для работы приложения.
   * Сборка обычно является физическим файлом (например, .dll или .exe), который может быть загружен и выполнен средой выполнения .NET (CLR).
   * Сборка содержит метаданные о типах и ресурсах, а также информацию о зависимостях и версии.
   * Сборки позволяют управлять версионированием, безопасностью и разделением кода между разными приложениями.
2. **Пространства имен**:
   * Пространство имен представляет собой организацию кода в именованные логические группы. Оно используется для структурирования кода и предотвращения конфликтов имен.
   * Пространства имен позволяют создавать иерархию и именовать типы, что делает код более организованным и читаемым.
   * Пространства имен могут быть вложенными, и они используются для организации типов и разграничения их области видимости.

**Связь между сборками и пространствами имен заключается в том, что сборки могут содержать типы, а типы могут находиться в пространствах имен. Когда вы создаете сборку, вы можете включить в нее один или несколько типов, и каждый тип должен находиться в каком-то пространстве имен.**

**15.Что такое примитивные типы данных? Перечислите их.**

**Примитивные типы данных** (primitive data types), также известные как базовые типы данных, представляют собой фундаментальные типы данных, которые предоставляются в языке программирования для представления основных значений. Они обладают фиксированным размером в памяти и обычно выполняются быстро и с минимальным использованием ресурсов. Примитивные типы данных используются для хранения чисел, символов, булевых значений и других простых данных. Вот некоторые из наиболее распространенных примитивных типов данных:

1. **Целочисленные типы (Integer Types)**:
   * **int**: 32-битное целое число со знаком.
   * **long**: 64-битное целое число со знаком.
   * **short**: 16-битное целое число со знаком.
   * **byte**: 8-битное целое число без знака.
   * **sbyte**: 8-битное целое число со знаком.
2. **Типы с плавающей запятой (Floating-Point Types)**:
   * **float**: 32-битное число с плавающей запятой одинарной точности.
   * **double**: 64-битное число с плавающей запятой двойной точности.
   * **decimal**: 128-битное число с фиксированной запятой для высокой точности десятичных чисел.
3. **Символьный тип (Character Type)**:
   * **char**: 16-битный символ Unicode.
4. **Логический тип (Boolean Type)**:
   * **bool**: Логическое значение, может быть **true** (истина) или **false** (ложь).
5. **Байтовый тип (Byte Type)**:
   * **byte**: 8-битное целое число без знака, используется для хранения байтовых данных.

**16.Что такое ссылочные типы? Какие типы относятся к ним?**

**Ссылочные типы** (reference types) в языках программирования представляют собой типы данных, которые хранят ссылки на объекты в памяти, а не сами данные. Когда вы работаете с ссылочным типом, вы обращаетесь к объекту по ссылке на него, а не напрямую к самим данным. Ссылочные типы обладают некоторой дополнительной сложностью, связанной с управлением памятью и жизненным циклом объектов.

1. **Классы (Classes)**:
2. **Интерфейсы (Interfaces)**:
3. **Делегаты (Delegates)**:
4. **Массивы (Arrays)**:
5. **Строки (Strings)**:
6. **Nullable-типы (Nullable Types)**:

Ссылочные типы обычно имеют более сложное поведение в сравнении с примитивными типами данных, так как они управляются динамически и требуют управления памятью, освобождения ресурсов и т. д. Однако они позволяют создавать более сложные структуры данных и объектно-ориентированный код, что является одной из основных особенностей современных языков программирования.

**17.Какие типы относятся к типам-значениям?**

**Типы-значения** (value types) в языках программирования представляют собой типы данных, которые хранят сами значения, а не ссылки на объекты. Это означает, что переменные, содержащие типы-значения, хранят собственные данные непосредственно в стеке или внутри других объектов. Типы-значения обычно более эффективны с точки зрения использования памяти и производительности, чем ссылочные типы.

