1. Когда говорят C#, нередко имеют в виду технологии платформы .NET (Windows Forms, WPF, ASP.NET, .NET MAUI). И, наоборот, когда говорят **.NET,** нередко имеют в виду C#. Однако, хотя эти понятия связаны, отождествлять их неверно. Язык C# был создан специально для работы с фреймворком .NET, однако само понятие .NET несколько шире.

Можно выделить следующие ее основные черты:

* Поддержка нескольких языков. Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), благодаря чему .NET поддерживает несколько языков: наряду с C# это также VB.NET, C++, F#, а также различные диалекты других языков, привязанные к .NET, например, Delphi.NET. При компиляции код на любом из этих языков компилируется в сборку на общем языке CIL (Common Intermediate Language) - своего рода ассемблер платформы .NET. Поэтому при определенных условиях мы можем сделать отдельные модули одного приложения на отдельных языках.
* Кроссплатформенность. .NET является переносимой платформой (с некоторыми ограничениями). Например, последняя версия платформы на данный момент - .NET 8 поддерживается на большинстве современных ОС Windows, MacOS, Linux. Используя различные технологии на платформе .NET, можно разрабатывать приложения на языке C# для самых разных платформ - Windows, MacOS, Linux, Android, iOS, Tizen.
* Мощная библиотека классов. .NET представляет единую для всех поддерживаемых языков библиотеку классов. И какое бы приложение мы не собирались писать на C# - текстовый редактор, чат или сложный веб-сайт - так или иначе мы задействуем библиотеку классов .NET.
* Разнообразие технологий. Общеязыковая среда исполнения CLR и базовая библиотека классов являются основой для целого стека технологий, которые разработчики могут задействовать при построении тех или иных приложений. Например, для работы с базами данных в этом стеке технологий предназначена технология ADO.NET и Entity Framework Core. Для построения графических приложений с богатым насыщенным интерфейсом - технология WPF и WinUI, для создания более простых графических приложений - Windows Forms. Для разработки кроссплатформенных мобильных и десктопных приложений - Xamarin/MAUI. Для создания веб-сайтов и веб-приложений - ASP.NET и т.д.

К этому стоит добавить активной развивающийся и набирающий популяность Blazor - фреймворк, который работает поверх .NET и который позволяет создавать веб-приложения как на стороне сервера, так и на стороне клиента. А в будущем будет поддерживать создание мобильных приложений и, возможно, десктоп-приложений.

Производительность. Согласно ряду тестов веб-приложения на .NET в ряде категорий сильно опережают веб-приложения, построенные с помощью других технологий. Приложения на .NET в принципе отличаются высокой производительностью.

Также еще следует отметить такую особенность языка C# и фреймворка .NET, как автоматическая сборка мусора. А это значит, что нам в большинстве случаев не придется, в отличие от С++, заботиться об освобождении памяти. Вышеупомянутая общеязыковая среда CLR сама вызовет сборщик мусора и очистит память.

**.NET Framework и .NET 8**

Стоит отметить, что .NET долгое время развивался премущественно как платформа для Windows под названием .NET Framework. В 2019 вышла последняя версия этой платформы - .NET Framework 4.8. Она больше не развивается

С 2014 Microsoft стал развивать альтернативную платформу - .**NET Core**, которая уже предназначалась для разных платформ и должна была вобрать в себя все возможности устаревшего .NET Framework и добавить новую функциональность. Затем Microsoft последовательно выпустил ряд версий этой платформы. И текущей версией является расматриваемая в этом руководстве платформа .NET 8. Поэтому следует различать .NET Framework, который предназначен преимущественно для Windows, и кросплатформенный .NET 8. В данном руководстве речь будет идти о C# 12 в связке с .NET 8, поскольку это актуальная платформа.

**ASP.NET Core** является кроссплатформенной, высокопроизводительной средой с открытым исходным кодом для создания современных облачных приложений, подключенных к Интернету.

ASP.NET Core позволяет выполнять следующие задачи:

* Создавать веб-приложения и службы, приложения Интернета вещей (IoT) и серверные части для мобильных приложений.
* Использовать избранные средства разработки в Windows, macOS и Linux.
* Выполнять развертывания в облаке или локальной среде.
* Запустите в .NET.

Преимущества, обеспечиваемые ASP.NET Core

ASP.NET Core предоставляет следующие преимущества:

* Единое решение для создания пользовательского веб-интерфейса и веб-API.
* Разработано для тестируемости.
* Razor Pages упрощает написание кода для сценариев страниц и повышает его эффективность.
* Blazor позволяет использовать в браузере язык C# вместе с JavaScript. совместное использование серверной и клиентской логик приложений, написанных с помощью .NET;
* Возможность разработки и запуска в ОС Windows, macOS и Linux.
* Открытый исходный код и ориентация на сообщество.
* Интеграция современных клиентских платформ и рабочих процессов разработки.
* Поддержка размещения служб удаленного вызова процедур (RPC) с помощью gRPC.
* Облачная система конфигурации на основе среды.
* Встроенное введение зависимостей.
* Упрощенный высокопроизводительный модульный конвейер HTTP-запросов.

Возможность размещения на следующих узлах:

Kestrel

IIS

HTTP.sys

Nginx

Docker

Управление параллельными версиями.

Инструментарий, упрощающий процесс современной веб-разработки.

**Common Intermediate Language (CIL)** — это байт-код, в какой

компилируется код на любом языке .NET. CIL не зависит от архитектуры процессора или операционной системы.

**Common Language Runtime (CLR)** — это виртуальная машина, исполняющая CIL-код. CLR обеспечивает такие функции, как сборка мусора, обработка исключений, безопасность и отладка.

**Common Type System (CTS) —** это набор правил для определения типов данных и их взаимодействия в CIL-коде. CTS гарантирует совместимость типов между разными языками.

**Common Language Specification (CLS)** — это подмножество CTS, которое определяет минимальный набор функций, которые должен поддерживать любой язык .NET. CLS обеспечивает взаимодействие кода на разных языках.

**Base Class Library (BCL)** — это набор стандартных классов и интерфейсов, которые предоставляют базовые функции для работы с файлами, строками, коллекциями, сетью.

