1. Платформа .Net. Принципы. Сборщик мусора

.NET — это бесплатная кроссплатформенная платформа разработчика с открытым исходным кодом для создания различных типов приложений. Платформа .NET создана на основе высокопроизводительной среды выполнения, которая используется в рабочей среде многими высокомасштабными приложениями.

Функции .NET позволяют разработчикам эффективно писать надежный и производительный код.

- Асинхронный код
- Атрибуты
- Отражение
- Анализаторы кода
- Делегаты и лямбда-выражения
- События
- Исключения
- Сборка мусора
- Универсальные типы
- LINQ (языковой интегрированный запрос).
- Параллельное программирование
- Вывод типа — C#, F#, Visual Basic.
- Система типов
- Небезопасный код

Приложения и библиотеки .NET создаются из исходного кода и файла проекта с помощью интерфейса командной строки .NET или интегрированной среды разработки (IDE), такой как Visual Studio.
Среда CLR является основой для всех приложений .NET. Ниже перечислены основные функции среды выполнения.

- сборка мусора;
- Безопасность памяти и безопасность типов.
- Высокоуровневая поддержка языков программирования.
- Кроссплатформенный дизайн.

.NET иногда называют средой выполнения управляемого кода. Он называется управляемым главным образом потому, что использует сборщик мусора для управления памятью и обеспечивает безопасность типа и памяти.
Среда CLR была разработана как кроссплатформенная среда выполнения с самого начала. Он был перенесен в несколько операционных систем и архитектур.
Кроссплатформенный код .NET обычно не требуется перекомпилировать для выполнения в новых средах. Вместо этого необходимо просто установить другую среду выполнения для запуска приложения.
Среда выполнения предназначена для поддержки нескольких языков программирования. Языки C#, F# и Visual Basic поддерживаются корпорацией Майкрософт и разработаны совместно с сообществом.
Приложения .NET (написанные на высокоуровневом языке, например C#) компилируются в промежуточный язык (IL). IL — это компактный формат кода, который может поддерживаться в любой операционной системе или архитектуре. Большинство

приложений .NET используют API, поддерживаемые в нескольких средах, для выполнения которых требуется только среда выполнения .NET.

В .NET есть полный стандартный набор библиотек классов. Эти библиотеки предоставляют реализации для многих общих и зависящих от рабочей нагрузки типов, а также функциональные возможности.
Приложения .NET можно публиковать в двух разных режимах:

- Автономные приложения включают среду выполнения .NET и зависимые библиотеки. Это может быть однофайловая или многофайловая. Пользователи приложения могут запустить его на компьютере, на котором не установлена среда выполнения .NET. Автономные приложения всегда нацелены на единую конфигурацию операционной системы и архитектуры.
- Для приложений, зависящих от платформы, требуется совместимая версия среды выполнения .NET, обычно устанавливаемая глобально. Приложения, зависящие от платформы, можно публиковать для одной конфигурации операционной системы и архитектуры или как переносимые, предназначенные для всех поддерживаемых конфигураций.

При использовании ссылочных типов, например, объектов классов, для них 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 увидит, что на данный участок памяти больше нет ссылок, и очистит его.
Функционал сборщика мусора в библиотеке классов .NET представляет класс System.GC. Через статические методы данный класс позволяет обращаться к сборщику мусора. Как правило, надобность в применении этого класса отсутствует. Наиболее распространенным случаем его использования является сборка мусора при работе с неуправляемыми ресурсами, при интенсивном выделении больших объемов памяти, при которых необходимо такое же быстрое их освобождение.

2. Сборщик мусора.

При использовании ссылочных типов, например, объектов классов, для них 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 увидит, что на данный участок памяти больше нет ссылок, и очистит его.

```
Test();
void Test()
{
    Person tom = new Person("Tom");
    Console.WriteLine(tom.Name);
}
record class Person(string Name);
```

С помощью оператора new в куче для хранения объекта CLR выделяет участок памяти. А в стек добавляет адрес на этот участок памяти. В неявно определенном методе Main мы вызываем метод. И после того, как метод отработает, место в стеке очищается, а сборщик мусора очищает ранее выделенный под хранение объекта участок памяти.
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. 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.
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.
Класс System.GC
Функционал сборщика мусора в библиотеке классов .NET представляет класс System.GC. Через статические методы данный класс позволяет обращаться к сборщику мусора. Как правило, надобность в применении этого класса отсутствует. Наиболее распространенным случаем его использования является сборка мусора при работе с неуправляемыми ресурсами, при интенсивном выделении больших объемов памяти, при которых необходимо такое же быстрое их освобождение.
Рассмотрим некоторые методы и свойства класса System.GC:

- Метод AddMemoryPressure информирует среду CLR о выделении большого объема неуправляемой памяти, которую надо учесть при планировании сборки мусора. В связке с этим методом используется метод RemoveMemoryPressure, который указывает CLR, что ранее выделенная память освобождена, и ее не надо учитывать при сборке мусора.

- Метод Collect приводит в действие механизм сборки мусора. Перегруженные версии метода позволяют указать поколение объектов, вплоть до которого надо произвести сборку мусора
- Метод GetGeneration(Object) позволяет определить номер поколения, к которому относится переданый в качестве параметра объект
- Метод GetTotalMemory возвращает объем памяти в байтах, которое занято в управляемой куче
- Метод WaitForPendingFinalizers приостанавливает работу текущего потока до освобождения всех объектов, для которых производится сборка мусора

```
// .....................................
long totalMemory = GC.GetTotalMemory(false);
GC.Collect();
GC.WaitForPendingFinalizers();
//.....................................
```

С помощью перегруженных версий метода GC.Collect можно выполнить более точную настройку сборки мусора. Так, его перегруженная версия принимает в качестве параметра число - номер поколения, вплоть до которого надо выполнить очистку. Например, GC.Collect(0) - удаляются только объекты поколения 0.
Еще одна перегруженная версия принимает еще и второй параметр - перечисление GCCollectionMode. Это перечисление может принимать три значения:

- Default: значение по умолчанию для данного перечисления (Forced)
- Forced: вызывает немедленное выполнение сборки мусора
- Optimized: позволяет сборщику мусора определить, является ли текущий момент оптимальным для сборки мусора

Например, немедленная сборка мусора вплоть до первого поколения объектов:

```
GC.Collect(1, GCCollectionMode.Forced);
```

3. Описание классов в C#. Создание объектов класса.

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.
Описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.
В принципе ранее уже использовались классы. Например, тип string, который представляет строку, фактически является классом. Или, например, класс Console, у которого метод WriteLine() выводит на консоль некоторую информацию.
По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова class:

```
class название_класса
{
    // содержимое класса
}
```

После слова class идет имя класса и далее в фигурных скобках идет собственно содержимое класса.

- Поля и методы класса

Класс может хранить некоторые данные. Для хранения данных в классе применяются поля. По сути поля класса - это переменные, определенные на уровне класса.
Кроме того, класс может определять некоторое поведение или выполняемые действия. Для определения поведения в классе применяются методы.
В отличие от переменных, определенных в методах, поля класса могут иметь модификаторы, которые указываются перед полем. Так, чтобы все поля были доступны вне класса поля определены с модификатором public.
При определении полей мы можем присвоить им некоторые значения. Если поля класса не инициализированы, то они получают значения по умолчанию. Для переменных числовых типов это число 0. Методы класса имеют доступ к его полям.

- Создание объектов класса

После определения класса мы можем создавать его объекты. Для создания объекта применяются конструкторы. По сути конструкторы представляют специальные методы, которые называются так же как и класс, и которые вызываются при создании нового объекта класса и выполняют инициализацию объекта. Общий синтаксис вызова конструктора:
new конструктор_класса(параметры_конструктора);
Сначала идет оператор new, который выделяет память для объекта, а после него идет вызов конструктора.
Если в классе не определено ни одного конструктора, то для этого класса автоматически создается пустой конструктор по умолчанию, который не принимает никаких параметров.

- Обращение к функциональности класса

Для обращения к функциональности класса - полям, методам (а также другим элементам класса) применяется точечная нотация точки - после объекта класса ставится точка, а затем элемент класса:
объект.поле_класса
объект.метод_класса(параметры_метода)

4. Реализация инкапсуляции в C#. Абстракция, интерфейс и реализация

Инкапсуляция определяется как процесс включения одного или нескольких элементов в физический или логический пакет. Скрытие внутреннего состояния и функций объекта и предоставление доступа только через открытый набор функций. Инкапсуляция в методологии объектно-ориентированного программирования препятствует доступу к деталям реализации.
Абстракция и инкапсуляция связаны с объектно-ориентированным программированием. Абстракция позволяет сделать соответствующую информацию видимой, а инкапсуляция позволяет программисту реализовать желаемый уровень абстракции.
Инкапсуляция реализуется с использованием спецификаторов доступа. Спецификатор доступа определяет область видимости члена класса. C # поддерживает следующие спецификации доступа -

- общий
- частный

- защищенный
- внутренний
- защищенный внутренний

1)Спецификатор общего доступа public

Открытый спецификатор доступа позволяет классу подвергать свои переменные-члены и функции-члены другим функциям и объектам. Доступ к любому публичному члену можно получить извне класса.

2) Спецификатор частного доступа private

Спецификатор частного доступа позволяет классу скрыть свои переменные-члены и функции-члены от других функций и объектов. Только функции одного класса могут обращаться к своим частным членам. Даже экземпляр класса не может получить доступ к своим частным членам.

3) Защищенный спецификатор доступа protected

Защищенный спецификатор доступа позволяет дочернему классу обращаться к переменным-членам и функциям-членам его базового класса. Таким образом, это помогает в реализации наследования.

4) Спецификатор внутреннего доступа internal

Спецификатор внутреннего доступа позволяет классу подвергать свои переменные-члены и функции-члены другим функциям и объектам в текущей сборке. Другими словами, любой член с внутренним спецификатором доступа может быть доступен из любого класса или метода, определенных в приложении, в котором определяется член.

5) Защищенный спецификатор внутреннего доступа internal protected

Защищенный спецификатор внутреннего доступа позволяет классу скрыть свои переменные-члены и функции-члены из других объектов и функций класса, кроме дочернего класса в одном приложении. Это также используется при реализации наследования. Protected internal можно увидеть внутри сборки или типов, производных от класса, в котором она определена (включая типы из других сборок).

Реализация инкапсуляции:

- приватные поля
- публичные методы доступа
- разделение внутреннего состояния и внешнего представления
- сокрытие реализации

Абстракция данных – это процесс сокрытия определенных деталей и показа пользователю только важной информации. Абстракция может быть достигнута либо с помощью абстрактных классов, либо с помощью интерфейсов.

Ключевое слово abstract используется для классов и методов:

- Абстрактный класс – это ограниченный класс, который нельзя использовать для создания объектов (для доступа к нему он должен быть унаследован от другого класса);
- Абстрактный метод: может использоваться только в абстрактном классе и не имеет тела. Тело предоставляется производным классом.

Абстрактный класс может иметь как абстрактные, так и обычные методы. Создавать объекты абстрактных классов нельзя. Для создания объектов должен использоваться класс наследник.

Интерфейсы

В C# можно добиться абстракции, используя интерфейсы. Interface – это полностью абстрактный класс, который может содержать только абстрактные методы и свойства

(без реализации). Правильнее писать имя интерфейса с буквы I, так легче распознать, что это интерфейс, а не класс. По умолчанию члены интерфейса являются abstract и public. Интерфейсы могут содержать свойства и методы, но не поля.
Чтобы получить доступ к методам интерфейса, интерфейс должен быть реализован в другом классе. Чтобы реализовать интерфейс, используйте «:» символ (так же, как с наследованием). Тело метода интерфейса предоставляется классом «реализовать». Обратите внимание, что не нужно использовать ключевое слово override при реализации интерфейса.

5. Модификаторы доступа к компонентам класса. Статические компоненты класса.

При использовании классов в языке C# используют модификаторы доступа, которые делятся на две категории:

- модификаторы, определяющие доступ ко всему классу в сборке и за ее пределами;
- модификаторы, определяющие доступ к члену (элементу) класса.

Модификаторы доступа для класса

Если в программе объявляется класс, то этот класс может иметь один из двух модификаторов доступа:

- public — в этом случае класс считается открытым. Он доступен любому коду любой сборки;
- internal — в этом случае класс считается доступным только в сборке, где он определен.

Модификатор доступа для класса ставится перед ключевым словом class. Класс можно объявлять без указания модификатора доступа. В этом случае, по умолчанию принимается internal модификатор для класса.

Модификаторы доступа для элемента класса

Если в классе объявляется элемент (метод, член данных и т.п.), то перед ним указывается модификатор доступа. Этот модификатор определяет уровень ограничения доступа к элементу класса. Язык C# предлагает следующие модификаторы доступа:

- private;
- protected;
- public;
- internal;
- protected internal.