Вот некоторые из наиболее распространенных типов-значений:

1. **Целочисленные типы (Integer Types)**:
   * **int**: 32-битное целое число со знаком.
   * **long**: 64-битное целое число со знаком.
   * **short**: 16-битное целое число со знаком.
   * **byte**: 8-битное целое число без знака.
   * **sbyte**: 8-битное целое число со знаком.
2. **Типы с плавающей запятой (Floating-Point Types)**:
   * **float**: 32-битное число с плавающей запятой одинарной точности.
   * **double**: 64-битное число с плавающей запятой двойной точности.
   * **decimal**: 128-битное число с фиксированной запятой для высокой точности десятичных чисел.
3. **Символьный тип (Character Type)**:
   * **char**: 16-битный символ Unicode.
4. **Логический тип (Boolean Type)**:
   * **bool**: Логическое значение, может быть **true** (истина) или **false** (ложь).
5. **Перечисления (Enums)**:
   * Перечисления позволяют создавать новые типы-значения, которые представляют собой набор именованных констант. Например, **enum Season { Spring, Summer, Autumn, Winter }**.
6. **Структуры (Structures)**:
   * Структуры (structs) представляют собой пользовательские типы-значения, которые могут содержать поля и методы. Они используются для создания компактных объектов.
7. **Nullable-типы (Nullable Types)**:
   * Nullable-типы позволяют переменным принимать значение **null** в дополнение к их обычным значениям. Например, **int?** или **double?**.
8. **Типы-значения для дат и времени (Date and Time Value Types)**:
   * В языке C#, существуют типы-значения для работы с датами и временем, такие как **DateTime**, **TimeSpan**, и **DateTimeOffset**.

**18. В чем отличие между ссылочными и значимыми типами данных?**

Отличие между ссылочными (reference types) и значимыми (value types) типами данных заключается в том, как они хранят и обрабатывают данные. Вот основные различия:

1. **Хранение данных**:
   * **Ссылочные типы**: Ссылочные типы хранят ссылки (адреса) на объекты в памяти. Переменные, содержащие ссылочные типы, хранят адреса объектов, а не сами объекты. Это означает, что несколько переменных могут ссылаться на один и тот же объект в памяти.
   * **Значимые типы**: Значимые типы хранят сами данные, а не ссылки на них. Переменные, содержащие значимые типы, хранят фактические значения данных. Каждая переменная значимого типа содержит свою копию данных.
2. **Передача аргументов**:
   * **Ссылочные типы**: Передача ссылочных типов в качестве аргументов функций или методов передает ссылку на оригинальный объект. Изменения, сделанные в объекте через одну переменную, отразятся на этом объекте, к которому обращаются другие переменные, которые ссылаются на него.
   * **Значимые типы**: Передача значимых типов в качестве аргументов функций или методов передает копию значения. Изменения, сделанные внутри функции, не влияют на оригинальное значение переменной, из которой была передана копия.
3. **Присваивание значений**:
   * **Ссылочные типы**: Присваивание одной переменной ссылочного типа другой переменной копирует ссылку, но не создает новую копию объекта.
   * **Значимые типы**: Присваивание одной переменной значимого типа другой переменной создает независимую копию значения.
4. **Поведение в памяти**:
   * **Ссылочные типы**: Объекты ссылочных типов хранятся в куче (heap), а переменные содержат ссылки на них.
   * **Значимые типы**: Значимые типы обычно хранятся в стеке (stack) или внутри других объектов и имеют фиксированный размер в памяти.

**19.Что такое упаковка и распаковка значимых типов?**

**Упаковка (boxing) и распаковка (unboxing)** - это операции, которые связаны с преобразованием значимых типов данных в ссылочные типы и обратно. Эти операции позволяют работать с значимыми типами как с объектами и использовать их в контексте, где ожидаются ссылочные типы. Важно понимать, что упаковка и распаковка могут повлиять на производительность вашего кода, поэтому их следует использовать осторожно.

**20.В чем заключается разница между int и System.Int32? double и  
System.Double и т.д.?**

В том что int это псевдоним для System.Int32 (компилятор автоматически подрузумевает это, и переделывает)

**21.Для чего используется тип dynamic?**

Тип **dynamic** в языке программирования C# используется для создания переменных, которые могут хранить данные разных типов во время выполнения (время выполнения, а не времени компиляции). Это означает, что переменные с типом **dynamic** обходят статическую типизацию, которая обычно характерна для C#. Вместо этого проверка типов выполняется во время выполнения кода. Тип **dynamic** предоставляет более гибкий способ работы с данными, но за счет утраты некоторых преимуществ статической типизации.

**22.В чем заключается главное отличие между var и dynamic?**

**var** и **dynamic** - это два разных ключевых слова в языке C#, которые оба используются для более гибкого определения типов переменных, но они имеют важные различия:

**var**:

* + **var** используется для определения переменных с неявным типом данных во время компиляции (время компиляции).
  + Переменные, объявленные с **var**, имеют статическую типизацию: тип переменной определяется компилятором на основе выражения, к которому переменная присваивается.
  + Однажды определенный тип переменной с **var** не может изменяться.