Рассмотрим поэтапно:

* Когда программист пишет код на любом языке .NET, он использует BCL и другие библиотеки .NET.
* Затем он компилирует код в CIL-код с помощью специального компилятора для своего языка. Например, для C# используется компилятор csc.exe.
* Полученный CIL-код сохраняется в файле с расширением .exe или .dll.
* Потом пользователь запускает приложение на .NET, CLR загружает CIL-код в виртуальную машину CLR.
* CLR транслирует CIL-код в нативный код для конкретной платформы с помощью технологии JIT (Just-In-Time) компиляции.
* Результат: CIL-код становится исполняемым на любой платформе, где есть CLR.

3. **Стек** — это область оперативной памяти, которая создаётся для каждого потока. Он работает в порядке LIFO (Last In, First Out), то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека. Каждый раз, когда функция объявляет новую переменную, она добавляется в стек, а когда эта переменная пропадает из области видимости (например, когда функция заканчивается), она автоматически удаляется из стека. Когда стековая переменная освобождается, эта область памяти становится доступной для других стековых переменных.

Из-за такой природы стека управление памятью оказывается весьма логичным и простым для выполнения на ЦП; это приводит к высокой скорости, в особенности потому, что время цикла обновления байта стека очень мало, т.е. этот байт скорее всего привязан к кэшу процессора. Тем не менее, у такой строгой формы управления есть и недостатки. Размер стека — это фиксированная величина, и превышение лимита выделенной на стеке памяти приведёт к переполнению стека. Размер задаётся при создании потока, и у каждой переменной есть максимальный размер, зависящий от типа данных. Это позволяет ограничивать размер некоторых переменных (например, целочисленных), и вынуждает заранее объявлять размер более сложных типов данных (например, массивов), поскольку стек не позволит им изменить его. Кроме того, переменные, расположенные на стеке, всегда являются локальными.

В итоге стек позволяет управлять памятью наиболее эффективным образом — но если вам нужно использовать динамические структуры данных или глобальные переменные, то стоит обратить внимание на кучу.

**Куча**

Куча — это хранилище памяти, также расположенное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении. Именно так определяются глобальные переменные. По завершении приложения все выделенные участки памяти освобождаются. Размер кучи задаётся при запуске приложения, но, в отличие от стека, он ограничен лишь физически, и это позволяет создавать динамические переменные.

Вы взаимодействуете с кучей посредством ссылок, обычно называемых указателями — это переменные, чьи значения являются адресами других переменных. Создавая указатель, вы указываете на местоположение памяти в куче, что задаёт начальное значение переменной и говорит программе, где получить доступ к этому значению. Из-за динамической природы кучи ЦП не принимает участия в контроле над ней; в языках без сборщика мусора (C, C++) разработчику нужно вручную освобождать участки памяти, которые больше не нужны. Если этого не делать, могут возникнуть утечки и фрагментация памяти, что существенно замедлит работу кучи.

В сравнении со стеком, куча работает медленнее, поскольку переменные разбросаны по памяти, а не сидят на верхушке стека. Некорректное управление памятью в куче приводит к замедлению её работы; тем не менее, это не уменьшает её важности — если вам нужно работать с динамическими или глобальными переменными, пользуйтесь кучей.

**4. Сборщик мусора**

В отличие от массивов, строк и перечислений структуры C# не имеют идентич-

но именованного представления в библиотеке .NET Core (т.е. класс вроде System.

Structure отсутствует), но они являются неявно производными от абстрактного

класса System.ValueType. Роль класса System.ValueType заключается в обеспе-

чении размещения экземпляра производного типа (например, любой структуры ) в

стеке, а не в куче с автоматической сборкой мусора. Выражаясь просто, данные,

размещаемые в стеке, могут создаваться и уничтожаться быстро, т.к. время их жиз-

ни определяется областью видимости, в которой они объявлены . С другой стороны ,

данные, размещаемые в куче, отслеживаются сборщиком мусора .NET Core и имеют

время жизни, которое определяется многими факторами, объясняемыми в главе 9.

Так, при использовании переменных типов значений в методе, все значения этих переменных попадают в стек. После завершения работы метода стек очищается.

При использовании же ссылочных типов, например, объектов классов, для них также будет отводиться место в стеке, только там будет храниться не значение, а адрес на участок памяти в хипе или куче, в котором и будут находиться сами значения данного объекта. И если объект класса перестает использоваться, то при очистке стека ссылка на участок памяти также очищается, однако это не приводит к немедленной очистке самого участка памяти в куче. Впоследствии сборщик мусора (garbage collector) увидит, что на данный участок памяти больше нет ссылок, и очистит его.

Сборщик мусора не запускается сразу после удаления из стека ссылки на объект, размещенный в куче. Он запускается в то время, когда среда CLR обнаружит в этом потребность, например, когда программе требуется дополнительная память.

Как правило, объекты в куче располагаются неупорядочено, между ними могут иметься пустоты. Куча довольно сильно фрагментирована. Поэтому после очистки памяти в результате очередной сборки мусора оставшиеся объекты перемещаются в один непрерывный блок памяти. Вместе с этим происходит обновление ссылок, чтобы они правильно указывали на новые адреса объектов.

Так же надо отметить, что для крупных объектов существует своя куча - Large Object Heap. В эту кучу помещаются объекты, размер которых больше 85 000 байт. Особенность этой кучи состоит в том, что при сборке мусора сжатие памяти не проводится по причине больших издержек, связанных с размером объектов.

Несмотря на то что, на сжатие занятого пространства требуется время, да и приложение не сможет продолжать работу, пока не отработает сборщик мусора, однако благодаря подобному подходу также происходит оптимизация приложения. Теперь чтобы найти свободное место в куче среде CLR не надо искать островки пустого пространства среди занятых блоков. Ей достаточно обратиться к указателю кучи, который указывает на свободный участок памяти, что уменьшает количество обращений к памяти.

Кроме того, чтобы снизить издержки от работы сборщика мусора, все объекты в куче разделяются по поколениям. Всего существует три поколения объектов: 0, 1 и 2-е.

К поколению 0 относятся новые объекты, которые еще ни разу не подвергались сборке мусора. К поколению 1 относятся объекты, которые пережили одну сборку, а к поколению 2 - объекты, прошедшие более одной сборки мусора.

Когда сборщик мусора приступает к работе, он сначала анализирует объекты из поколения 0. Те объекты, которые остаются актуальными после очистки, повышаются до поколения 1.