Модификаторы доступа protected и protected internal используются при наследовании. Элемент класса также может быть объявлен без модификатора. В этом случае принимается private уровень доступа к элементу класса.

private

Если в классе объявляется элемент (метод, член данных, ссылки и т.д.) с модификатором доступа private, то для него действуют следующие правила доступа:

- элемент недоступен любым экземплярам (объектам) класса;
- элемент недоступен любым унаследованным классам;
- элемент доступен методам класса в котором он объявлен.

public

При объявлении элемента класса с ключевым словом public действуют следующие правила:

- элемент считается доступным всем методам текущей сборки;

- элемент доступен из методов внешних сборок, если класс элемента объявлено как public. Если класс, с public-элементом объявлен как internal, то этот элемент не доступен из методов внешних сборок.

protected

Если в классе объявляется элемент (метод, член данных и т.п.) с модификатором protected, то действуют следующие ограничения доступа:

- элемент недоступен из экземпляра класса;
- элемент доступен из методов и вложенных классов, реализованных в текущем классе;
- элемент доступен из методов унаследованного класса.

internal

Если элемент (метод, член данных и т.д.) класса объявлен с ключевым словом internal, то действуют следующие правила доступа:

- элемент класса доступен из всех методов текущей сборки. В текущей сборке действие модификатора internal совпадает с действием модификатора public;
- элемент класса недоступен из методов других сборок.

protected internal

Если в классе объявляется элемент с модификатором доступа protected internal, то действуют следующие правила:

- элемент класса доступен из всех методов текущей сборки. Здесь действует расширение модификатора internal;
- элемент класса доступен из методов унаследованных классов независимо от сборки. Здесь действует расширение модификатора protected. Классы могут быть унаследованы в других сборках;
- элемент класса недоступен из методов других сборок (экземпляров класса). Здесь действует ограничение модификатора internal.

Статические компоненты класса

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. На уровне памяти для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса. При этом память для статических переменных выделяется даже в том случае, если не создано ни одного объекта этого класса.

Подобным образом мы можем создавать и использовать статические свойства.

```
public static int RetirementAge
   {
      get { return retirementAge; }
      set { if (value > 1 && value < 100) retirementAge = value; }
   }
```

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:

```
Person bob = new(68);
```

```
Person.CheckRetirementStatus(bob);
class Person
{
   public int Age { get; set; }
   static int retirementAge = 65;
   public Person(int age) => Age = age;
   public static void CheckRetirementStatus(Person person)
   {
      if (person.Age >= retirementAge)
         Console.WriteLine("Уже на пенсии");
      else
         Console.WriteLine($"Сколько лет осталось до пенсии: {retirementAge -
person.Age}") ;
   }
}
```

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

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

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

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

```
Console.WriteLine(Person.RetirementAge);
class Person
{
   static int retirementAge;
   public static int RetirementAge => retirementAge;
   static Person()
   {
      if (DateTime.Now.Year == 2022)
         retirementAge = 65;
      else
         retirementAge = 67;
   }
}
```

6. Конструкторы. Порядок вызова конструкторов. Ключевое слово this в C#. Деструкторы.

Конструктор - это специальный метод класса, который автоматически вызывается всякий раз, когда создается экземпляр класса. Как и методы, конструктор также содержит набор инструкций, которые выполняются во время создания объекта. Он используется для присвоения начальных значений элементам данных одного и того же класса.
На уровне кода конструктор представляет метод, который называется по имени класса, который может иметь параметры, но для него не надо определять возвращаемый тип. Конструкторы могут иметь модификаторы, которые указываются перед именем конструктора.
Типы конструкторов

- Конструктор по умолчанию
- Параметризованный конструктор
- Конструктор копирования
- Частный конструктор (Другие классы не могут быть производными от этого класса, а также невозможно создать экземпляр этого класса.)
- Статический конструктор (Статический конструктор должен вызываться в классе только один раз, и он был вызван во время создания первой ссылки на статический член в классе. Статический конструктор - это инициализированные статические поля или данные класса, которые должны выполняться только один раз.)

Ключевое слово this.
Ключевое слово this представляет ссылку на текущий экземпляр/объект класса. В каких ситуациях оно нам может пригодиться?

```
Person sam = new("Sam", 25);
sam.Print();          // Имя: Sam  Возраст: 25
class Person
{
    public string name;
    public int age;
    public Person() { name = "Неизвестно"; age = 18; }
    public Person(string name) { this.name = name; age = 18; }
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");
}
```

В примере выше во втором и третьем конструкторе параметры называются также, как и поля класса. И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово this. Так, в выражении this.name = name;
первая часть - this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно. Также через ключевое слово this можно обращаться к любому полю или методу.
Цепочка вызовов конструктора

В примере выше определены три конструктора. Все три конструктора выполняют однотипные действия - устанавливают значения полей name и age. Но этих повторяющихся действий могло быть больше. И мы можем не дублировать функциональность конструкторов, а просто обращаться из одного конструктора к другому также через ключевое слово this, передавая нужные значения для параметров:

```
class Person
{
    public string name;
    public int age;
    public Person() : this("Неизвестно")    // первый конструктор
    { }
    public Person(string name) : this(name, 18) // второй конструктор
    { }
    public Person(string name, int age)     // третий конструктор
    {
        this.name = name;
        this.age = age;
    }
    public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");
}
```

В данном случае первый конструктор вызывает второй, а второй конструктор вызывает третий. По количеству и типу параметров компилятор узнает, какой именно конструктор вызывается. Например, во втором конструкторе:

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

идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

Инициализаторы

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 };
// или так
// Person tom = new() { name = "Tom", age = 31 };
tom.Print();          // Имя: Tom  Возраст: 31
```

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания. При использовании инициализаторов следует учитывать следующие моменты:

- С помощью инициализатора мы можем установить значения только доступных из вне класса полей и свойств объекта. Например, в примере выше поля name и age имеют модификатор доступа public, поэтому они доступны из любой части программы.
- Инициализатор выполняется после конструктора, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, заменяются значениями из инициализатора.

Деконструкторы

Деконструкторы (не путать с деструкторами) позволяют выполнить декомпозицию объекта на отдельные части.
Например, пусть у нас есть следующий класс Person:

```
class Person
{
   string name;
   int age;
   public Person(string name, int age)
   {
      this.name = name;
      this.age = age;
   }

   public void Deconstruct(out string personName, out int personAge)
   {
      personName = name;
      personAge = age;
   }
}
```

В этом случае мы могли бы выполнить декомпозицию объекта Person так:

```
Person person = new Person("Tom", 33);
(string name, int age) = person;
Console.WriteLine(name);    // Tom
Console.WriteLine(age);     // 33
```

Значения переменным из деконструктора передаюся по позиции. То есть первое возвращаемое значение в виде параметра personName передается первой переменной - name, второе возвращаемое значение - переменной age. По сути деконструкторы это не более чем синтаксический сахар.
При получении значений из деконструктора нам необходимо предоставить столько переменных, сколько деконструктор возвращает значений. Однако бывает, что не все эти значения нужны. И вместо возвращаемых значений мы можем использовать прочерк _.
Деструкторы
в C# есть деструкторы и описывается он как метод, начинающийся со знака ~ и имеющий название класса. Своего рода конструктор, только наоборот и без модификаторов доступа и параметров.

```
~MyClass() { //Код деструктора }
```

Деструкторы можно определить только в классах. Деструктор в отличие от конструктора не может иметь модификаторов доступа и параметры. При этом каждый класс может иметь только один деструктор.
Однако, если капнуть глубже, то оказывается, что деструктор является синтаксическим сахаром и компилятор "разворачивает" деструктор в следующую конструкцию, где идёт вызов метода Finalize класса Object:

```
protected override void Finalize()
{
  try
  {
    //код деструктора
  }
```

```
  finally
  {
    base.Finalize();
  }
}
```
Метод Finalize уже определен в базовом для всех типов классе Object, однако данный метод нельзя так просто переопределить. И фактическая его реализация происходит через создание деструктора.
На уровне памяти это выглядит так: сборщик мусора при размещении объекта в куче определяет, поддерживает ли данный объект метод Finalize. И если объект имеет метод Finalize, то указатель на него сохраняется в специальной таблице, которая называется очередь финализации. Когда наступает момент сборки мусора, сборщик видит, что данный объект должен быть уничтожен, и если он имеет метод Finalize, то он копируется в еще одну таблицу и окончательно уничтожается лишь при следующем проходе сборщика мусора.
Интерфейс IDisposable объявляет один единственный метод Dispose, в котором при реализации интерфейса в классе должно происходить освобождение неуправляемых ресурсов. Конструкцию try...finally предпочтительнее использовать при вызове метода Dispose, так как она гарантирует, что даже в случае возникновения исключения произойдет освобождение ресурсов в методе Dispose.

7. Основные типы данных. Булевские типы и операции. Числовые типы. Строки и символы.

Как и во многих языках программирования, в C# есть своя система типов данных, которая используется для создания переменных. Тип данных определяет внутреннее представление данных, множество значений, которые может принимать объект, а также допустимые действия, которые можно применять над объектом.
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
  ```
  uint a = 10;
  uint b = 0b101;
  uint c = 0xFF;
  ```

- 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
  long a = -10;
  long b = 0b101;
  long c = 0xFF;
- 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.

Использование суффиксов

При присвоении значений надо иметь в виду следующую тонкость: все вещественные литералы (дробные числа) рассматриваются как значения типа double. И чтобы указать, что дробное число представляет тип float или тип decimal, необходимо к литералу добавлять суффикс: F/f - для float и M/m - для decimal.

```
float a = 3.14F;
float b = 30.6f;
decimal c = 1005.8M;
decimal d = 334.8m;
```

Подобным образом все целочисленные литералы рассматриваются как значения типа int. Чтобы явным образом указать, что целочисленный литерал представляет значение типа uint, надо использовать суффикс U/u, для типа long - суффикс L/l, а для типа ulong - суффикс UL/ul:

```
uint a = 10U;
long b = 20L;
ulong c = 30UL;
```

Неявная типизация

Ранее мы явным образом указывали тип переменных, например, int x;. И компилятор при запуске уже знал, что x хранит целочисленное значение.

Однако мы можем использовать и модель неявной типизации:

```
var hello = "Hell to World";
var c = 20;
```

Для неявной типизации вместо названия типа данных используется ключевое слово var. Затем уже при компиляции компилятор сам выводит тип данных исходя из присвоенного значения. Так как по умолчанию все целочисленные значения рассматриваются как

значения типа int, то поэтому в итоге переменная c будет иметь тип int. Аналогично переменной hello присваивается строка, поэтому эта переменная будет иметь тип string
Эти переменные подобны обычным, однако они имеют некоторые ограничения.
Во-первых, мы не можем сначала объявить неявно типизируемую переменную, а затем инициализировать:

```
// этот код работает
int a;
a = 20;
// этот код не работает
var c;
c= 20;
```

Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной null:

```
// этот код не работает
var c=null;
```

Так как значение null, то компилятор не сможет вывести тип данных.

Логические операторы

Унарные:

- логическое отрицание (!)

Бинарные:

- логическое И (&),//Это оператор всегда обрабатывает оба операнда и возвращает True только в том случае, если оба оператора равны True
- логическое ИЛИ (|), // оператор вернет True, если хотя бы один операнд будет иметь значение True
- логическое исключающее ИЛИ (^). // возвращает True только в том случае, если левый и правы операторы не равны
- условное логическое И (&&)//оператор обрабатывает правый операнд только тогда, когда это необходимо. Действие этого оператора следующее — он возвращает True только тогда, когда оба оператора равны True
- условное логическое ИЛИ (||)

Операторы равенства:

- равенство (==)
- неравенство (!=)

Операторы сравнения

- меньше чем (<),
- больше чем (>),
- меньше или равно (<=)
- больше или равно (>=)

8. Ссылочные и значимые типы. Boxing/Unboxing. Класс object.

Ссылочные и значимые типы

Все типы данных можно разделить на типы значений, еще называемые значимыми типами, (value types) и ссылочные типы (reference types). Важно понимать между ними различия.

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)

В чем же между ними различия? Для этого надо понять организацию памяти в .NET. Здесь память делится на два типа: стек и куча (heap).

Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу вверх: каждый новый добавляемый элемент помещается поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве. Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.

Ссылочные типы хранятся в куче или хипе, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу. 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.

Boxing/Unboxing

Упаковка и распаковка - важные понятия в C #. Система типов C # содержит три типа данных: типы значений (int, char и т.д.), ссылочные типы (object) и типы указателей. По сути, упаковка преобразует переменную типа значения в переменную ссылочного типа, а распаковка приводит к обратному результату. Упаковка и распаковка обеспечивают унифицированное представление системы типов, в которой значение любого типа может обрабатываться как объект.

- Упаковка на C#

Процесс преобразования переменной в переменную ссылочного типа (object) называется "Упаковка". Упаковка - это неявный процесс преобразования, в котором используется тип объекта (super type). Переменные типа значения всегда хранятся в стековой памяти, в то время как переменные ссылочного типа хранятся в памяти кучи.
Пример:

```
int num = 23; // 23 will assigned to num
Object Obj = num; // Boxing
```

Описание: Сначала мы объявляем переменную типа значения num типа int и инициализируем ее значением 23. Теперь мы создаем переменную ссылочного типа obj типа Object и присваиваем ей число. Это присвоение неявно приводит к тому, что переменная типа значения num копируется и сохраняется в переменной ссылочного типа obj.

- Распаковка на C#

Процесс преобразования переменной ссылочного типа в переменную типа-значения известен как, так называемая, "Распаковка".
Это явный процесс преобразования.
Пример :

```
int num = 23;          // value type is int and assigned value 23
Object Obj = num;     // Boxing
int i = (int)Obj;     // Unboxing
```

Описание : Мы объявляем переменную типа значения num, которая имеет тип int, и присваиваем ей целое значение 23. Теперь мы создаем переменную ссылочного типа obj типа Object, в которую мы помещаем переменную num. Теперь мы создаем значение типа integer i, чтобы распаковать значение из obj. Это делается с помощью метода приведения, в котором мы явно указываем, что obj должен быть приведен как значение int . Таким образом, переменная ссылочного типа, находящаяся в памяти кучи, копируется в стек.
Класс object.
Все классы в .NET, даже те, которые мы сами создаем, а также базовые типы, такие как System.Int32, являются неявно производными от класса Object. Даже если мы не указываем класс Object в качестве базового, по умолчанию неявно класс Object все равно стоит на вершине иерархии наследования. Поэтому все типы и классы могут реализовать те методы, которые определены в классе System.Object. Рассмотрим эти методы.

- ToString

Метод ToString служит для получения строкового представления данного объекта. Для базовых типов просто будет выводиться их строковое значение:
int i = 5;
Console.WriteLine(i.ToString()); // выведет число 5
Для классов же этот метод выводит полное название класса с указанием пространства имен, в котором определен этот класс. И мы можем переопределить данный метод.
Для переопределения метода ToString() в классе используется ключевое слово override (как и при обычном переопределении виртуальных или абстрактных методов).
Стоит отметить, что различные технологии на платформе .NET активно используют метод ToString для разных целей. В частности, тот же метод Console.WriteLine() по умолчанию выводит именно строковое представление объекта. Поэтому, если нам надо вывести строковое представление объекта на консоль, то при передаче объекта в метод Console.WriteLine необязательно использовать метод ToString() - он вызывается неявно

- Метод GetHashCode

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

```
class Person
{
   public string Name { get; set; } = "";
        public override int GetHashCode()
         {
             return Name.GetHashCode();
          }
}
```

В данном случае метод GetHashCode возвращает хеш-код для значения свойства Name. То есть два объекта Person, которые имеют одно и то же имя, будут возвращать один и тот же хеш-код. Однако в реальности алгоритм может быть самым различным.

- Получение типа объекта и метод GetType

Метод GetType позволяет получить тип данного объекта:

```
Person person = new Person { Name = "Tom" };
Console.WriteLine(person.GetType());    // Person
```

Этот метод возвращает объект Type, то есть тип объекта.

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

```
object person = new Person { Name = "Tom" };
if (person.GetType() == typeof(Person))
   Console.WriteLine("Это реально класс Person");
```

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

Стоит отметить, что проверку типа в примере выше можно сократить с помощью оператора is:

```
object person = new Person { Name = "Tom" };
if (person is Person)
   Console.WriteLine("Это реально класс Person");
```

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

- Метод Equals

Метод Equals позволяет сравнить два объекта на равенство. В качестве параметра он принимает объект для сравнения в виде типа object и возвращает true, если оба объекта равны:

```
public override bool Equals(object? obj) {.....}
```

Например, реализуем данный метод в классе Person:

```
class Person
{
        public string Name { get; set; } = "";
        public override bool Equals(object? obj)
          {
```

```
            // если параметр метода представляет тип Person
            // то возвращаем true, если имена совпадают
            if (obj is Person person) return Name == person.Name;
            return false;
        }
        // вместе с методом Equals следует реализовать метод GetHashCode
        public override int GetHashCode() => Name.GetHashCode();
    }
```

Метод Equals принимает в качестве параметра объект любого типа, который мы затем приводим к текущему классу - классу Person.
Если переданный объект представляет тип Person, то возвращаем результат сравнения имен двух объектов Person. Если же объект представляет другой тип, то возвращается false.

9. Хранение данных в коллекциях и извлечение данных из коллекций. Использование массивов.

Самым примитивным способом хранения объектов в C# является использование массивов. Одной из основных проблем, с которой столкнется разработчик следуя такому подходу, является то, что массивы не предоставляют инструментов для динамического изменения размера. В языке C# есть два пространства имен для работы со структурами данных:
System.Collections;
System.Collections.Generic.
Первое из них – System.Collections предоставляет структуры данных для хранения объектов типа Object. У этого решения есть две основных проблемы – это производительность и безопасность типов. В настоящее время не рекомендуется использовать объекты классов из System.Collections.
Для решения указанных выше проблем Microsoft были разработаны коллекции с обобщенными типами (их ещё называют дженерики), они расположены в пространстве имен System.Collections.Generic. Суть их заключается в том, что вы не просто создает объект класса List, но и указываете, объекты какого типа будут в нем храниться, делается это так: List<T>, где T может быть int, string, double или какой-то ваш собственный класс.
Пространство *System.Collections.Generic* содержит большой набор коллекций, которые позволяют удобно и эффективно решать широкий круг задач. Ниже, в таблице, перечислены некоторые из обобщенных классов с указанием интерфейсов, которые они реализуют.

Обобщенный класс	Основные интерфейсы	Описание
List<T>	ICollection<T>, IEnumerable<T>, IList<T>	Список элементов с динамически изменяемым размером
Dictionary<TKey, TValue>	ICollection<T>, IDictionary<TKey, TValue>,	Коллекция элементов

	IEnumerable<T>	связанных через уникальный ключ
Queue<T>	ICollection, IEnumerable<T>	Очередь – список, работающий по алгоритму FIFO
Stack<T>	ICollection, IEnumerable<T>	Стэк – список, работающий по алгоритму LIFO
SortedList<TKey,TValue>	IComparer<T>, ICollection<KeyValuePair<TKey,TValue>>, IDictionary<TKey,TValue>	Коллекция пар “ключ-значение”, упорядоченных по ключу

Класс List<T>

Эта коллекция является аналогом типизированного массива, который может динамически расширяться. В качестве типа можно указать любой встроенный либо пользовательский тип.

- Создание объекта класса List<T>

Можно создать пустой список и добавить в него элементы позже, с помощью метода Add().

Либо воспользоваться синтаксисом, позволяющем указать набор объектов, который будет храниться в списке:

List<int> nums = new List<int> {1, 2, 3, 4, 5};

- Работа с объектами List<T>

Ниже приведены таблицы, в которых перечислены некоторые полезные свойства и методы класса List<T>.

Свойства класса List<T>

Count - Количество элементов в списке

Capacity - Емкость списка – количество элементов, которое может вместить список без изменения размера

Методы класса List<T>

Add(T) Добавляет элемент к списку

BinarySearch(T) Выполняет поиск по списку

Clear() Очистка списка

Contains(T) Возвращает true, если список содержит указанный элемент

IndexOf(T) Возвращает индекс переданного элемента

ForEach(Action<T>) Выполняет указанное действие для всех элементов списка

Insert(Int32, T) Вставляет элемент в указанную позицию

Find(Predicate<T>) Осуществляет поиск первого элемент, для которого выполняется заданный предикат
Remove(T) Удаляет указанный элемент из списка
RemoveAt(Int32) Удаляет элемент из заданной позиции
Sort() Сортирует список
Reverse() Меняет порядок расположения элементов на противоположный

Класс Dictionary<TKey,TValue>

Класс Dictionary реализует структуру данных Отображение, которую иногда называют Словарь или Ассоциативный массив. Идея довольно проста: в обычном массиве доступ к данным мы получаем через целочисленный индекс, в словаре используется ключ, который может быть числом, строкой или любым другим типом данных, который реализует метод GetHashCode(). При добавлении нового объекта в такую коллекцию для него указывается уникальный ключ, который используется для последующего доступа к нему.

- Создание объекта класса Dictionary

Пустой словарь:

```
var dict = new Dictionary<string, int>();
```

Словарь с набором элементов:

```
var prodPrice = new Dictionary<string, double>()
{
   ["bread"] = 23.3,
   ["apple"] = 45.2
};
```

- Работа с объектами Dictionary

Свойства класса Dictionary

Count - Количество объектов в словаре
Keys - Ключи словаря
Values - Значения элементов словаря

Методы класса Dictionary

Add(TKey, TValue) Добавляет в словарь элемент с заданным ключом и значением
Clear() Удаляет из словаря все ключи и значения
ContainsValue(TValue) Проверяет наличие в словаре указанного значения
ContainsKey(TKey) Проверяет наличие в словаре указанного ключа
GetEnumerator() Возвращает перечислитель для перебора элементов словаря
Remove(TKey) Удаляет элемент с указанным ключом
TryAdd(TKey, TValue) Метод, реализующий попытку добавить в словарь элемент с заданным ключом и значением
TryGetValue(TKey, TValue) Метод, реализующий попытку получить значение по заданному ключу

Извлечение данных

Нумератор, или перечислитель (enumerator) — доступный только для чтения однонаправленный курсор (forward-only cursor — курсор, движущийся только вперёд, без возможности обратного перемещения) перебирающий последовательность (sequence) значений. Нумератор представляет собой объект, реализующий интерфейс System.Collections.IEnumerator или System.Collections.Generic.IEnumerator<T>.
Инструкция foreach перебирает, или выполняет итерацию (iterate) над перечислимыми (enumerable) объектами. Перечислимые объекты — это логическое представление последовательностей. Перечислимый объект — это не курсор, о котором говорилось

выше, а объект, содержащий такой курсор, способный перемещаться по объекту и перебирать его. Перечислимый объект либо реализует интерфейс IEnumerable или IEnumerable<T>, либо содержит метод GetEnumerator, который возвращает нумератор (enumerator).
Интерфейс IEnumerator определяет базовый низкоуровневый протокол, посредством которого производится проход по элементам (перечисление) последовательности в одонаправленной манере. Коллекции обычно не реализуют перечислители, а предоставляют их через интерфейс IEnumerable.
Инструкция foreach предоставляет высокоуровневый и лаконичный способ перебрать перечислимый объект:

```
foreach (char c in "beer") Console.WriteLine (c);
```

Тоже самое можно сделать более низкоуровневым способом без использования инструкции foreach:

```
using (var enumerator = "beer".GetEnumerator())
   while (enumerator.MoveNext())
   {
      var element = enumerator.Current;
      Console.WriteLine (element);
   }
```

Если нумератор реализует интерфейс IDisposable, инструкция foreach действует также как инструкция using, автоматически уничтожая объект нумератора.

Массивы

Массив представляет набор однотипных данных. Объявление массива похоже на объявление переменной за тем исключением, что после указания типа ставятся квадратные скобки:

```
тип_переменной[] название_массива;
```

После определения переменной массива мы можем присвоить ей определенное значение:

```
int[] nums = new int[4];
```

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

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

Для обращения к элементам массива используются индексы. Индекс представляет номер элемента в массиве, при этом нумерация начинается с нуля, поэтому индекс первого элемента будет равен 0, индекс четвертого элемента - 3.
Каждый массив имеет свойство Length, которое хранит длину массива.
Для перебора массивов мы можем использовать различные типы циклов. Например, цикл foreach. Подобные действия мы можем сделать и с помощью цикл for. В то же время цикл for более гибкий по сравнению с foreach. Если foreach последовательно извлекает элементы контейнера и только для чтения, то в цикле for мы можем перескакивать на несколько элементов вперед в зависимости от приращения счетчика, а также можем изменять элементы. Также можно использовать и другие виды циклов, например, while.

Класс “Array”

Класс Array является неявным базовым классом для всех одномерных и многомерных массивов.

Массивы могут дублироваться с помощью метода Clone. Однако результатом будет поверхностная (неглубокая) копия, означающая копирование только памяти, в которой представлен сам массив. Если массив содержит элементы значимых типов, копируются сами значения, если же ссылочных типов — копируются только ссылки, давая в результате два массива, элементы которого ссылаются на одни и те же объекты. Чтобы создать глубокую копию, в которой объекты ссылочных типов дублируются, нужно пройти в цикле по массиву и клонировать каждый его элемент вручную. Эти же правила применяются и к остальным коллекциям.
Массив можно создать не только с помощью синтаксиса C#, но и динамически с помощью статического метода Array.CreateInstance. При этом можно указать тип элементов и ранг (количество измерений), а также создать массив с индексацией начинающейся не с нуля, за счет указания нижней границы:
Array a = Array.CreateInstance (typeof(string), 2);

- Статические методы GetValue и SetValue позволяют получать доступ к элементам в динамически созданном массиве (также работают и с обычными массивами). Метод SetValue генерирует исключение, если элемент имеет тип несовместимый с массивом.
  a.SetValue ("hi", 0); // эквивалент a[0] = "hi";
  a.SetValue ("there", 1); // эквивалент a[1] = "there";
  string s = (string) a.GetValue (0); // эквивалент s = a[0];
- Статический метод Clear выполняет очистку массива, при этом размер массива не изменяется, а его членам присваивается либо значение null (для ссылочных типов) либо значение по умолчанию для конкретного значимого типа (в противовес этому коллекция при использовании метода Clear сокращается до нуля).
- С помощью статического метода Array.ForEach можно выполнить перечисление:
  // Пример:
  Array.ForEach (new[] { 1, 2, 3 }, Console.WriteLine);

Класс Array содержит ряд статических методов для поиска в массиве:

- BinarySearch — поиск элемента в отсортированном массиве.
- IndexOf и LastIndex — поиск элемента в несортированном массиве. Методы выполняют перечисление массива и возвращают индекс первого (или последнего) элемента, совпавшего с заданным значением.
- Find, FindLast, FindIndex, FindLastIndex, FindAll, Exists иTrueForAll — поиска элемента/элементов, удовлетворяющих заданному предикату, в несортированном массиве. Предикат — это просто делегат, принимающий объект и возвращающий true или false.
- С помощью методов Sort можно выполнять сортировку массива:
- Метод Reverse изменяет порядок всех или части элементов на противоположный:
- Класс Array содержит четыре метода для выполнения поверхностного копирования: Clone, CopyTo, Copy и ConstrainedCopy.
- Статический метод Resize позволяет изменить размер массива. Пир этом исходный массив не изменяется: метод создает и возвращает новый массив.

10.Передача параметров функций. Модификаторы out и ref. Методы с переменным числом параметров.

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

Передача параметров по значению

Наиболее простой способ передачи параметров представляет передача по значению, по сути это обычный способ передачи параметров:

```
void Increment(int n)
{
    n++;
    Console.WriteLine($"Число в методе Increment: {n}");
}
int number = 5;
Console.WriteLine($"Число до метода Increment: {number}");
Increment(number);
Console.WriteLine($"Число после метода Increment: {number}");
```

При передаче аргументов параметрам по значению параметр метода получает не саму переменную, а ее копию и далее работает с этой копией независимо от самой переменной.

Так, выше при вызове метод Increment получает копию переменной number и увеличивает значение этой копии. Поэтому в самом методе Increment мы видим, что значение параметра n увеличилось на 1, но после выполнения метода переменная number имеет прежнее значение - 5. То есть изменяется копия, а сама переменная не изменяется.

Передача параметров по ссылке и модификатор ref

При передаче параметров по ссылке перед параметрами используется модификатор ref:

```
void Increment(ref int n)
{
    n++;
    Console.WriteLine($"Число в методе Increment: {n}");
}
int number = 5;
Console.WriteLine($"Число до метода Increment: {number}");
Increment(ref number);
Console.WriteLine($"Число после метода Increment: {number}");
```

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.

Выходные параметры. Модификатор out

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

```
void Sum(int x, int y, out int result)
{
   result = x + y;
}
```
Здесь результат возвращается не через оператор return, а через выходной параметр result.
Использование в программе:
```
void Sum(int x, int y, out int result)
{
   result = x + y;
}
int number;
Sum(10, 15, out number);
Console.WriteLine(number);   // 25
```
Причем, как и в случае с ref ключевое слово out используется как при определении метода, так и при его вызове.
Также обратите внимание, что методы, использующие такие параметры, обязательно должны присваивать им определенное значение.
Прелесть использования подобных параметров состоит в том, что по сути мы можем вернуть из метода не одно значение, а несколько. Например:
```
void GetRectangleData(int width, int height, out int rectArea, out int rectPerimetr)
{
   rectArea = width * height;       // площадь прямоугольника - произведение ширины на высоту
   rectPerimetr = (width + height) * 2; // периметр прямоугольника - сумма длин всех сторон
}
int area;
int perimetr;
GetRectangleData(10, 20, out area, out perimetr);
Console.WriteLine($"Площадь прямоугольника: {area}");       // 200
Console.WriteLine($"Периметр прямоугольника: {perimetr}");   // 60
```
При этом можно определять переменные, которые передаются out-параметрам в непосредственно при вызове метода. То есть мы можем сократить предыдущий пример следующим образом:
```
void GetRectangleData(int width, int height, out int rectArea, out int rectPerimetr)
{
   rectArea = width * height;
   rectPerimetr = (width + height) * 2;
}
GetRectangleData(10, 20, out int area, out int perimetr);
Console.WriteLine($"Площадь прямоугольника: {area}");       // 200
Console.WriteLine($"Периметр прямоугольника: {perimetr}");   // 60
```
При этом, если нам неизвестен тип значений, которые будут присвоены параметрам, то мы можем для их определения использовать оператор var

Входные параметры. Модификатор in

Кроме выходных параметров с модификатором out метод может использовать входные параметры с модификатором in. Модификатор in указывает, что данный параметр будет

передаваться в метод по ссылке, однако внутри метода его значение параметра нельзя будет изменить. Например, возьмем следующий метод:

```
void GetRectangleData(in int width, in int height, out int rectArea, out int rectPerimetr)
{
    //width = 25; // нельзя изменить, так как width - входной параметр
    rectArea = width * height;
    rectPerimetr = (width + height) * 2;
}
int w = 10;
int h = 20;
GetRectangleData(w, h, out var area, out var perimetr);
Console.WriteLine($"Площадь прямоугольника: {area}");       // 200
Console.WriteLine($"Периметр прямоугольника: {perimetr}");   // 60
```

В данном случае через входные параметры width и height в метод передаются значения, но в самом методе мы не можем изменить значения этих параметров, так как они определены с модификатором in.

Передача по ссылке в некоторых случаях может увеличить производительность, а использование оператора in гарантирует, что значения переменных, которые передаются параметрам, нельзя будет изменить в этом методе.

Методы с переменным числом параметров.

Иногда бывают случаи, когда метод нужно вызвать несколько раз и каждый раз в этот метод нужно передавать разное количество аргументов. Такую ситуацию можно обойти создав перегруженные реализации метода.

Но не всегда можно знать сколько аргументов будет передаваться в метод, в особенности, если количество аргументов может быть довольно большим. В этом случае метод объявляется с переменным количеством аргументов.

Общая форма объявления метода с переменным количеством аргументов имеет вид:

```
return_type MethodName(params type[] parameters)
{
    // ...
}
```

Использование переменного количества аргументов дает следующие преимущества:

- в один и тот же метод можно передавать разное количество аргументов. Благодаря этому уменьшается количество реализаций метода. Нет потребности в «перегруженных» реализациях метода;
- упрощается программный код объявления метода за счет обобщенного представления массива параметров с помощью модификатора params;
- число аргументов может быть любым и задаваться по ходу выполнения программы (например n аргументов). В случае «перегрузки» метода, варианты реализации метода с разным количеством аргументов есть известны заранее.

```
// метод, который находит максимальное значение между списком параметров
double Max(params double[] values)
{
  if (values.Length==0)
  {
    Console.WriteLine("Ошибка: нет аргументов в вызове метода");
    return 0;
  }
```

```
  double max = 0;
  for (int i = 0; i < values.Length; i++)
    if (max < values[i])
      max = values[i];
  return max;
}
```

Вызов метода из другого программного кода

```
double maximum;
// нахождение максимума между 3 значениями
maximum = Max(8.5, 9.3, 2.9); // maximum = 9.3
// нахождение максимума между 5 значениями
maximum = Max(0.3, 2.33, 12.91, 8.93, 7.55); // maximum = 12.91
// нахождение максимума между 7 значениями
double[] arguments = { 2.8, 3.6, 1.7, 0.9, 4.45, 7.32, 2.83 };
maximum = Max(arguments); // maximum = 7.32
```

Количество аргументов, которые передаются в метод можно определить с помощью свойства Length.

11.Операторы. Операторы объявления, выражений, выбора, итераций, перехода

Методы пользовательских типов состоят из операторов, которые выполняются последовательно. Часто используется операторный блок – последовательность операторов, заключённая в фигурные скобки. Иногда возникает необходимость в пустом операторе – он записывается как символ ; (точка с запятой).

Операторы объявления

К операторам объявления относятся операторы объявления переменных и операторы объявления констант. Для объявления локальных переменных метода применяется оператор следующего формата:

тип имя-переменной [= начальное-значение];

Здесь тип – тип переменной, имя-переменной – допустимый идентификатор, необязательное начальное-значение – литерал или выражение, соответствующее типу переменной. Локальные переменные методов не могут использоваться в вычислениях, не будучи инициализированы.

Если необходимо объявить несколько переменных одного типа, то идентификаторы переменных можно перечислить через запятую после имени типа. При этом для каждой переменной можно выполнить инициализацию.

```
int a;// простейший вариант объявления
int a = 20;// объявление с инициализацией
int a, b, c;// объявление однотипных переменных
int a = 20, b = 10; // инициализация нескольких переменных
```

Локальная переменная может быть объявлена без указания типа, с использованием ключевого слова var. В этом случае компилятор выводит тип переменной из обязательного выражения инициализации.

```
var x = 3;
var y = "Student";
var z = new Student();
```

Не стоит воспринимать переменные, объявленные с var, как некие универсальные контейнеры для данных любого типа. Все эти переменные строго типизированы. Так, переменная x в приведённом выше примере имеет тип int.
Оператор объявления константы имеет следующий синтаксис:
const тип-константы имя-константы = выражение;
Допустимый тип-константы – это числовой тип, bool, string, перечисление или произвольный ссылочный тип. Выражение, которое присваивается константе, должно быть полностью вычислимо на момент компиляции. Обычно в качестве выражения используется литерал соответствующего типа. Для ссылочных типов (за исключением string) единственно допустимым выражением является null. Как и при объявлении переменных, можно определить в одном операторе несколько однотипных констант:
const double Pi = 3.1415926, E = 2.718281828; const string Name = "Student";
const object locker = null;
Область доступа к переменной или константе ограничена операторным блоком, содержащим объявление:

```
{
        int i = 10;
}
Console.WriteLine(i); // ошибка компиляции, переменная i не доступна
```

Если операторные блоки вложены друг в друга, то внутренний блок не может содержать объявлений переменных, идентификаторы которых совпадают с переменными внешнего блока:

```
{
        int i = 10;
        {
                int i = 20; // ошибка компиляции
        }
}
```

Операторы выражений
Операторы выражений – это выражения, одновременно являющиеся допустимыми операторами:
–операция присваивания (включая инкремент и декремент);
–операция вызова метода или делегата;
–операция создания объекта;
–операция асинхронного ожидания.
Приведём несколько примеров операторов выражений:

```
x = 1 + 2; // присваивание
x++; // инкремент
Console.Write(x);  // вызов метода
new StringBuilder(); //создание объекта
await Task.Delay(1000); //асинхронное ожидание
```

Заметим, что при вызове конструктора или метода, возвращающего значение, результат их работы использовать не обязательно.
Операторы перехода
К операторам перехода относятся break, continue, goto, return, throw. Оператор break используется для выхода из операторного блока циклов и оператора switch. Оператор break выполняет переход на оператор за блоком.
Оператор continue располагается в теле цикла и применяется для запуска новой итерации

цикла. Если циклы вложены, то запускается новая итерация того цикла, в котором непосредственно располагается continue.
Оператор goto передаёт управление на помеченный оператор. Обычно данный оператор употребляется в форме goto метка, где метка – это допустимый идентификатор. Метка должна предшествовать помеченному оператору и заканчиваться двоеточием, отдельно описывать метки не требуется:

```
goto label;
. . .
label: A = 100;
```

Оператор goto и помеченный оператор должны располагаться в одном операторном блоке. Возможно использование оператора goto в одной из следующих форм:
goto case константа; goto default;
Данные формы обсуждаются при рассмотрении оператора switch.
Оператор return служит для завершения методов.
Оператор throw генерирует исключительную ситуацию.

Операторы выбора

Операторы выбора – это операторы if и switch. Оператор if в языке C# имеет следующий синтаксис:

```
if (условие) вложенный-оператор-1
[else
вложенный-оператор-2]
```

Здесь условие – это некоторое булево выражение, вложенный-оператор – оператор (за исключением оператора объявления) или операторный блок. Ветвь else является необязательной.
Оператор switch выполняет одну из групп инструкций в зависимости от значения тестируемого выражения. Синтаксис оператора switch:

```
switch (выражение)
{
        case константное-выражение-1: операторы оператор-перехода
        case константное-выражение-2: операторы оператор-перехода
        . . .
        [default:
        операторы оператор-перехода]
}
```

Тестируемое выражение должно возвращать значение одного из следующих типов: целочисленный тип (включая char), тип bool, перечисление, строка. При совпадении тестируемого и константного выражений выполняется соответствующая ветвь case. Если совпадения не обнаружено, то выполняется ветвь default (если она есть).

Операторы циклов

К операторам циклов относятся операторы for, while, do-while, foreach.
Для циклов с известным числом итераций используется оператор for:
for ([инициализатор]; [условие]; [итератор]) вложенный-оператор
Для целочисленных типов, типа bool и перечислений допустимо использовать соответствующие типы с поддержкой null (например, int?).
Здесь инициализатор задаёт начальное значение счётчика (или счётчиков) цикла. Для счётчика может использоваться существующая переменная или объявляться новая переменная, время жизни которой будет ограничено циклом (при этом вместо типа

переменной допустимо указать var). Цикл выполняется, пока булево условие истинно, а итератор определяет изменение счётчика цикла на каждой итерации.
Простейший пример использования цикла for:

```
for (int i = 0; i < 10; i++)
//i доступна только в цикле for
{
        Console.WriteLine(i);// вывод чисел от 0 до 9
}
```

В инициализаторе можно объявить и задать начальные значения для нескольких счётчиков одного типа. В этом случае итератор может представлять собой последовательность из нескольких операторов, разделённых запятой:

```
// цикл выполнится 5 раз, на последней итерации i = 4, j = 6
 for (int i = 0, j = 10; i < j; i++, j--)
{
        Console.WriteLine("i = {0}, j = {1}", i, j);
}
```

Если число итераций цикла заранее неизвестно, можно использовать цикл while или цикл do-while. Данные циклы имеют схожий синтаксис:

```
while (условие) вложенный-оператор

do
вложенный-оператор while (условие);
```

В обоих операторах цикла тело цикла выполняется, пока булево условие истинно. В цикле while условие проверяется в начале очередной итерации, а в цикле do-while – в конце. Таким образом, цикл do-while всегда выполнится, по крайней мере, один раз.
Обратите внимание: условие должно присутствовать обязательно. Для организации бесконечных циклов на месте условия можно использовать литерал true:

```
while (true) Console.WriteLine("Endless loop");
```

Для перебора элементов объектов перечисляемых типов (например, массивов) в C# существует специальный цикл foreach:

```
foreach (тип идентификатор in коллекция) вложенный-оператор
```

В заголовке цикла объявляется переменная, которая будет последовательно принимать значения элементов коллекции. Вместо указания типа этой переменной можно использовать ключевое слово var. Присваивание переменной новых значений допустимо, но не отражается на элементах коллекции.
Следующие операторы C# не попадают ни в одну из перечисленных выше категорий. Их синтаксис подробно рассматривается при изучении соответствующих разделов и тем.
–Операторы checked и unchecked позволяют описать блоки контролируемого и неконтролируемого контекстов вычислений.
–Оператор try (в различных формах) применяется для перехвата и обработки исключительных ситуаций.
–Оператор using используется при освобождении управляемых ресурсов.
–Оператор yield служит для создания итераторов.
–Оператор lock применяется для объявления критической секции.

12.Наследование в C#.

Наследование (inheritance) является одним из ключевых моментов ООП. Благодаря наследованию один класс может унаследовать функциональность другого класса.
В языке C# класс, который наследуется, называется базовым, а класс, который наследует, — производным. Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе, добавляя к ним свои собственные элементы.
Поддержка наследования в C# состоит в том, что в объявление одного класса разрешается вводить другой класс. Для этого при объявлении производного класса указывается базовый класс. При установке между классами отношения "является" строится зависимость между двумя или более типами классов. Базовая идея, лежащая в основе классического наследования, заключается в том, что новые классы могут создаваться с использованием существующих классов в качестве отправной точки.

```
class Employee : Person
{
}
```

После двоеточия мы указываем базовый класс для данного класса.
Таким образом, наследование реализует отношение is-a (является), объект класса Employee также является объектом класса Person.
Для любого производного класса можно указать только один базовый класс. В C# не предусмотрено наследование нескольких базовых классов в одном производном классе. (В этом отношении C# отличается от C++, где допускается наследование нескольких базовых классов. Данное обстоятельство следует принимать во внимание при переносе кода C++ в C#.) Тем не менее можно создать иерархию наследования, в которой производный класс становится базовым для другого производного класса. (Разумеется, ни один из классов не может быть базовым для самого себя как непосредственно, так и косвенно.) Но в любом случае производный класс наследует все члены своего базового класса, в том числе переменные экземпляра, методы, свойства и индексаторы.
По умолчанию все классы наследуются от базового класса Object, даже если мы явным образом не устанавливаем наследование. Поэтому выше определенные классы Person и Employee кроме своих собственных методов, также будут иметь и методы класса Object: ToString(), Equals(), GetHashCode() и GetType().
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. Например, следующий класс не допускает создание наследников:

```
sealed class Admin
{
}
```

- Нельзя унаследовать класс от статического класса.

Доступ к членам базового класса из класса-наследника

Производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами private protected (если базовый и производный класс находятся в одной сборке), public, internal (если базовый и производный класс находятся в одной сборке), protected и protected internal.

Ключевое слово base

```
class Person
{
   public string Name { get; set;}
   public Person(string name)
   {
      Name = name;
   }
   public void Print()
   {
      Console.WriteLine(Name);
   }
}
class Employee : Person
{
   public string Company { get; set; }
   public Employee(string name, string company)
      : base(name)
   {
      Company = company;
   }
}
```

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

Конструкторы в производных классах

Конструкторы не передаются производному классу при наследовании. И если в базовом классе не определен конструктор по умолчанию без параметров, а только конструкторы с параметрами (как в случае с базовым классом Person), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово base.
То есть в классе через ключевое слово base надо явным образом вызвать конструктор базового класса. Либо в качестве альтернативы мы могли бы определить в базовом классе конструктор без параметров. Тогда в любом конструкторе производного класса, где нет обращения конструктору базового класса, все равно неявно вызывался бы этот конструктор по умолчанию.

Порядок вызова конструкторов

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

13.Пространства имен. Директива using. Оператор ::

Обычно определяемые классы и другие типы в .NET не существуют сами по себе, а заключаются в специальные контейнеры - пространства имен. Пространства имен позволяют организовать код программы в логические блоки, поволяют объединить и отделить от остального кода некоторую функциональность, которая связана некоторой общей идеей или которая выполняет определенную задачу.
Для определения пространства имен применяется ключевое слово namespace, после которого идет название название пространства имен:

```
namespace имя_пространства_имен
{
   // содержимое пространства имен
}
```

Одни пространства имен могут содержать другие. Для обращения к этим классам вне пространства необходимо использовать всю цепочку пространств имен.
Начиная с .NET 6 и C# 10 можно определять пространства имен на уровне файла.
Полное имя класса с учетом пространства имен добавляет в код избыточность - особенно, если пространство имен содержит множество классов, которые мы хотим использовать. И чтобы не писать полное имя класса, мы можем просто подключить пространство имен с помощью директивы using.
К директиве using можно применить два модификатора:

- Модификатор global действует так же, как и добавление одной директивы using к каждому исходному файлу в проекте. Впервые этот модификатор появился в C# 10.
- Модификатор static импортирует элементы static и вложенные типы из одного типа, а не из всех типов в пространстве имен.

Оба модификатора можно применить вместе для импорта статических членов из определенного типа во всех исходных файлах проекта.
Область директивы using без модификатора global ограничена файлом, в котором она находится. Добавление модификатора global к директиве using означает, что директива using должна применяться ко всем файлам в компиляции (обычно это проект). Директива global using впервые появилась в C# 10.
Директива using static указывает тип, доступ к статическим членам и вложенным типам которого можно получить, не указывая имя типа. Синтаксис:

```
using static <fully-qualified-type-name>;
```

<fully-qualified-type-name> — это имя типа, на статические члены и вложенные типы которого можно ссылаться, не указывая имя типа. Если полное доменное имя (полное имя пространства имен вместе с именем типа) не указано, C# создает ошибку компилятора.
Директива using static импортирует тип, а не пространство имен. Затем все статические члены импортированного типа можно использовать, не снабжая их именем типа. В показанном ниже примере вызывается статический метод WriteLine класса Console без ссылки на тип:

```
using static System.Console;
WriteLine ("Hello");
```

Директива using static импортирует все доступные статические члены типа, включая поля, свойства и вложенные типы. Ее также можно применять к перечислимым типам, что приведет к импортированию их членов.

Таким образом, если мы импортируем следующий перечислимый тип:
using static System.Windows.Visibility;
то вместо Visibility.Hidden сможем указывать просто Hidden:
var textBox = new TextBox { Visibility = Hidden };
Если между несколькими директивами using static возникнет неоднозначность, тогда компилятор C# не сумеет вывести корректный тип из контекста и сообщит об ошибке.
Директива using может отображаться:

- В начале файла исходного кода перед объявлениями пространств имен и типов.
- В любом пространстве имен, но до любых пространств имен или типов, объявленных в этом пространстве имен, если только не используется модификатор global, если он используется, директива должна располагаться перед всеми объявлениями пространств имен и типов.

Импортирование пространства имен может привести к конфликту имен типов. Вместо полного пространства имен можно импортировать только конкретные типы, которые нужны, и назначать каждому типу псевдоним:
using PropertyInfo2 = System.Reflection.Propertylnfo;
class Program { PropertyInfo2 p; }
Псевдоним можно назначить целому пространству имен:
using R = System.Reflection;
class Program { R.Propertylnfo p; }
Оператор ::
Маркер :: выполняет уточнение псевдонима пространства имен. В этом примере мы уточняем, используя глобальное пространство имен (чаще всего такое уточнение можно встречать в автоматически генерируемом коде — оно направлено на устранение конфликтов имен):

```
namespace N
{
	class A
	{
		static void Main()
		{
			System.Console.WriteLine (new A.B());
			System.Console.WriteLine (new global::A.B());
		}
		public class B {}
	}
}
namespace A
{
	class B {}
}
```

Ниже приведен пример уточнения псевдонима:

```
extern alias Wl;
extern alias W2;
Wl::Widgets.Widget wl = new Wl::Widgets.Widget();
W2::Widgets.Widget w2 = new W2::Widgets.Widget();
```

14.Ключевое слово base в C#. Сокрытие методов при наследовании.

Ключевое слово base

Ключевое слово base похоже на ключевое слово this. Оно служит двум важным целям:
• доступ к функции-члену базового класса при ее переопределении в подклассе;
• вызов конструктора базового класса.
В приведенном ниже примере ключевое слово base в классе House используется для доступа к реализации Liability из Asset:

```
public class House : Asset
{
        public override decimal Liability => base.Liability + Mortgage;
}
```

С помощью ключевого слова base мы получаем доступ к свойству Liability класса Asset невиртуальным образом. Это значит, что мы всегда обращаемся к версии данного свойства из Asset независимо от действительного типа экземпляра во время выполнения.
Тот же самый подход работает в ситуации, когда Liability скрывается, а не переопределяется. (Получить доступ к скрытым членам можно также путем приведения к базовому классу перед обращением к члену.)

```
class Person
{
   public string Name { get; set;}
   public Person(string name)
   {
      Name = name;
   }
   public void Print()
   {
      Console.WriteLine(Name);
   }
}
class Employee : Person
{
   public string Company { get; set; }
   public Employee(string name, string company)
      : base(name)
   {
      Company = company;
   }
}
```

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

Конструкторы в производных классах

Конструкторы не передаются производному классу при наследовании. И если в базовом классе не определен конструктор по умолчанию без параметров, а только конструкторы с

параметрами (как в случае с базовым классом Person), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово base. То есть в классе через ключевое слово base надо явным образом вызвать конструктор базового класса. Либо в качестве альтернативы мы могли бы определить в базовом классе конструктор без параметров. Тогда в любом конструкторе производного класса, где нет обращения конструктору базового класса, все равно неявно вызывался бы этот конструктор по умолчанию.

Сокрытие методов при наследовании

Фактически скрытие метода/свойства представляет определение в классе-наследнике метода или свойства, которые соответствует по имени и набору параметров методу или свойству базового класса. Для скрытия членов класса применяется ключевое слово new.