**dynamic**:

* **dynamic** используется для определения переменных с динамическим типом данных во время выполнения (время выполнения).
* Переменные, объявленные с **dynamic**, обходят статическую типизацию и тип проверяется только во время выполнения кода.
* Тип переменной с **dynamic** может изменяться в процессе выполнения.

**Главное отличие между var и dynamic заключается в моменте определения типа переменной:**

* **var** определяет тип переменной на этапе компиляции и обеспечивает статическую проверку типов во время компиляции.
* **dynamic** позволяет отложить определение типа до времени выполнения и предоставляет динамическую проверку типов.

Использование **var** подразумевает статическую типизацию и сохраняет преимущества статической проверки типов и подсказок IDE. В то время как **dynamic** используется, когда необходимо обойти статическую типизацию и работать с данными динамически, например, при взаимодействии с данными из динамических источников.

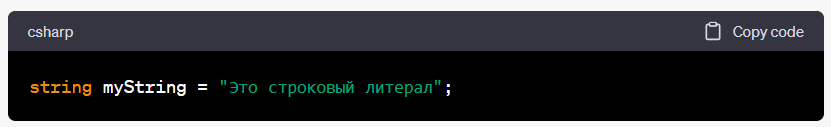
**23.Что такое неявно типизированная переменная?**

**Неявно типизированная переменная** (implicitly typed variable) в C# обозначается ключевым словом **var** и используется для объявления переменных, у которых тип данных определяется компилятором на основе выражения, к которому они присваиваются. В других словах, тип переменной вычисляется неявно из выражения, исходя из типа данных выражения, при этом сама переменная остается типизированной.

**24.Для чего используют Nullable тип?**

Nullable тип (или Nullable значение) в C# используется для представления значений, которые могут иметь как конкретное значение, так и значение **null**. Это особенно полезно для типов-значений (value types), таких как целые числа (**int**, **double**, и так далее), которые не могут содержать **null** по умолчанию.

**25.Как объявить строковый литерал? Какие операции можно  
выполнять со строкой?**



**Строка в C#** - это последовательность символов, и в языке предоставляется множество операций для работы с ней. Вот некоторые из наиболее часто используемых операций:

**Конкатенация строк**:

* + Вы можете объединять строки с помощью оператора **+** или метода **string.Concat**.

**Длина строки**:

* Чтобы узнать длину строки, используйте свойство **Length**

**Сравнение строк**:

* Для сравнения строк используйте методы **string.Compare** или операторы сравнения (**==**, **!=**, **<**, **>**, **<=**, **>=**).

**Извлечение подстроки**:

* Метод **Substring** позволяет извлекать подстроки из исходной строки.

**Поиск подстроки**:

* Методы **IndexOf** и **LastIndexOf** используются для поиска индекса первого и последнего вхождения подстроки.

**Замена подстроки**:

* Метод **Replace** позволяет заменить одну подстроку на другую.

**Разделение строки**:

* Метод **Split** разделяет строку на массив подстрок на основе заданного разделителя.

**Преобразование регистра**:

* Методы **ToUpper** и **ToLower** используются для преобразования символов в верхний или нижний регистр.

**Интерполяция**

**Форматирование строки**:

* С помощью метода **string.Format** или интерполяции строк (начиная с C# 6.0) можно создавать отформатированные строки.

1. **Другие операции**:
   * Есть множество других методов и операций для работы со строками, таких как удаление начальных и конечных пробелов (**Trim**), проверка начала и конца строки (**StartsWith**, **EndsWith**), и многое другое.

Строки в C# являются неизменяемыми (immutable), что означает, что любая операция, модифицирующая строку, создает новую строку. Строковые операции могут быть неэффективными при выполнении большого количества операций над строками, поэтому важно использовать методы оптимальным образом в зависимости от конкретных задач.