Если после обработки объектов поколения 0 все еще необходима дополнительная память, то сборщик мусора приступает к объектам из поколения 1. Те объекты, на которые уже нет ссылок, уничтожаются, а те, которые по-прежнему актуальны, повышаются до поколения 2.

Поскольку объекты из поколения 0 являются более молодыми и нередко находятся в адресном пространстве памяти рядом друг с другом, то их удаление проходит с наименьшими издержками. Функционал сборщика мусора в библиотеке классов .NET представляет класс System.GC.

**5. Обьявление переменных. Типы var, def, dynamic.**

Перед использованием любую переменную надо определить. Синтаксис определения переменной выглядит следующим образом:

Вначале идет тип переменной, потом ее имя. В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:

* имя может содержать любые цифры, буквы и символ подчеркивания, при этом первый символ в имени должен быть буквой или символом подчеркивания
* в имени не должно быть знаков пунктуации и пробелов
* имя не может быть ключевым словом языка C#. Таких слов не так много, и при работе в Visual Studio среда разработки подсвечивает ключевые слова синим цветом.

Хотя имя переменой может быть любым, но следует давать переменным описательные имена, которые будут говорить об их предназначении.

Ключевое слово var в языке программирования C# предназначено для объявления переменных без явного указания их типа данных. Вместо этого компилятор C# автоматически определяет тип переменной на основе значения, которое ей присваивается при инициализации. Ключевое слово var нельзя применить к полям. Ключевое слово var никогда не может использоваться с возвращаемым значением, параметром или членом класса/структуры .

Dynamic - динамический тип данных. Это такой тип данных, который не имеет привязанного типа и в любой момент может быть переопределен другим типом. Тип dynamic не типизирован статически. То есть такой переменной можно присвоить любое значение в любой момент времени.

При передаче параметров по ссылке перед параметрами используется модификатор ref

При передаче значений параметрам по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место..

Так, в метод Increment передается ссылка на саму переменную number в памяти. И если значение параметра n в Increment изменяется, то это приводит и к изменению переменной number, так как и параметр n и переменная number указывают на один и тот же адрес в памяти.

Обратите внимание, что модификатор ref указывается как перед параметром при объявлении метода, так и при вызове метода перед аргументом, который передается параметру.

**6. Иммутабельные переменные(Константы)**

Константа должна быть обязательно инициализирована при определении, и после определения значение константы не может быть изменено

Константы предназначены для описания таких значений, которые не должны изменяться в программе. Для определения констант используется ключевое слово const, которое указывается перед типом константы.

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

**readonly**

**7. Типы данных в C#**

Типы значений:

* 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)

Ссылочные типы:

* Тип object
* Тип string
* Классы (class)
* Интерфейсы (interface)
* Делегаты (delegate)

Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, условно удаляет этот объект и очищает память - фактически помечает, что данный сегмент памяти может быть использован для хранения других данных.

**8. Примитивные типы данных**

В языке C# есть следующие базовые типы данных:

* bool
* byte
* sbyte
* short
* ushort
* int
* uint
* long
* ulong
* float
* double
* decimal
* char
* string
* object

**9 .Ссылочные типы данных**

Ссылочными типами данных называют типы, которые не хранятся непосредственно в стеке(Там хранятся их адреса в хипе). Фрагментированные ссылочные переменных хранятся в куче, до момента работы сборщика мусора. Преимущество ссылочных типов заключает в месте их хранения: они не ограничены размерами примитивных переменных, а так же могут быть доступны в том или ином виде из любой точки программы.

В С# существуют следующие ссылочные типы данных:

* Обьект
* Класс
* Интерфейс
* Делегат

**10. Размеры и значения типов данных**

В языке C# есть следующие базовые типы данных:

* bool: хранит значение true или false (логические литералы). Представлен системным типом System.Boolean
* byte: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte
* sbyte: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte
* short: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16
* ushort: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16
* int: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:
* uint: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32
* long: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом System.Int64
* ulong: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64
* float: хранит число с плавающей точкой от -3.4\*1038 до 3.4\*1038 и занимает 4 байта. Представлен системным типом System.Single
* double: хранит число с плавающей точкой от ±5.0\*10-324 до ±1.7\*10308 и занимает 8 байта. Представлен системным типом System.Double
* decimal: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от ±1.0\*10-28 до ±7.9228\*1028, может хранить 28 знаков после запятой и занимает 16 байт. Представлен системным типом System.Decimal
* char: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом System.Char. Этому типу соответствуют символьные литералы:
* string: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.
* object: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом System.Object, который является базовым для всех других типов и классов .NET.

Если превысить предел типа переменной, то переменная будет равна 0.

**11. Checked и Unchecked**

Контексты checked и unchecked влияют на реакцию компилятора на арифметическое переполнение. Если задан контекст checked арифметическое переполнение, например, в цикле, сгенерирует исключение, а если контекст unchecked то переполнение будет проигнорировано. Это помогает отследить и не допустить переполнение какой либо переменной.

Для того чтобы проверять арифметическое переполнение существует флаг /checked или можно задать в vscode проверку на арифметическое переполнение.

В таком случае, если нужно НЕ проверять и выдавать исключение Overflow exception, существует контекст Unchecked.

**12. Приведение типов. Boxing и unboxing.**

Операция преобразования типов предполагает указание в скобках того типа, к которому надо преобразовать значение.Преобразования могут быть сужающие (narrowing) и расширяющие (widening). Расширяющие преобразования расширяют размер объекта в памяти.Сужающие преобразования, наоборот, сужают значение до типа меньшей разядности.

**Неявные преобразования**

В случае с расширяющими преобразованиями компилятор за нас выполнял все преобразования данных, то есть преобразования были неявными (implicit conversion). Такие преобразования не вызывают каких-то затруднений. Тем не менее стоит сказать пару слов об общей механике подобных преобразований.

Если производится преобразование от безнакового типа меньшей разрядности к безнаковому типу большой разрядности, то добавляются дополнительные биты, которые имеют значени 0. Это называется дополнение нулями или zero extension.

Если производится преобразование к знаковому типу, то битовое представление дополняется нулями, если число положительное, и единицами, если число отрицательное. Последний разряд числа содержит знаковый бит - 0 для положительных и 1 для отрицательных чисел. При расширении в добавленные разряды компируется знаковый бит.