```
class Person
{
   public string Name { get; set; }
   public Person(string name)
   {
      Name = name;
   }
   public void Print()
   {
      Console.WriteLine($"Name: {Name}");
   }
}
class Employee : Person
{
   public string Company { get; set; }
   public Employee(string name, string company)
            : base(name)
   {
      Company = company;
   }
   public new void Print()
   {
      Console.WriteLine($"Name: {Name}   Company: {Company}");
   }
}
```

Здесь определен класс Person, представляющий человека, и класс Employee, представляющий работника предприятия. Employee наследует от Person все свойства и методы. Но в классе Employee кроме унаследованных свойств есть также и собственное свойство Company, которое хранит название компании. И мы хотели бы в методе Print выводить информацию о компании вместе с именем на консоль. Для этого определяется метод Print с ключевым словом new, который скрывает реализацию данного метода из базового класса.

В каких ситуациях можно использовать скрытие? Например, в примере выше метод Print в базовом классе не является виртуальным, мы не можем его переопределить, но, допустим, нас не устраивает его реализация для производного класса, поэтому мы можем воспользоваться сокрытием, чтобы определить нужный нам функционал.

При этом если мы хотим обратиться именно к реализации свойства или метода в базовом классе, то опять же мы можем использовать ключевое слово base и через него обращаться к функциональности базового класса.
Подобным образом мы можем организовать скрытие свойств.

15.Классы, закрытые для наследования. Отличия классов от структур

В C# существует возможность создать классы наследование от которых невозможно. Делается это при помощи ключевого слова sealed (запечатанный).

```
public sealed class SealedClass
{…}
```

Наиболее вероятная ситуация, когда может понадобиться пометить класс или метод как sealed — это когда класс или метод обеспечивает внутренние действия библиотеки, класса или других разрабатываемых классов, поэтому вы уверены, что любая попытка переопределить некоторую его функциональность приведет к нестабильности кода.
Также можно помечать класс или метод как sealed из коммерческих соображений, чтобы предотвратить использование классов способом, противоречащим лицензионным соглашениям.

Отличия классов от структур

Структуры синтаксически очень похожи на классы, но существует принципиальное отличие, которое заключается в том, что класс – является ссылочным типом (reference type), а структуры – значимым типом (value type) (см. статью «Типы данных«).
Следовательно, классы всегда создаются в так называемой “куче” (heap), а структуры создаются в стеке (stack).
Но это справедливо в очень простых случаях, главное отличие структур и классов: структуры, указываемые в списке параметров метода, передаются по значению (то есть копируются), объекты классов — по ссылке. Именно это является главным различием в их поведении, а не то, где они хранятся. Примечание: структуру тоже можно передать по ссылке, используя модификаторы out и ref.
Чем больше вы будете использовать структуры вместо маленьких классов, тем менее затратным по ресурсам будет использование памяти.
Так же как и классы, структуры могут иметь поля, методы и конструкторы.
В отличие от классов, использование публичных полей в структурах в большинстве случаев не рекомендуется, потому что не существует способа контролирования значений в них. Например, кто-либо может установить значение минут или секунд более 60. Более правильный вариант в данном случае использовать свойства, а в конструкторе осуществить проверку:

```
using System;
namespace ConsoleApplication1
{
  struct Time
  {
    private int hours, minutes, seconds;
    public Time(int hh, int mm, int ss)
    {
      hours = hh % 24;
      minutes = mm % 60;
      seconds = ss % 60;
```

```
    }
    public int Hours()
    {
      return hours;
    }
  }
  class Program
  {
    static void Main()
    {
      Time t = new Time(30,69,59);
      Console.WriteLine(t.Hours());
      Console.ReadKey();
    }
  }
}
```

В результате будет напечатано число часов: 6 (остаток от деления 30 на 24).

Заменим конструктор Time(…) конструктором без параметров:

```
public Time()
{
  hours = 7;
  minutes = 4;
  seconds = 0;
}
```

Получим сообщение об ошибке:

«Структуры не могут содержать явных конструкторов без параметров»

Причина возникновения ошибки в том, что вы не можете использовать конструктор по умолчанию (без параметров) для структуры, потому что компилятор всегда генерирует его сам.

Что же касается класса, то компилятор создает конструктор по умолчанию только в том случае, если Вы его не создали. При объявлении класса нет проблем создать собственный конструктор без параметров (замените в программе ключевое слово struct на class, вы получите результат — 7).

Если Вы не хотите использовать значения по умолчанию, то можете инициализировать поля своими значениями в конструкторе с параметрами для инициализации.

Однако если в этом конструкторе не будет инициализировано какое-нибудь значение, компилятор не будет его инициализировать и покажет ошибку.

Два правила структур

Первое правило структуры: Всегда все переменные должны быть инициализированы.

В классах Вы можете инициализировать значение полей непосредственно при их объявлении. В структурах такого сделать нельзя, и поэтому данный код вызовет ошибку при компиляции. Поэтому:

Второе правило структуры: Нельзя инициализировать переменные в том месте, где они объявляются.

К особенностям структур можно отнести еще и тот факт, что вследствие того, что структуры являются значимым типом, то можно создать структуру без использования конструктора, например:

```
using System;
namespace ConsoleApplication1
{
  struct Time
  {
    public int hours, minutes, seconds;
  }
  class Program
  {
    static void Main()
    {
      Time t;
      t.hours=7;
      Console.WriteLine(t.hours.ToString());
      Console.ReadKey();
    }
  }
}
```

В таком случае, переменная t создается, но поля не будут инициализированы конструктором (нет оператора Time t = new Time();). Правда, теперь поля структуры придется объявлять только с модификатором public.

16.Операторы is и as. Класс Type. Использование typeof.

В некоторых случаях, когда при написании программ нужно определить тип данных во время выполнения программы. Это значит, что на момент компиляции не известно какой тип будут иметь те или иные данные. Для определения типа в языке C# разработан соответствующий механизм, который называется динамическая идентификация типов. Динамическая идентификация типов есть эффективной в следующих случаях:

- если классы наследуют друг друга или образовывают иерархию, то при наличии ссылки на базовый класс, можно определить тип объекта по этой ссылке (см. п. 5);
- если нужно заведомо определить результат операции приведения типов для того, чтобы избегнуть генерирования исключительной ситуации в случае ошибочного приведения;
- динамическая идентификация типов эффективно используется в рефлексии.

Для обеспечения динамической идентификации типов в языке C# введенны три оператора:

- оператор is – проверяет, совпадает ли тип выражения с заданным типом данных;
- оператор as – предназначен для избежания возникновения исключительной ситуации в случае неудачного приведения типов;
- оператор typeof. – используется для получения информации о заданном типе (классе).

Оператор is

Общая форма оператора is имеет следующий вид:

выражение is тип

где выражение есть обозначением некоторого выражения, описывающего объект;

тип – некоторый тип, с которым сравнивается тип выражения. Если выражение и тип имеют одинаковый тип данных, то результат операции будет true. В противном случае, результат операции is будет false.
Чаще всего оператор is используется в условии оператора if. В этом случае, общая форма оператора if с оператором is имеет следующий вид

```
if (выражение is тип)
{
   // операции, которые нужно выполнить, если типы совпадают
   // ...
}
else
{
   // операции, которые нужно выполнить, если типы не совпадают
   // ...
}
```

Преимущества использования динамической идентификации типов в особенности проявляются в случае наследования классов. Если есть ссылка на базовый класс, то по этой ссылке можно определить тип объекта в иерархии наследования.

Оператор as

Оператор as применяется в задачах, в которых осуществляются операции приведения типов. Если приведение типа есть неудачным, то генерируется исключительная ситуация. Оператор as предназначен для предотвращения возникновения исключительной ситуации в случае неудачного приведения типов.
Общая форма оператора as имеет вид:
выражение as тип
где выражение – отдельное выражение, которое приводится к типу тип. Если выражение приводится к типу тип корректно, то результатом операции as есть ссылка на тип. В противном случае результатом операции as есть пустая ссылка null.
С учетом вышесказанного, общая форма оператора as, который размещается в правой части оператора присваивания, имеет вид:
objName = выражение as тип;
где objName – имя объекта, который будет содержать результат. Результатом может быть null или ссылка на объект класса тип.

Класс Type

Рефлексия (reflection) в .NET – это процесс выявления метаданных (типов) во время выполнения программы в виде объектной модели. Другими словами, рефлексия – это средство для получения сведений о типе данных.
С помощью средств рефлексии (класс System.Type) можно получить данные о типах и их характеристике в сборке во время выполнения программы. Это дает следующие преимущества:

- можно создавать типы и вызвать их методы без предыдущих знаний об именах, которые помещаются в этих типах. Это реализует так называемая динамическая идентификация типов;
- не нужно на этапе компиляции знать информацию о типе, из которого будут вытягиваться метаданные. Достаточно знать только название типа. Название типа задается в строке String(), которая есть общедоступной в любом месте;
- представление типов, которые вытягиваются из сборки, в виде удобной объектной модели.

Чтобы в программе на C# использовать рефлексию, нужно подключить пространство имен System.Reflection.
Пространство имен System.Reflection имеет много типов для реализации рефлексии. Однако, наиболее важными есть следующие:

- Assembly – абстрактный класс. Он содержит статические методы работы со сборкой. Эти методы позволяют, например, загружать сборку;
- AssemblyName – это есть класс, который содержит информацию которая используется для идентификации сборки. Например: номер версии сборки, информация о культуре и т.п.;
- EventInfo – абстрактный класс. Содержит информацию о заданном событии;
- FieldInfo – абстрактный класс. Может содержать информацию о заданных членах данных класса;
- MemberInfo – абстрактный класс. Содержит информацию об общем поведении для классов (типов) EventInfo, FieldInfo, MethodInfo и PropertyInfo;
- MethodInfo – абстрактный класс. Содержит информацию о заданном методе;
- Module – абстрактный класс. Позволяет получить информацию о заданном модуле в случае многофайловой сборки;
- ParameterInfo – класс, который содержит информацию о заданном параметре в заданном методе;
- PropertyInfo – абстрактный класс. Содержит информацию о заданном свойстве.

Чтобы получить информацию о заданном классе, используются методы класса System.Type, которые возвращают результат в виде массивов или отдельных классов вышеприведенных типов (Assembly, MethodInfo, ParameterInfo и т.д.).
Класс System.Type есть полезным, когда нужно изучать метаданные некоторого типа (класса, интерфейса, структуры, перечисления, делегата). Класс System.Type инкапсулирует в себе тип данных. Методы класса System.Type возвращают типы из пространства имен System.Reflection.
Класс System.Type служит основой рефлексии. Благодаря методам класса System.Type можно определять информацию об используемых типах во время выполнения программы. Затем эту информацию можно анализировать, обрабатывать в зависимости от поставленной задачи.
В C# .NET существует три способа программного получения информации о типе:

- Использование метода System.Object.GetType(). В этом случае метод возвращает метаданные текущего объекта типа.
- Использование средства typeof(). В этом случае создавать объект типа (например, объект класса) не нужно. Достаточно иметь объявленный тип.
- Использование статического метода System.Type.GetType(). Этот подход не нуждается в объявлении типа, информацию о котором нужно получить. Достаточно знать только имя этого типа.

typeof
Типом может быть класс, интерфейс, структура, перечисление, делегат. Чтобы получить информацию о типе с помощью средства typeof нужно располагать информацией о типе на этапе компиляции.
Например, чтобы использовать этот способ для класса, не нужно предварительно создавать объект класса. В общем случае, строка, которая получает информацию о некотором классе, имеет следующий вид:

```
Type tp = typeof(ClassName);
```

где ClassName – имя некоторого класса, для которого создается объект с именем tp, содержащий информацию об этом классе.
Предварительно, в начале программного модуля должно быть подключено пространство имен System.Reflection:
using System.Reflection;
Пример. Пусть задан класс MathFunctions. Класс содержит 3 метода с именами Min2(), Min3(), Max2() и 3 внутренние переменные с именами a, b, c.
Тогда, получение информации о методах и внутреннииx переменных класса с использованием средства typeof() может быть следующим:

```
...
class Program
{
   static void Main(string[] args)
   {
      // получить значение типа
      Type tp = typeof(MathFunctions);
      // взять перечень методов из класса MathFunctions
      MethodInfo[] mi = tp.GetMethods();
      // получить названия методов
      string m1 = mi[0].Name; // m1 = "Min2"
      string m2 = mi[1].Name; // m2 = "Min3"
      string m3 = mi[2].Name; // m3 = "Max2"
      // взять перечень внутренних данных класса
      FieldInfo[] fi = tp.GetFields();
      string f1 = fi[0].Name; // название внутренней переменной a, f1 = "a"
      string f2 = fi[1].Name; // f2 = "b"
      string f3 = fi[2].Name; // f3 = "c"
      Console.WriteLine("Method1 = {0}", m1); // Method1 = Min2
      Console.WriteLine("Method2 = {0}", m2); // Method2 = Min3
      Console.WriteLine("Method3 = {0}", m3); // Method3 = Max2
      Console.WriteLine("Field1 = {0}", f1); // Field1 = a
      Console.WriteLine("Field2 = {0}", f2); // Field2 = b
      Console.WriteLine("Field3 = {0}", f3); // Field3 = c
   }
}