**26.Какие есть способы для задания и инициализации строк?**

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string s1 = "hello";  string s2 = new String('a', 6); // результатом будет строка "aaaaaa"  string s3 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' });  string s4 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' }, 1, 3); // orl    Console.WriteLine(s1);  // hello  Console.WriteLine(s2);  // aaaaaaa  Console.WriteLine(s3);  // world  Console.WriteLine(s4);  // orl |

**27.Какие методы есть у типа String?**

Основная функциональность класса String раскрывается через его методы, среди которых можно выделить следующие:

* **Compare**: сравнивает две строки с учетом текущей культуры (локали) пользователя
* **CompareOrdinal**: сравнивает две строки без учета локали
* **Contains**: определяет, содержится ли подстрока в строке
* **Concat**: соединяет строки
* **CopyTo**: копирует часть строки, начиная с определенного индекса в массив
* **EndsWith**: определяет, совпадает ли конец строки с подстрокой
* **Format**: форматирует строку
* **IndexOf**: находит индекс первого вхождения символа или подстроки в строке
* **Insert**: вставляет в строку подстроку
* **Join**: соединяет элементы массива строк
* **LastIndexOf**: находит индекс последнего вхождения символа или подстроки в строке
* **Replace**: замещает в строке символ или подстроку другим символом или подстрокой
* **Split**: разделяет одну строку на массив строк
* **Substring**: извлекает из строки подстроку, начиная с указанной позиции
* **ToLower**: переводит все символы строки в нижний регистр
* **ToUpper**: переводит все символы строки в верхний регистр
* **Trim**: удаляет начальные и конечные пробелы из строки
* **Объединение строк**
* Конкатенация строк или объединение может производиться как с помощью операции +, так и с помощью метода **Concat**.

**28.В чем отличие пустой и null строки?**

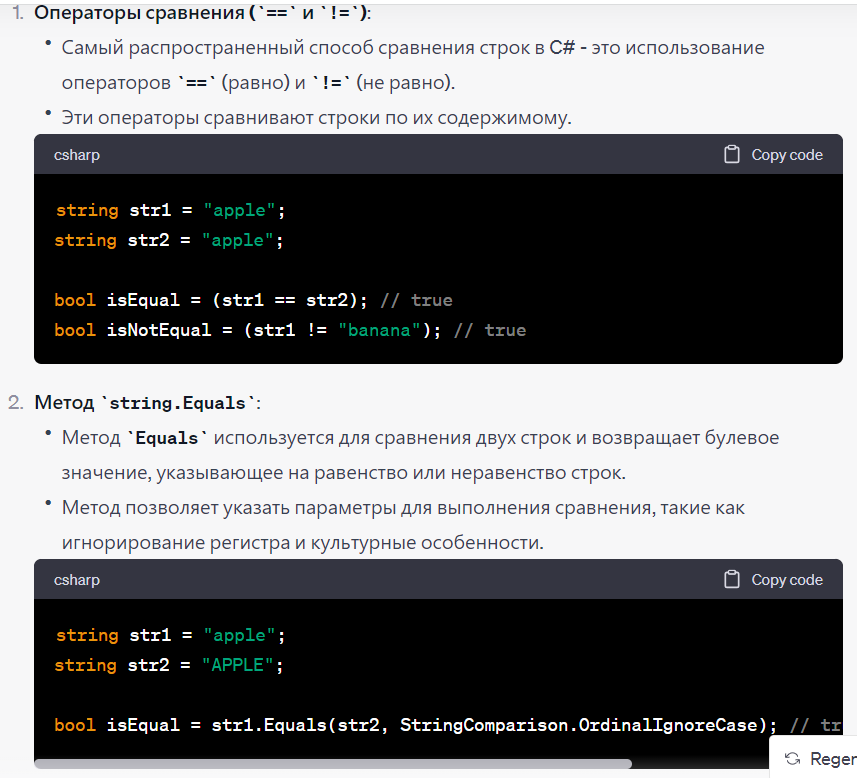
В C# есть два понятия: пустая строка и строка со значением **null**, и они имеют различное значение и семантику:

1. **Пустая строка (Empty String)**:
   * Пустая строка представляет собой строку, которая содержит нулевое количество символов. Фактически, это строка, в которой нет символов вообще.
   * В C#, пустую строку можно создать, используя строковый литерал **""** или вызывая метод **string.Empty**.
   * Пустая строка не равна **null**; она представляет собой строку с нулевой длиной, но она существует как объект типа **string**.

**Строка со значением null (Null String)**:

* Строка со значением **null** представляет собой строку, которая не имеет ссылки на объект строки. Она не указывает на какой-либо объект в памяти.
* В C#, строка со значением **null** обозначается как **null**.
* Это означает, что переменная типа **string**, содержащая **null**, не имеет ссылки на объект строки и не содержит данных.

**29.Как можно выполнить сравнение строк?**



**30.В чем отличие типов String и StringBuilder?**

StringBuilder - динамическая строка, используется чаще чем System.String когда нужно обработать больше строк и больше методов над ними.