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:

**Явные преобразования**

Расширяющие преобразования от типа с меньшей разрядностью к типу с большей разрядностью компилятор проводит неявно. Это могут быть следующие цепочки преобразований:

* byte -> short -> int -> long -> decimal
* int -> double
* short -> float -> double
* char -> int

Упаковка (boxing) и распаковка (unboxing) — операции преобразования значимых типов данных в ссылочные и обратно. Эти операции могут негативно сказываться на производительности из-за дополнительных вычислений: выделения памяти под новый объект и копирования данных.

Упаковка из за сущности ссылочных типов очень полезна для получения доступа к нужному локальному значению.

**13. if...else**

Конструкция if/else проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код.

Ее простейшая форма состоит из блока if:

После ключевого слова if ставится условие. Условие должно представлять значение типа bool. Это может быть непосредственно значение типа bool или результат условного выражения или другого выражения, которое возвращает значение типа bool. И если это условие истинно (равно true), то срабатывает код, который помещен далее после условия внутри фигурных скобок.

В данном случае у нас первое число больше второго, поэтому выражение num1 > num2 истинно и возвращает true, следовательно, управление переходит к строке Console.WriteLine("Число {num1} больше числа {num2}");

Если блок if содержит одну инструкцию, то мы можем его сократить, убрав фигурные скобки:

Также мы можем соединить сразу несколько условий, используя логические операторы:

Но что, если мы захотим, чтобы при несоблюдении условия также выполнялись какие-либо действия? В этом случае мы можем добавить блок else

**14. Тернарный оператор**

Тернарную операция также позволяет проверить некоторое условие и в зависимости от его истинности выполнить некоторые действия. Она имеет следующий синтаксис:

[первый операнд - условие] ? [второй операнд] : [третий операнд]

Здесь сразу три операнда. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий. Тернарный оператор выгодно использовать, когда инструкции которые нужно выполнить в зависимости от true или false не превышают по обьему 1 строку.

**15. switch..case и matching patterms**

Конструкция switch/case оценивает некоторое выражение и сравнивает его значение с набором значений. И при совпадении значений выполняет определенный код.:

После ключевого слова switch в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора сase. И если совпадение будет найдено, то будет выполняться определенный блок сase.

В конце каждого блока сase должен ставиться один из операторов перехода: break, goto case, return или throw. Как правило, используется оператор break. При его применении другие блоки case выполняться не будут.

Мы можем добавить в конструкцию switch блок default, в котором будет возвращаться значение при всех остальных случаях.

Также, кроме обычных переменных мы можем использовать в switch конструкции matching patterns для сравнения с образцом

**16. Оператор goto**

Оператор goto используется для безусловного перехода по метке, которая указана в коде. Это устаревший оператор, использование которого не рекомендовано в настоящее время, так как высок риск бесконечного цикла. Оператор goto можно использовать в конструкции switch для перехода к определенному case по условию.

**17. Циклы**

Циклы являются управляющими конструкциями, позволяя в зависимости от определенных условий выполнять некоторое действие множество раз. В C# имеются следующие виды циклов:

* for
* foreach
* while
* do...while

Объявление цикла for состоит из трех частей. Первая часть объявления цикла - некоторые действия, которые выполняются один раз до выполнения цикла. Обычно здесь определяются переменные, которые будут использоваться в цикле.

Вторая часть - условие, при котором будет выполняться цикл. Пока условие равно true, будет выполняться цикл.

И третья часть - некоторые действия, которые выполняются после завершения блока цикла. Эти действия выполняются каждый раз при завершении блока цикла.

После объявления цикла в фигурных скобках помещаются сами действия цикла.

В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while. И пока это условие истинно, цикл повторяется.

В отличие от цикла do цикл while сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется.

Цикл foreach предназначен для перебора набора или коллекции элементов.

После оператора foreach в скобках сначала идет определение переменной. Затем ключевое слово in и далее коллекция, элементы которой надо перебрать.

При выполнении цикл последовательно перебирает элементы коллекции и помещает их в переменную, и таким образом в блоке цикла мы можем выполнить с ними некоторые действия.

Иногда возникает ситуация, когда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором break.

Оператор continue используется если необходимо проигнорировать какую то итерацию. С помощью return мы можем вернуть какие либо значения из цикла.

**18. Классы и методы**

Класс представляет собой шаблон, по которому определяется форма объекта. В нем указываются данные и код, который будет оперировать этими данными. В C# используется спецификация класса для построения объектов, которые являются экземплярами класса. Следовательно, класс, по существу, представляет собой ряд схематических описаний способа построения объекта. При этом очень важно подчеркнуть, что класс является логической абстракцией. Физическое представление класса появится в оперативной памяти лишь после того, как будет создан объект этого класса.

При определении класса объявляются данные, которые он содержит, а также код, оперирующий этими данными. Если самые простые классы могут содержать только код или только данные, то большинство настоящих классов содержит и то и другое. Если переменные хранят некоторые значения, то методы содержат собой набор инструкций, которые выполняют определенные действия. По сути метод - это именованный блок кода, который выполняет некоторые действия. Метод при определение может содержать: модификаторы доступа, входные параметры, специальные параметры, и значение которое он должен отдать. Экземпляр класса вызывается следующим образом: Class var = new Class()

**19. Конструктор класса**

На уровне кода конструктор представляет метод, который называется по имени класса, который может иметь параметры, но для него не надо определять возвращаемый тип. Как правило, конструктор выполняет инициализацию объекта. Если конструктор задан вручную, то класс лишается неявно заданного конструктора. Для конструктора нет необходимости определять возвращаемый тип. Конструктор можно использовать для определения значения переменных по умолчанию. Приоритет исполнения кода конструктора всегда выше чем остальных участков. В с# конструкторов может быть несколько. Он всегда вызывается при создании экземпляра класса. С С# 12 версии существует первичный конструктор. Он позволяет добавлять параметры к определению класса и использовать эти параметры внутри класса. Кроме первичного конструктора можно добавлять и другие, но при этом другие должны вызывать первичный конструктор. Несколько конструкторов нужно для передачи в конструкторов разного количества параметров.

**public Person(string name) : this(name, 18) { }**

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

**Person tom = new Person { name = "Tom", age = 31 }**

Также для работы с полями можно использовать деконструктор.

Деконструкторы (не путать с деструкторами) позволяют выполнить декомпозицию объекта на отдельные части.

**public void Deconstruct(out string personName, out int personAge)**

**{**

**personName = name;**

**personAge = age;**

**}**

**(string name, int age) = person;**

https://metanit.com/sharp/tutorial/3.35.php

**20. Свойства get set**

Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют свойства. Они обеспечивают простой доступ к полям классов и структур, узнать их значение или выполнить их установку.

Вначале определения свойства могут идти различные модификаторы, в частности, модификаторы доступа. Затем указывается тип свойства, после которого идет название свойства. Полное определение свойства содержит два блока: get и set.

В блоке get выполняются действия по получению значения свойства. В этом блоке с помощью оператора return возвращаем некоторое значение.

В блоке set устанавливается значение свойства. В этом блоке с помощью параметра value мы можем получить значение, которое передано свойству.

Блоки get и set еще называются акссесорами или методами доступа (к значению свойства), а также геттером и сеттером. В C# геттеры и сеттеры не идут отдельными методами а находятся в методе. Блоки set и get не обязательно одновременно должны присутствовать в свойстве. Если свойство определяет только блок get, то такое свойство доступно только для чтения - мы можем получить его значение, но не установить.

И, наоборот, если свойство имеет только блок set, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить.

Свойства необязательно связаны с определенной переменной. Они могут вычисляться на основе различных выражений.

Свойства управляют доступом к полям класса. Однако что, если у нас с десяток и более полей, то определять каждое поле и писать для него однотипное свойство было бы утомительно. Поэтому в .NET были добавлены автоматические свойства. Они имеют сокращенное объявление:

**public string Name { get; set; } = "Tom";**

**public int Age { get; set; } = 37;**

Начиная с версии C# 9.0 сеттеры в свойствах могут определяться с помощью оператора init (от слова "инициализация" - это есть блок init призван инициализировать свойство). Для установки значений свойств с init можно использовать только инициализатор, либо конструктор, либо при объявлении указать для него значение. После инициализации значений подобных свойств их значения изменить нельзя - они доступны только для чтения. В этом плане init-свойства сближаются со свойствами для чтения. Разница состоит в том, что init-свойства мы также можем установить в инициализаторе (свойства для чтения установить в инициализаторе нельзя).

https://metanit.com/sharp/tutorial/3.4.php

**21. Модификаторы доступа.**

Все поля, методы и остальные компоненты класса имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для компонентов класса. То есть модификаторы доступа определяют контекст, в котором можно употреблять данную переменную или метод.

Когда мы создаем приложение в результате компиляции в Visual Studio или в консоли, результатом этой работы является файл exe или dll (в зависимости от выбранных настроек), который называется сборкой приложения. Сборка является базовой структурной единицей в .NET, на уровне которой проходит контроль версий, развертывание и конфигурация приложения.

Сборки кристаллизуют всю библиотеку классов .NET - при написании кода и создании сборки своего приложения мы используем пространства имен, которые размещены в других сборках .NET.

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

Стоит отметить, что эти модификаторы могут применяться как к компонентам класса, так и к компонентам структуры за тем исключением, что структуры не могут использовать модификаторы private protected, protected и protected internal, поскольку структуры не могут быть унаследованы.

Все классы и структуры, определенные напрямую вне других типов (классов и структур) могут иметь только модификаторы public, file или internal.

Мы можем явно задать модификатор доступа, а можем его и не указывать:

Если для компонентов не определен модификатор доступа, то по умолчанию для них применяется модификатор private. Например, в примере выше переменная name неявно будет иметь модификатор private.

**22. Статические поля, свойства, классы**

Кроме обычных полей, методов, свойств классы и структуры могут иметь статические поля, методы, свойства. Статические поля, методы, свойства относятся ко всему классу/всей структуре и для обращения к подобным членам необязательно создавать экземпляр класса / структуры. Статические поля хранят состояние всего класса / структуры. Статическое поле определяется как и обычное, только перед типом поля указывается ключевое слово static.

Обращаться к статическим полям нужно также как и к обычным — Class.field.

Подобным образом мы можем создавать и использовать статические свойства.

Таким образом, переменные и свойства, которые хранят состояние, общее для всех объектов класса / структуры, следует определять как статические.

Статические методы определяют общее для всех объектов поведение, которое не зависит от конкретного объекта. Для обращения к статическим методам также применяется имя класса / структуры. Следует учитывать, что статические методы могут обращаться только к статическим членам класса. Обращаться к нестатическим методам, полям, свойствам внутри статического метода мы не можем.

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

* Статические конструкторы не должны иметь модификатор доступа и не принимают параметров
* Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса
* Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются)

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

Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы. Например, определим класс, который выполняет ряд арифметических операций

https://metanit.com/sharp/tutorial/3.4.php

**23.Наследование**

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

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

* Не поддерживается множественное наследование, класс может наследоваться только от одного класса.
* При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа internal, то производный класс может иметь тип доступа internal или private, но не public. Однако следует также учитывать, что если базовый и производный класс находятся в разных сборках (проектах), то в этом случае производый класс может наследовать только от класса, который имеет модификатор public.
* Если класс объявлен с модификатором sealed, то от этого класса нельзя наследовать и создавать производные классы.

С помощью ключевого слова base мы можем обратиться к базовому классу.

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных.

https://metanit.com/sharp/tutorial/3.7.php

**24. Полиморфизм**

Полиморфизм— одна из трех основных парадигм ООП. Если говорить кратко, полиморфизм— это способность обьекта использовать методы производного класса, который не существует на момент создания базового.

Основная выгода полиморфизма — легкость, с которой можно создавать новые классы, «ведущие себя» аналогично родственным, что, в свою очередь, позволяет достигнуть расширяемости и модифицируемости.