**31.Поясните явные преобразования переменных с помощью команд  
Convert.**

При явных преобразованиях (**explicit conversion**) мы сами должны применить операцию преобразования (операция ()). Суть операции преобразования типов состоит в том, что перед значением указывается в скобках тип, к которому надо привести данное значение.

Класс **Convert** представляет еще один способ для преобразования значений. Для этого в нем определены следующие статические методы:

* **ToBoolean(value)**
* **ToByte(value)**
* **ToChar(value)**
* **ToDateTime(value)**
* **ToDecimal(value)**
* **ToDouble(value)**
* **ToInt16(value)**
* **ToInt32(value)**
* **ToInt64(value)**
* **ToSByte(value)**
* **ToSingle(value)**
* **ToUInt16(value)**
* **ToUInt32(value)**
* **ToUInt64(value)**

В качестве параметра в эти методы может передаваться значение различных примитивных типов, необязательно строки.

**32.Как выполнить консольный ввод/вывод?**

Console.Write() (or WriteLine)

Console.Read() (or ReadLine)

**33.Приведите примеры определения и инициализации одномерных и  
двумерных массивов.**

Двумерный:

int[,] dbArray;

dbArray = new int[3, 3]

{

{1, 2 ,3},

{4, 5, 6},

{7, 8, 9}

};

Одномерный:

var undStr = new[]{ "Hello", "World", "!" };

**34.Что такое ступенчатый массив? Как его задать?**

**Ступенчатый массив** представляет собой массив массивов, в котором длина каждого массива может быть разной.

Пример:

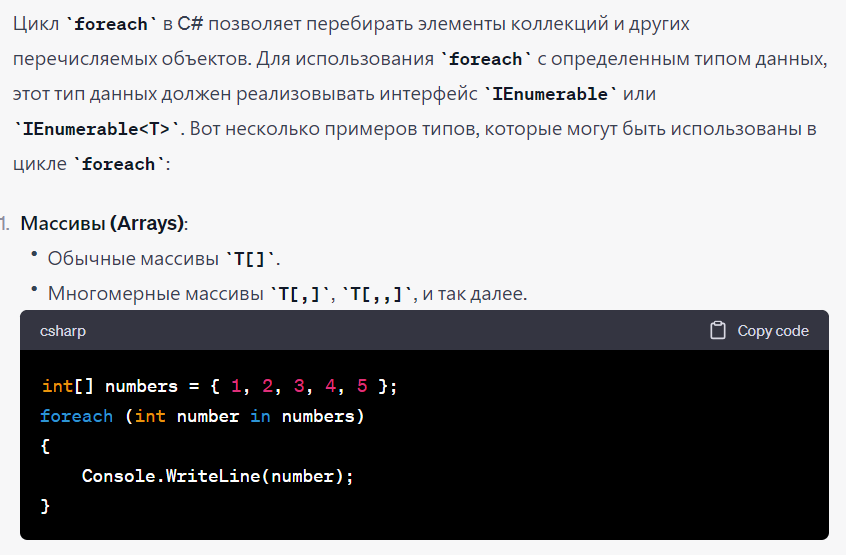
int[][] stairsArray = new int[3][];

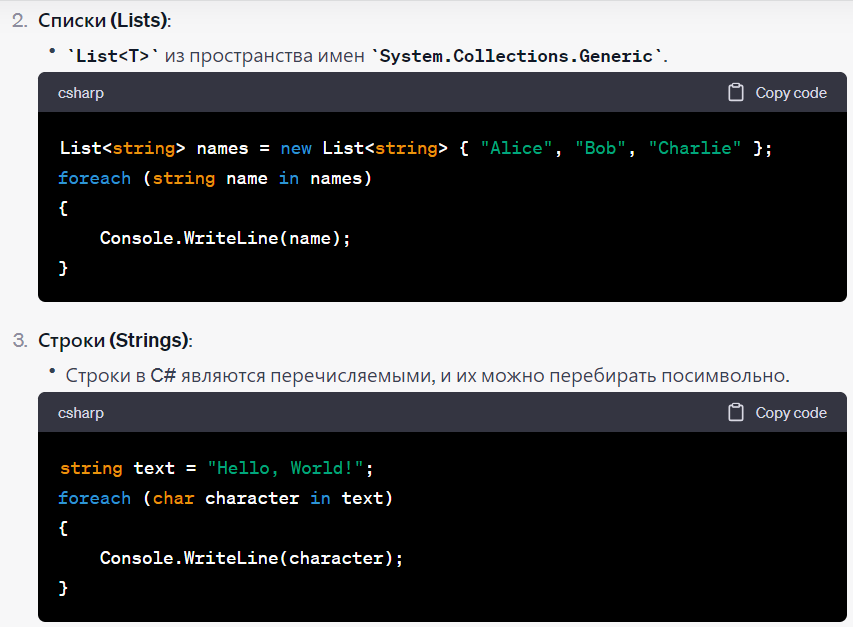
stairsArray[0] = new int[2];

stairsArray[1] = new int[3];