Преимущества принципа полиморфизма

* Гибкость и расширяемость. Полиморфизм позволяет добавлять новые типы объектов и операций без изменения существующего кода. Новые классы, реализующие общий интерфейс, могут быть легко интегрированы в существующую систему.
* Упрощение кода. Полиморфизм способствует уменьшению дублирования кода. Общий интерфейс или абстрактный базовый класс позволяют описать общее поведение, и каждый конкретный класс реализует только свою специфичную логику.
* Читаемость кода. Полиморфизм делает код более интуитивно понимаемым, так как работа с различными объектами происходит через общий интерфейс. Это упрощает восприятие кода другими разработчиками и способствует поддержке программы.
* Расширение функциональности. Добавление новых функций или операций для существующих классов становится проще. Достаточно реализовать необходимые методы в новых классах, которые наследуют общий интерфейс.
* Повторное использование кода. Полиморфизм позволяет использовать одни и те же методы для разных типов данных. Это устраняет необходимость создания аналогичных функций для разных классов.
* Улучшение тестирования. Тестирование становится более удобным, так как можно создать общие тестовые сценарии для всех классов, реализующих один интерфейс. Это способствует повышению качества и надежности программы.
* Абстракция и инкапсуляция. Полиморфизм позволяет абстрагироваться от конкретных реализаций и сосредоточиться на общем поведении объектов. Также он способствует инкапсуляции, разделяя интерфейс от деталей реализации.
* Облегчение командной разработки. Когда разработчики работают над разными частями программы, полиморфизм позволяет им взаимодействовать через общие интерфейсы без необходимости глубокого понимания внутренней реализации друг друга.

Виды полиморфизма в объектно-ориентированном программировании

* Полиморфизм подтипов (наследования)
* Параметрический полиморфизм (обобщённое программирование)
* Полиморфизм в интерфейсах

**25. Интерфейсы**

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.

Определение интерфейса

Для определения интерфейса используется ключевое слово interface. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования.

Что может определять интерфейс? В целом интерфейсы могут определять следующие сущности:

* Методы
* Свойства
* Индексаторы
* События
* Статические поля и константы (начиная с версии C# 8.0)

Однако интерфейсы не могут определять нестатические переменные. Например, простейший интерфейс, который определяет все эти компоненты:

Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами и свойствами абстрактных классов.

Еще один момент в объявлении интерфейса: если его члены - методы и свойства не имеют модификаторов доступа, то фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Это касается также и констант и статических переменных, которые в классах и структурах по умолчанию имееют модификатор private. В интерфейсах же они имеют по умолчанию модификатор public. И например, мы могли бы обратиться к константе minSpeed и переменной maxSpeed интерфейса IMovable:

Но также, начиная с версии C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса:

Как и классы, интерфейсы по умолчанию имеют уровень доступа internal, то есть такой интерфейс доступен только в рамках текущего проекта. Но с помощью модификатора public мы можем сделать интерфейс общедоступным.

Также начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Это значит, что мы можем определить в интерфейсах полноценные методы и свойства, которые имеют реализацию как в обычных классах и структурах. Например, определим реализацию метода Move по умолчанию:

С реализацией свойств по умолчанию в интерфейсах дело обстоит несколько сложнее, поскольку мы не можем определять в интерфейсах нестатические переменные, соответственно в свойствах интерфейса мы не можем манипулировать состоянием объекта. Тем не менее реализацию по умолчанию для свойств мы тоже можем определять:

Стоит отметить, что если интерфейс имеет приватные методы и свойства (то есть с модификатором private), то они должны иметь реализацию по умолчанию. То же самое относится к статическим методам (не обязательно приватным)

**26. Абстрактный класс**

Кроме обычных классов в C# есть абстрактные классы. Зачем они нужны? Классы обычно представляют некий план определенного рода объектов или сущностей. Например, мы можем определить класс Car для преставления машин или класс Person для представления людей, вложив в эти классы соответствующие свойства, поля, методы, которые будут описывать данные объекты. Однако некоторые сущности, которые мы хотим выразить с помощью языка программирования, могут не иметь конкретного воплощения. Например, в реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами. И для описания подобных сущностей, которые не имеют конкретного воплощения, предназначены абстрактные классы.

Абстрактный класс похож на обычный класс. Он также может иметь переменные, методы, конструкторы, свойства. Единственное, что при определении абстрактных классов используется ключевое слово abstract.

Кроме обычных свойств и методов абстрактный класс может иметь абстрактные члены классов, которые определяются с помощью ключевого слова abstract и не имеют никакого функционала. В частности, абстрактными могут быть:

* Методы
* Свойства
* Индексаторы
* События

Абстрактные члены классов не должны иметь модификатор private. При этом производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе. При переопределении в производном классе такой метод или свойство также объявляются с модификатором override (как и при обычном переопределении виртуальных методов и свойств). Также следует учесть, что если класс имеет хотя бы один абстрактный метод (или абстрактные свойство, индексатор, событие), то этот класс должен быть определен как абстрактный.

Абстрактные члены также, как и виртуальные, являются частью полиморфного интерфейса. Но если в случае с виртуальными методами мы говорим, что класс-наследник наследует реализацию, то в случае с абстрактными методами наследуется интерфейс, представленный этими абстрактными методами.

https://metanit.com/sharp/tutorial/3.8.php

**27. Принципы SOLID**

SOLID - это принципы разработки программного обеспечения, следуя которым Вы получите хороший код, который в дальнейшем будет хорошо масштабироваться и поддерживаться в рабочем состоянии.

S - Single Responsibility Principle - принцип единственной ответственности. Каждый класс должен иметь только одну зону ответственности.

O - Open closed Principle - принцип открытости-закрытости. Классы должны быть открыты для расширения, но закрыты для изменения.

L - Liskov substitution Principle - принцип подстановки Барбары Лисков. Должна быть возможность вместо базового (родительского) типа (класса) подставить любой его подтип (класс-наследник), при этом работа программы не должна измениться.

I - Interface Segregation Principle - принцип разделения интерфейсов. Данный принцип обозначает, что не нужно заставлять клиента (класс) реализовывать интерфейс, который не имеет к нему отношения.

D - Dependency Inversion Principle - принцип инверсии зависимостей. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракции. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций

**28. Инкапсуляция**

Инкапсуляция — это возможность (механизм) с помощью которой мы можем спрятать от конечного пользователя реализацию того или иного метода, устройства объекта и так далее и представить в пользование только то, что необходимо для работы, обеспечив тем самым целостность объекта.

Инкапсуляция в ООП позволяет скрыть от конечного пользователя реализацию той или иной части объекта (класса), обеспечивая тем самым целостность работы. Инкапсуляция в C# реализуется с использованием различных модификаторов доступа к полям и свойствам объектов. Без инкапсуляции нам бы пришлось перед использованием того или иного метода или свойства вникать в реализацию, чтобы не натыкаться на различного рода ограничения и ошибки, с инкапсуляцией же мы в большей степени уверены в том, что внутренние механизмы объекта максимально «разрулят» все возможные проблемы и, в случае необходимости, сообщат нам об исключительной ситуации.

**29. Перечисление Enum**

Кроме примитивных типов данных в языке программирования C# есть такой тип как enum или перечисление. Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum. После оператора enum идет название перечисления. И затем в фигурных скобках через запятую перечисляются константы перечисления. Перечисления выгодно использовать в if, switch. Синтаксис:

enum Test

{

item1;

item2;

item3;

}

Test.item3;

**30 Подключение модулей в C#**

Вместо создания своих собственных библиотек и определения своего собственного функционала мы можем использовать уже ранее созданные другими разработчиками библиотеки и добавить их свой проект. Для управления внешними библиотеками в виде отдельных пакетов Microsoft предоставляет специальный менеджер пакетов под названием Nuget.

Физически NuGet-пакет представляет единый архивный файл с расширением .nupkg, который содержит скомпилированный код в виде библиотек dll и других файлов, используемых в коде. Также пакет включает некоторое описание в виде номера версии и вспомогательной информации.

Для упрощения работы с пакетами, их распространенния Microsoft организовал глобальный репозиторий пакетов в виде сервиса nuget.org. И если разработчик хочет поделиться своими наработками, он может загрузить свой код в виде Nuget-пакета в этот репозиторий, а другие разработчики могут скачать этот пакет и использовать его в свой программе. Посмотрим, как устанавливать Nuget-пакеты и использовать их в своем проекте. Прежде всего стоит отметить, что есть различные способы установки пакетов. В данном случае рассмотрим установку пакетов с помощью .NET CLI и в Visual Studio. Подключаются пакеты при помощи директивы Using.

**31. Обобщения типов(generics)**

Угловые скобки в описании class Product<T> указывают, что класс является обобщенным, а тип T, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву T, это может быть и любая другая буква. Причем сейчас на этапе написания кода нам неизвестно, что это будет за тип, это может быть любой тип. Поэтому параметр T в угловых скобках еще называется универсальным параметром, так как вместо него можно подставить любой тип. Их можно использовать, когда непонятно какой тип нужно будет использовать в классе, методе, или поле.

https://metanit.com/sharp/tutorial/3.8.php

**32. Иммутабельные коллекции и почему их выгодно использовать.**

<https://bool.dev/blog/detail/kak-pravilno-vybrat-net-kollektsiyu>

Иммутабельные коллекции не входят в стандартную библиотеку классов (BCL). Для их использования нужно установить System.Collections.Immutable NuGet пакет.

Они позволяют безопасно работать в многопоточной среде. Вместо того чтобы использовать блокировки синхронизации, как это делают многопоточные коллекции, неизменяемые коллекции не могут быть изменены после создания. Это автоматически делает их безопасными для использования в многопоточных сценариях, так как нет возможности другому потоку изменить коллекцию.

Сами же коллекции можно поделить на несколько видов:

Mutable — обычные коллекции которые поддерживают изменения.

Immutable — коллекции, которые полностью запрещают изменения. Хотя на самом деле любое изменение иммутабельной коллекции приводит к созданию новой.

ReadOnly — обертки над стандартными коллекциями, которые не дают поменять данные. Из-за того что это всего лишь обертка мы можем поменять данные в оригинальной коллекции и ead only коллекция подтянет изменения.

**33. Коллекции в C#: List<T>, LinkedList<T>.**

Список List

Класс List представляет из себя простейший список однотипных элементов, которые можно получить по индексу. Предоставляет методы для поиска, сортировки и изменения списка.

var linkedList = new List<string>();

linkedList.Add("A");

linkedList.Add("B");

linkedList.Add("C");

Для своей работы списки используют обычные массивы. Это значит что могут быть проблемы с производительностью из-за частого создания нового массива. Также у списков есть еще два нюанса:

Вставка элемента в середину списка приведет к созданию нового массива и копирование данных, что негативно влияет на производительность.

Списки не могут хранить экстремально огромное количество элементов из-за фрагментации адресного пространства.

Класс LinkedList реализует простой двухсвязный список, каждый элемент которого имеет ссылка на предыдущий и следующий элемент.

Каждый элемент списка оборачивается в специальный класс LinkedListNode, который имеет ссылку на следующий элемент (Next), на предыдущий элемент (Previous) и само значение (Value).

var linkedList = new LinkedList<string>();

linkedList.AddFirst("A");

linkedList.AddLast("B");

linkedList.AddLast("C");

Console.WriteLine(linkedList.First.Previous == null); // True

Console.WriteLine(linkedList.Last.Next == null); // True

Связанный список позволяет вставлять и удалять элементы со сложностью O (1). Также мы можем удалить элемент и заново вставить в тот же или другой список без дополнительного выделения памяти.

**34. Коллекции в C#: *Queue<T>*, *Stack<T>*, *Dictionary<T, V>*.**

Стек Stack и Очередь Queue.Очереди и стеки полезны, когда нужно временно хранить какие-то элементы, то есть удалять элемент после его извлечения. Также они позволяют определить строгую очередность записи и извлечения элементов.

Стеки Stack — реализуют подход LIFO (last in — first out).

var stack = new Stack<int>();

stack.Push(1); // stack = [1]

stack.Push(2); // stack = [1,2]

var item = stack.Pop(); // stack = [1], item = 2

Очереди Queue — реализуют подход (first in — first out).

var queue = new Queue<int>();

queue.Enqueue(1); // queue = [1]

queue.Enqueue(2); // queue = [1,2]

item = queue.Dequeue(); // queue = [2], item = 1

Внутри они реализованы с помощью обычных массивов.

https://ci-sharp.ru/obuchenie/oshibki-i-ih-resheniya/vozvrat-neskolkikh-znachenii-vyzyvaiushchemu-metodu-v-c/

**35.Коллекции в C#: *ArrayList<T>*, *HashSet<T>*.**

Эти классы реализуют математические множества. По своей природе множество — набор уникальных элементов с общими характеристиками, в нашем случае одного типа.

Также множества отличаются от обычных списков тем что они предоставляют набор методов, которые реализуют операции с теории множеств.

Внутренняя реализация этих классов отличается:

HashSet — множество, построенное на базе хеш-таблицы.