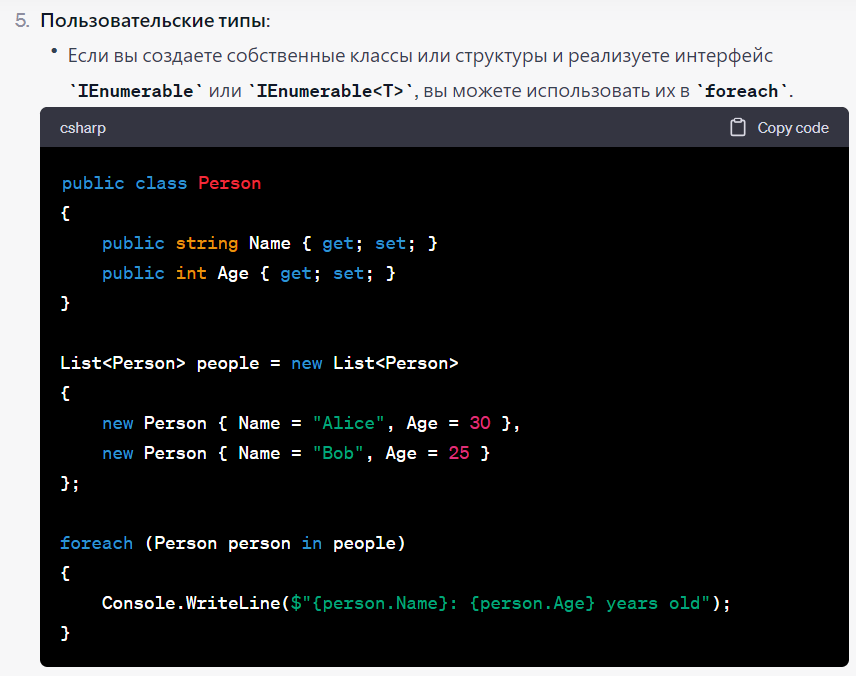
stairsArray[2] = new int[4];

**35.Какие типы можно использовать в foreach? Приведите пример.**









**36.Что такое кортеж? Для чего и как он используется?**

**Кортеж** — это структура данных, которая имеет определенное число и последовательность элементов. Пример кортежа — это структура данных с тремя элементами (известной как три кортежа или тройная), которая используется для хранения идентификатора, такого как имя человека в первом элементе, год во втором элементе и доход человека за этот год в третьем элементе.

Одной из задач, которую позволяет элегантно решить кортеж - это обмен значениями.

**37.Что такое локальная функция? Какова область ее видимости?**

**Локальные** функции представляют собой частные методы типа, вложенные в другой элемент. Они могут вызываться только из того элемента, в который вложены.

static (int, int, int, char) LocFun(int[] intArr, string str)

Данная лок ф-ия возвращает кортеж из 4 эл-ов. Принимает массив чисел и строку. Ее область видимости не выходит за границы самой ф-ии.

**38. В чем разница между кодом, заключенным в блок checked и кодом,  
заключенным в блок unchecked?**

При выходе за пределы типа данных (пр, int), то при checked появится ошибка при сборке, а при unchecked - при компиляции.

**39.Какой контекст (checked/unchecked) применяется по умолчанию?  
Как можно переопределить это поведение?**

Если контекст проверки переполнения не указан, значение параметра компилятора CheckForOverflowUnderflow определяет контекст по умолчанию для неконстантных выражений. По умолчанию значение этого параметра не задано, а арифметические операции целочисленного типа и преобразования выполняются в неспроверяемом контексте.

Переопр с пом опер-в checked И unchecked

**40.Для чего используется ключевое слово fixed? Каковы особенности  
его использования?**

Ключевое fixed слово позволяет закрепить локальный элемент в стеке, чтобы предотвратить его сбор или перемещение во время сборки мусора. Он используется для низкоуровневых сценариев программирования.