**36.Что такое строковая интерполяция и как ее можно применять? Как сложить несколько строк вместе?**

Интерполяция строк призвана упростить форматирование строк. Знак доллара перед строкой указывает, что будет осуществляться интерполяция строк. Внутри строки опять же используются плейсхолдеры {...}, только внутри фигурных скобок уже можно напрямую писать те выражения, которые мы хотим вывести.

Интерполяция по сути представляет более лаконичное форматирование. При этом внутри фигурных скобок мы можем указывать не только свойства, но и различные выражения языка C#. Также внутри фигурных скобок можно выполнять более сложные выражения, например, вызывать методы. Уже внутри строки можно применять форматирование.

Сложение нескольких строк вместе называется конкатенацией. Операция конкатенации может производиться в C# при помощи метода Concat или +, или при помощи метода Join.

**37.Как в C# взять подстроку строки? Как обрезать лишние пробелы в начале и в конце строки? Как поменять регистр строки? Как разделить строку по символу?**

С помощью метода IndexOf мы можем определить индекс первого вхождения отдельного символа или подстроки в строке:

Подобным образом действует метод LastIndexOf, только находит индекс последнего вхождения символа или подстроки в строку.

Еще одна группа методов позволяет узнать начинается или заканчивается ли строка на определенную подстроку. Для этого предназначены методы StartsWith и EndsWith. Например, в массиве строк хранится список файлов, и нам надо вывести все файлы с расширением exe:

Для обрезки начальных или концевых символов используется функция Trim:

Функция Trim без параметров обрезает начальные и конечные пробелы и возвращает обрезанную строку. Чтобы явным образом указать, какие начальные и конечные символы следует обрезать, мы можем передать в функцию массив этих символов.

Эта функция имеет частичные аналоги: функция TrimStart обрезает начальные символы, а функция TrimEnd обрезает конечные символы.

Обрезать определенную часть строки позволяет функция Substring:

Функция Substring также возвращает обрезанную строку. В качестве параметра первая использованная версия применяет индекс, начиная с которого надо обрезать строку. Вторая версия применяет два параметра - индекс начала обрезки и длину вырезаемой части строки.

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

С помощью функции Split мы можем разделить строку на массив подстрок. В качестве параметра функция Split принимает массив символов или строк, которые и будут служить разделителями. Например, подсчитаем количество слов в сроке, разделив ее по пробельным символам:

Это не лучший способ разделения по пробелам, так как во входной строке у нас могло бы быть несколько подряд идущих пробелов и в итоговый массив также бы попадали пробелы, поэтому лучше использовать другую версию метода:

Второй параметр StringSplitOptions.RemoveEmptyEntries говорит, что надо удалить все пустые подстроки.

**38.Как в C# работать со значением null? Для чего оно нужно?**

Если мы собираемся использовать переменную или параметр, которые допускают значение null, то есть представляют nullable-тип (не важно значимый или ссылочный), то, чтобы избежать возникновения NullReferenceException, мы можем проверить на null:

В данном случае если параметр text не равен null, то вызываем у строки метод ToUpper(), который переводит символы строки в верхний регистр.

Кроме того, с помощью оператора is мы можем проверить значение объекта.

Если объект слева от оператора is имеет значение справа от оператора. тогда оператор is возвращает true, иначе возвращается false

Также можно проверить на соответствие типу, значение которого мы собираемся использовать:

Подобные проверки еще называются null guard или условно говоря "защита от null".

Значение null в C# используется для обозначения отсутствия значения у переменной. Это может быть полезно, когда вы не знаете начальное значение переменной или когда нужно указать на то, что результат еще не был получен.

https://metanit.com/sharp/tutorial/3.26.php

**39. Как обрабатывать ошибки? Как обработать определенную ошибку?**

Иногда при выполнении программы возникают ошибки, которые трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Например, при передачи файла по сети может неожиданно оборваться сетевое подключение. такие ситуации называются исключениями. Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция try...catch...finally.

При использовании блока try...catch..finally вначале выполняются все инструкции в блоке try. Если в этом блоке не возникло исключений, то после его выполнения начинает выполняться блок finally. И затем конструкция try..catch..finally завершает свою работу.

Если же в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается, и среда CLR начинает искать блок catch, который может обработать данное исключение. Если нужный блок catch найден, то он выполняется, и после его завершения выполняется блок finally.

Если нужный блок catch не найден, то при возникновении исключения программа аварийно завершает свое выполнение.

https://metanit.com/sharp/tutorial/2.14.php

Базовым для всех типов исключений является тип Exception. Этот тип определяет ряд свойств, с помощью которых можно получить информацию об исключении.

* 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 вызывов, которые привели к возникновению исключения
* TargetSite: возвращает метод, в котором и было вызвано исключение

Однако так как тип Exception является базовым типом для всех исключений, то выражение catch (Exception ex) будет обрабатывать все исключения, которые могут возникнуть.

Но также есть более специализированные типы исключений, которые предназначены для обработки каких-то определенных видов исключений. Их довольно много, я приведу лишь некоторые:

* DivideByZeroException: представляет исключение, которое генерируется при делении на ноль
* ArgumentOutOfRangeException: генерируется, если значение аргумента находится вне диапазона допустимых значений
* ArgumentException: генерируется, если в метод для параметра передается некорректное значение
* IndexOutOfRangeException: генерируется, если индекс элемента массива или коллекции находится вне диапазона допустимых значений
* InvalidCastException: генерируется при попытке произвести недопустимые преобразования типов
* NullReferenceException: генерируется при попытке обращения к объекту, который равен null (то есть по сути неопределен)

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.

Например, в нашей программе происходит ввод имени пользователя, и мы хотим, чтобы, если длина имени меньше 2 символов, то возникало исключение:

После оператора throw указывается объект исключения, через конструктор которого мы можем передать сообщение об ошибке. Естественно вместо типа Exception мы можем использовать объект любого другого типа исключений.

Затем в блоке catch сгенерированное нами исключение будет обработано.

Подобным образом мы можем генерировать исключения в любом месте программы. Но существует также и другая форма использования оператора throw, когда после данного оператора не указывается объект исключения. В подобном виде оператор throw может использоваться только в блоке catch

В данном случае при вводе имени с длиной меньше 2 символов возникнет исключение, которое будет обработано внутренним блоком catch. Однако поскольку в этом блоке используется оператор throw, то исключение будет передано дальше внешнему блоку catch, который получит то же самое исключение и выведет то же самое сообщение на консоль.