...
```

Важно: чтобы получить информацию о внутренних переменных и методах класса, эти переменные и методы должны быть общедоступными, то есть, объявлены с модификатором доступа public. В противном случае, скрытые внутренние переменные и скрытые методы есть невидимыми.

17.Полиморфизм. Виртуальные функции.

Слово полиморфизм означает наличие многих форм. В парадигме объектно-ориентированного программирования полиморфизм часто выражается как «один интерфейс, несколько функций».
Полиморфизм может быть статическим или динамическим. В статическом полиморфизме ответ на функцию определяется во время компиляции. В динамическом полиморфизме он решается во время выполнения.
Статический полиморфизм
Механизм связывания функции с объектом во время компиляции называется ранним связыванием. Он также называется статической привязкой. C # предоставляет два метода для реализации статического полиморфизма

- Перегрузка функций
- Перегрузка оператора

Перегрузка функции
Вы можете иметь несколько определений для одного и того же имени функции в той же области. Определение функции должно отличаться друг от друга по типам и / или количеству аргументов в списке аргументов. Вы не можете перегружать объявления функций, которые отличаются только возвращаемым типом.
В следующем примере показано использование функции print () для печати различных типов данных:

```
using System;
namespace PolymorphismApplication {
  class Printdata {
    void print(int i) {
      Console.WriteLine("Printing int: {0}", i );
    }
    void print(double f) {
      Console.WriteLine("Printing float: {0}" , f);
    }
    void print(string s) {
      Console.WriteLine("Printing string: {0}", s);
    }
    static void Main(string[] args) {
      Printdata p = new Printdata();
      // Call print to print integer
      p.print(5);
      // Call print to print float
      p.print(500.263);
      // Call print to print string
      p.print("Hello C++");
      Console.ReadKey();
    }
  }
}
```

Когда приведенный выше код компилируется и выполняется, он производит следующий результат:

Printing int: 5
Printing float: 500.263
Printing string: Hello C++

Динамический полиморфизм

C # позволяет создавать абстрактные классы, которые используются для обеспечения частичной реализации класса интерфейса. Реализация завершается, когда производный класс наследуется от него. Абстрактные классы содержат абстрактные методы, которые реализуются производным классом. Производные классы имеют более специализированную функциональность.

Вот правила об абстрактных классах:

- Вы не можете создать экземпляр абстрактного класса
- Вы не можете объявить абстрактный метод вне абстрактного класса
- Когда класс объявляется запечатанным, его нельзя унаследовать, абстрактные классы не могут быть объявлены герметичными.

Следующая программа демонстрирует абстрактный класс:

```
using System;
namespace PolymorphismApplication {
  abstract class Shape {
    public abstract int area();
  }
  class Rectangle:  Shape {
    private int length;
    private int width;
    public Rectangle( int a = 0, int b = 0) {
      length = a;
      width = b;
    }
    public override int area () {
      Console.WriteLine("Rectangle class area :");
      return (width * length);
    }
  }
  class RectangleTester {
    static void Main(string[] args) {
      Rectangle r = new Rectangle(10, 7);
      double a = r.area();
      Console.WriteLine("Area: {0}",a);
      Console.ReadKey();
    }
  }
}
```

Когда приведенный выше код компилируется и выполняется, он производит следующий результат:

```
Rectangle class area :
Area: 70
```

Виртуальные функции

Когда у вас есть функция, определенная в классе, который вы хотите реализовать в унаследованном классе (-ах), вы используете виртуальные функции. Виртуальные

функции могут быть реализованы по-разному в разных унаследованных классах, и вызов этих функций будет решаться во время выполнения. Динамический полиморфизм реализуется абстрактными классами и виртуальными функциями. Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечается модификатором virtual. Такие методы и свойства называют виртуальными. А чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором override. Переопределенный метод в классе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.
Следующая программа демонстрирует это:

```
using System;
namespace PolymorphismApplication {
  class Shape {
    protected int width, height;
    public Shape( int a = 0, int b = 0) {
      width = a;
      height = b;
    }
    public virtual int area() {
      Console.WriteLine("Parent class area :");
      return 0;
    }
  }
  class Rectangle: Shape {
    public Rectangle( int a = 0, int b = 0): base(a, b) {
    }
    public override int area () {
      Console.WriteLine("Rectangle class area :");
      return (width * height);
    }
  }
  class Triangle: Shape {
    public Triangle(int a = 0, int b = 0): base(a, b) {
    }
    public override int area() {
      Console.WriteLine("Triangle class area :");
      return (width * height / 2);
    }
  }
  class Caller {
    public void CallArea(Shape sh) {
      int a;
      a = sh.area();
      Console.WriteLine("Area: {0}", a);
    }
  }
  class Tester {
    static void Main(string[] args) {
      Caller c = new Caller();
```

```
        Rectangle r = new Rectangle(10, 7);
        Triangle t = new Triangle(10, 5);
        c.CallArea(r);
        c.CallArea(t);
        Console.ReadKey();
    }
  }
}
```

Когда приведенный выше код компилируется и выполняется, он производит следующий результат:

```
Rectangle class area:
Area: 70
Triangle class area:
Area: 25
```

18.Абстрактные классы. Различие между абстрактным классом и интерфейсом.

Абстрактные классы

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. Например, определим абстрактный класс, который представляет некое транспортное средство:

```
abstract class Transport
{
   public void Move()
   {
      Console.WriteLine("Транспортно средство движется");
   }
}
```

Транспортное средство представляет некоторую абстракцию, которая не имеет конкретного воплощения. То есть есть легковые и грузовые машины, самолеты, морские судна, кто-то на космическом корабле любит покататься, но как такового транспортного средства нет. Тем не менее все транспортные средства имеют нечто общее - они могут перемещаться. И для этого в классе определен метод Move, который эмулирует перемещение.

Но главное отличие абстрактных классов от обычных состоит в том, что мы НЕ можем использовать конструктор абстрактного класса для создания экземпляра класса. Например, следующим образом:

```
Transport tesla = new Transport();
```

Тем не менее абстрактные классы полезны для описания некоторого общего функционала, который могут наследовать и использовать производные классы:

```
Transport car = new Car();
Transport ship = new Ship();
Transport aircraft = new Aircraft();
car.Move();
ship.Move();
aircraft.Move();
abstract class Transport
{
    public void Move()
    {
        Console.WriteLine("Транспортное средство движется");
    }
}
// класс корабля
class Ship : Transport { }
// класс самолета
class Aircraft : Transport { }
// класс машины
class Car : Transport { }
```

В данном случае от класса Transport наследуются три класса, которые представляют различные типы транспортных средств. Тем не менее они имеют общую черту - они могут перемещаться с помощью метода Move().

Выше писалось, что мы не можем использовать конструктор абстрактного класса для создания экземпляра этого класса. Тем не менее такой класс также может определять конструкторы:

```
Transport car = new Car("машина");
Transport ship = new Ship("корабль");
Transport aircraft = new Aircraft("самолет");
car.Move();         // машина движется
ship.Move();        // корабль движется
aircraft.Move();    // самолет движется
abstract class Transport
{
    public string Name { get; }
    // конструктор абстрактного класса Transport
    public Transport(string name)
    {
        Name = name;
    }
    public void Move() =>Console.WriteLine($"{Name} движется");
}
// класс корабля
```

```
class Ship : Transport
{
    // вызываем конструктор базового класса
    public Ship(string name) : base(name) { }
}
// класс самолета
class Aircraft : Transport
{
    public Aircraft(string name) : base(name) { }
}
// класс машины
class Car : Transport
{
    public Car(string name) : base(name) { }
}
```

В данном случае в абстрактном классе Transport определен конструктор - с помощью параметра он устанавливает значение свойства Name, которое хранит название транспортного средства. И в этом случае производные классы должны в своих конструкторах вызвать этот конструктор.

Абстрактные члены классов

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.

Абстрактные методы

Например, выше в примере с транспортными средствами метод Move описывает передвижение транспортного средства. Однако различные типы транспорта перемещаются по разному - ездят по земле, летят по воздуху, плывут на воде и т.д. В этом случае мы можем сделать метод Move абстрактным, а его реализацию переложить на производные классы:

```
abstract class Transport
{
    public abstract void Move();
}
```

```
// класс корабля
class Ship : Transport
{
    // мы должны реализовать все абстрактные методы и свойства базового класса
    public override void Move()
    {
        Console.WriteLine("Корабль плывет");
    }
}
// класс самолета
class Aircraft : Transport
{
    public override void Move()
    {
        Console.WriteLine("Самолет летит");
    }
}
// класс машины
class Car : Transport
{
    public override void Move()
    {
        Console.WriteLine("Машина едет");
    }
}
```

Применение классов:

```
Transport car = new Car();
Transport ship = new Ship();
Transport aircraft = new Aircraft();
car.Move();         // машина едет
ship.Move();        // корабль плывет
aircraft.Move();    // самолет летит
```

Отказ от реализации абстрактных членов

Производный класс обязан реализовать все абстрактные члены базового класса. Однако мы можем отказаться от реализации, но в этом случае производный класс также должен быть определен как абстрактный:

```
Transport tesla = new Auto();
tesla.Move();           // легковая машина едет
abstract class Transport
{
    public abstract void Move();
}
// класс машины
abstract class Car :Transport{}
class Auto: Car
{
    public override void Move()
    {
```

```
        Console.WriteLine("легковая машина едет");
    }
}
```

В данном случае класс Car не реализует абстрактный метод Move базового класса Transport и поэтому также определен как абстрактный. Однако любые неабстрактные классы, производные от Car, все равно должны реализовать все унаследованные абстрактные методы и свойства.

Различие между абстрактным классом и интерфейсом

Абстрактный класс	Интерфейс
Он содержит как части объявления, так и части реализации.	Он содержит только объявление методов, свойств, событий или индексаторов. Начиная с C # 8, в интерфейсы также могут быть включены реализации по умолчанию.
Множественное наследование не достигается абстрактным классом.	Множественное наследование достигается интерфейсом.
Он содержит конструктор.	Он не содержит конструктора.
Он может содержать статические элементы.	Он не содержит статических членов.
Он может содержать различные типы модификаторов доступа, такие как public, private, protected и т.д.	Он содержит только модификатор общедоступного доступа, потому что все в интерфейсе является общедоступным.
Производительность абстрактного класса высокая.	Производительность интерфейса низкая, потому что требуется время для поиска фактического метода в соответствующем классе.
Он используется для реализации базовой идентичности класса.	Он используется для реализации периферийных возможностей класса.
Класс может использовать только один абстрактный класс.	Класс может использовать несколько интерфейсов.
Если многие реализации одного и того же типа и используют общее поведение, то лучше использовать абстрактный класс.	Если многие реализации используют только общие методы, то лучше использовать Interface.
Абстрактный класс может содержать методы, поля, константы и т.д.	Интерфейс может содержать только методы, свойства, индексаторы, события.
Ключевое слово “:” может использоваться для реализации абстрактного класса.	Ключевое слово “:” и ”," может быть использовано для реализации интерфейса.
Это может быть реализовано полностью, частично или не реализовано.	Это должно быть полностью реализовано.
Для объявления абстрактного класса мы используем ключевое слово abstract.	Для объявления интерфейса мы используем ключевое слово interface.

19.Интерфейсы в C#. Наследование интерфейсов. Сравнение и клонирование объектов.

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:

```
interface IMovable
{
    // константа
    const int minSpeed = 0;     // минимальная скорость
    // статическая переменная
    static int maxSpeed = 60;   // максимальная скорость
    // метод
    void Move();                // движение
    // свойство
    string Name { get; set; }   // название

    delegate void MoveHandler(string message);  // определение делегата для события
    // событие
    event MoveHandler MoveEvent;    // событие движения
}
```

В данном случае определен интерфейс IMovable, который представляет некоторый движущийся объект. Данный интерфейс содержит различные компоненты, которые описывают возможности движущегося объекта. То есть интерфейс описывает некоторый функционал, который должен быть у движущегося объекта.

Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами и свойствами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не имеет реализации, не принимает никаких параметров и ничего не возвращает.

То же самое в данном случае касается свойства Name. На первый взгляд оно похоже на автоматическое свойство. Но в реальности это определение свойства в интерфейсе, которое не имеет реализации, а не автосвойство.

Модификаторы доступа

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:

```
Console.WriteLine(IMovable.maxSpeed);   // 60
Console.WriteLine(IMovable.minSpeed);   // 0
```

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

```
interface IMovable
{
    public const int minSpeed = 0;     // минимальная скорость
    private static int maxSpeed = 60;   // максимальная скорость
    public void Move();
    protected internal string Name { get; set; }    // название
    public delegate void MoveHandler(string message);  // определение делегата для события
    public event MoveHandler MoveEvent;    // событие движения
}
```

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:

```
public interface IMovable
{
    void Move();
}
```

Реализация по умолчанию

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:

```
interface IMovable
{
    // реализация метода по умолчанию
    void Move()
    {
        Console.WriteLine("Walking");
    }
}
```

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:

```
interface IMovable
{
    // реализация метода по умолчанию
```

```
    void Move() => Console.WriteLine("Walking");
    // реализация свойства по умолчанию
    // свойство только для чтения
    int MaxSpeed { get { return 0; } }
}
```
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):
```
Console.WriteLine(IMovable.MaxSpeed);   // 60
IMovable.MaxSpeed = 65;
Console.WriteLine(IMovable.MaxSpeed);   // 65
double time = IMovable.GetTime(500, 10);
Console.WriteLine(time);    // 50
interface IMovable
{
    public const int minSpeed = 0;     // минимальная скорость
    private static int maxSpeed = 60;   // максимальная скорость
    // находим время, за которое надо пройти расстояние distance со скоростью speed
    static double GetTime(double distance, double speed) => distance / speed;
    static int MaxSpeed
    {
        get => maxSpeed;
        set
        {
            if (value > 0) maxSpeed = value;
        }
    }
```
Наследование интерфейсов

Интерфейсы, как и классы, могут наследоваться:
```
interface IAction
{
    void Move();
}
interface IRunAction : IAction
{
    void Run();
}
class BaseAction : IRunAction
{
    public void Move()
    {
        Console.WriteLine("Move");
    }
    public void Run()
    {
        Console.WriteLine("Run");
    }
}
```

При применении этого интерфейса класс BaseAction должен будет реализовать как методы и свойства интерфейса IRunAction, так и методы и свойства базового интерфейса IAction, если эти методы и свойства не имеют реализации по умолчанию.
Однако в отличие от классов мы не можем применять к интерфейсам модификатор sealed, чтобы запретить наследование интерфейсов.
Также мы не можем применять к интерфейсам модификатор abstract, поскольку интерфейс фактически итак, как правило, предоставляет абстрактный функционал, который должен быть реализован в классе или структуре (за исключением методов и свойств с реализацией по умолчанию).
Однако методы интерфейсов могут использовать ключевое слово new для скрытия методов из базового интерфейса:

```
IAction action1 = new RunAction();
action1.Move(); // I am moving
IRunAction action2 = new RunAction();
action2.Move(); // I am running

interface IAction
{
   void Move() => Console.WriteLine("I am moving");
}
interface IRunAction : IAction
{
   // скрываем реализацию из IAction
   new void Move() => Console.WriteLine("I am running");
}
class RunAction : IRunAction { }
```

Здесь метод Move из IRunAction скрывает метод Move из базового интерфейса IAction. Это имеет смысл, если в базовом интерфейсе определена реализация по умолчанию, как в случае выше, которую нужно переопределить.
При наследовании интерфейсов следует учитывать, что, как и при наследовании классов, производный интерфейс должен иметь тот же уровень доступа или более строгий, чем базовый интерфейс.

Сравнение и клонирование объектов.

Клонирование

Поскольку классы представляют ссылочные типы, то это накладывает некоторые ограничения на их использование. В частности, допустим, у нас есть следующий класс:

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

Создадим один объект Person и попробуем скопировать его данные в другой объект Person:

```
var tom = new Person("Tom", 23);
var bob = tom;
bob.Name = "Bob";
Console.WriteLine(tom.Name); // Bob
```
В данном случае объекты tom и bob будут указывать на один и тот же объект в памяти, поэтому изменения свойств для переменной bob затронут также и переменную tom. Чтобы переменная bob указывала на новый объект, но при этом имела значения из переменной tom, мы можем применить клонирование с помощью реализации интерфейса ICloneable:
```
public interface ICloneable
{
   object Clone();
}
```
- Поверхностное копирование

Реализация интерфейса в классе Person могла бы выглядеть следующим образом:
```
class Person : ICloneable
{
   public string Name { get; set; }
   public int Age { get; set; }
   public Person(string name, int age)
   {
      Name = name;
      Age = age;
   }
   public object Clone()
   {
      return new Person(Name, Age);
   }
}
```
Использование:
```
var tom = new Person("Tom", 23);
var bob = (Person)tom.Clone();
bob.Name = "Bob";
Console.WriteLine(tom.Name); // Tom
```
Теперь все нормально копируется, изменения в свойствах переменной bob не сказываются на свойствах из переменной tom.
Для сокращения кода копирования мы можем использовать специальный метод MemberwiseClone(), который возвращает копию объекта:
```
class Person : ICloneable
{
   public string Name { get; set; }
   public int Age { get; set; }
   public Person(string name, int age)
   {
      Name = name;
      Age = age;
   }
   public object Clone()
```

```
    {
        return MemberwiseClone();
    }
}
```
Этот метод реализует поверхностное (неглубокое) копирование. Однако данного копирования может быть недостаточно.

- Глубокое копирование

Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять глубокое копирование:
```
class Person : ICloneable
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Company Work { get; set; }
    public Person(string name, int age, Company company)
    {
        Name = name;
        Age = age;
        Work = company;
    }
    public object Clone() => new Person(Name, Age, new Company(Work.Name));
}
class Company
{
    public string Name { get; set; }
    public Company(string name) => Name = name;
}
```
Сравнение объектов

Интерфейсы IComparable определены следующим образом:
```
public interface IComparable { int CompareTo (object other); }
public interface IComparable<in T> { int CompareTo (T other); }
```
Оба интерфейса предоставляют одинаковую функциональность. Метод CompareTo (обоих интерфейсов) работает по следующим правилам:

- если a находится после b, a.CompareTo(b) возвращает положительное число
- если a и b одинаковые, a.CompareTo(b) возвращает 0
- если a находится перед b, a.CompareTo(b) возвращает отрицательное число

Большинство базовых типов реализуют оба интерфейса.

При реализации IComparable следует учитывать одно важное правило: эквивалентность может быть более придирчива, чем сравнение порядка, но не наоборот (если это нарушить, алгоритмы сортировки перестанут работать). Для типа переопределяющего Equals и реализующего IComparable, когда Equals возвращает true, CompareTo должен возвращать 0. Но когда Equals возвращает false, CompareTo может вернуть любое значение. Другими словами эквивалентные объекты всегда равны в плане порядка, но не эквивалентные объекты могут располагаться в разном порядке, в т.ч. быть равными по порядку. При реализации IComparable, чтобы не нарушить это правило, достаточно в первой строке метода CompareTo написать:

if (Equals (other)) return 0;
После этого можно возвращать то, что нравится.

20.Исключения (exception). Операторы try, throw, checked, unchecked. Фильтры исключений

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 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.
Определение блока catch
За обработку исключения отвечает блок catch, который может иметь следующие формы:

```
catch
{
    // выполняемые инструкции
}
```

Обрабатывает любое исключение, которое возникло в блоке try.

```
catch (тип_исключения)
{
    // выполняемые инструкции
}
```

Обрабатывает только те исключения, которые соответствуют типу, указаному в скобках после оператора catch.
Например, обработаем только исключения типа DivideByZeroException:

```
try
{
    int x = 5;
    int y = x / 0;
    Console.WriteLine($"Результат: {y}");
}
catch(DivideByZeroException)
{
    Console.WriteLine("Возникло исключение DivideByZeroException");
}
```

Однако если в блоке try возникнут исключения каких-то других типов, отличных от DivideByZeroException, то они не будут обработаны.

```
catch (тип_исключения имя_переменной)
{
```

```
    // выполняемые инструкции
}
```
Обрабатывает только те исключения, которые соответствуют типу, указаному в скобках после оператора catch. А вся информация об исключении помещается в переменную данного типа. Например:
```
try
{
    int x = 5;
    int y = x / 0;
    Console.WriteLine($"Результат: {y}");
}
catch(DivideByZeroException ex)
{
    Console.WriteLine($"Возникло исключение {ex.Message}");
}
```
Фактически этот случай аналогичен предыдущему за тем исключением, что здесь используется переменная. В данном случае в переменную ex, которая представляет тип DivideByZeroException, помещается информация о возникшем исключении. И с помощью свойства Message мы можем получить сообщение об ошибке.
Если нам не нужна информация об исключении, то переменную можно не использовать как в предыдущем случае.

Фильтры исключений

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

}
```
В этом случае обработка исключения в блоке catch производится только в том случае, если условие в выражении when истинно. Например:
```
int x = 1;
int y = 0;
try
{
    int result1 = x / y;
    int result2 = y / x;
}
catch (DivideByZeroException) when (y == 0)
{
    Console.WriteLine("y не должен быть равен 0");
}
catch(DivideByZeroException ex)
{
    Console.WriteLine(ex.Message);
}
```
В данном случае будет выброшено исключение, так как y=0. Здесь два блока catch, и оба они обрабатывают исключения типа DivideByZeroException, то есть по сути все

исключения, генерируемые при делении на ноль. Но поскольку для первого блока указано условие y == 0, то именно этот блок будет обрабатывать данное исключение - условие, указанное после оператора when возвращает true.

Exception

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.

- 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)

Генерация исключения и оператор throw

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:

```
try
{
    Console.Write("Введите имя: ");
    string? name = Console.ReadLine();
    if (name== null || name.Length < 2)
    {
        throw new Exception("Длина имени меньше 2 символов");
    }
    else
    {
        Console.WriteLine($"Ваше имя: {name}");
```

```
    }
}
catch (Exception e)
{
    Console.WriteLine($"Ошибка: {e.Message}");
}
```

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.

checked и unchecked

В C# допускается указывать, будет ли в коде сгенерировано исключение при переполнении, с помощью ключевых слов checked и unchecked. Так, если требуется указать, что выражение будет проверяться на переполнение, следует использовать ключевое слово checked, а если требуется проигнорировать переполнение — ключевое слово unchecked. В последнем случае результат усекается, чтобы не выйти за пределы диапазона представления чисел для целевого типа выражения.

У ключевого слова checked имеются две общие формы. В одной форме проверяется конкретное выражение, и поэтому она называется операторной. А в другой форме проверяется блок операторов, и поэтому она называется блочной. Ниже приведены обе формы:

```
checked (выражение)
checked {
// проверяемые операторы
}
```

где выражение обозначает проверяемое выражение. Если вычисление проверяемого выражения приводит к переполнению, то генерируется исключение OverflowException.

У ключевого слова unchecked также имеются две общие формы. В первой, операторной форме переполнение игнорируется при вычислении конкретного выражения. А во второй, блочной форме оно игнорируется при выполнении блока операторов:

```
unchecked (выражение)
unchecked {
// операторы, для которых переполнение игнорируется
}
```

где выражение обозначает конкретное выражение, при вычислении которого переполнение игнорируется. Если же в непроверяемом выражении происходит переполнение, то результат его вычисления усекается.

Давайте проиллюстрируем вышесказанное примером:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
```

```
        byte a, b, result;
        Console.Write("Введите количество опросов: ");
        int i = int.Parse(Console.ReadLine());

        for (int j = 1; j <= i; j++)
        {
            try
            {
                Console.Write("Введите a: ");
                // Используем unchecked в одном выражении
                a = unchecked((byte)int.Parse(Console.ReadLine()));
                Console.Write("Введите b: ");
                b = unchecked((byte)int.Parse(Console.ReadLine()));

                // Используем checked для всего блока операторов
                checked
                {
                    result = (byte)(a + b);
                    Console.WriteLine("a + b = " + result);
                    result = (byte)(a * b);
                    Console.WriteLine("a*b = " + result + "\n");
                }
            }
            catch (OverflowException)
            {
                Console.Write("Переполнение\n\n");
            }
        }
        Console.ReadLine();
    }
  }
}
```

Потребность в применении ключевого слова checked или unchecked может возникнуть, в частности, потому, что по умолчанию проверяемое или непроверяемое состояние переполнения определяется путем установки соответствующего параметра компилятора и настройки самой среды выполнения. Поэтому в некоторых программах состояние переполнения лучше проверять явным образом.

21.Перегрузка методов (overload). Перегрузка операторов.

Перегрузка методов

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

И в языке C# мы можем создавать в классе несколько методов с одним и тем же именем, но разной сигнатурой. Что такое сигнатура? Сигнатура складывается из следующих аспектов:

- Имя метода
- Количество параметров
- Типы параметров
- Порядок параметров
- Модификаторы параметров

Но названия параметров в сигнатуру НЕ входят. Например, возьмем следующий метод:

```
public int Sum(int x, int y)
{
    return x + y;
}
```

У данного метода сигнатура будет выглядеть так: Sum(int, int)

И перегрузка метода как раз заключается в том, что методы имеют разную сигнатуру, в которой совпадает только название метода. То есть методы должны отличаться по:

- Количеству параметров
- Типу параметров
- Порядку параметров
- Модификаторам параметров

Например, пусть у нас есть следующий класс:

```
class Calculator
{
    public void Add(int a, int b)
    {
        int result = a + b;
        Console.WriteLine($"Result is {result}");
    }
    public void Add(int a, int b, int c)
    {
        int result = a + b + c;
        Console.WriteLine($"Result is {result}");
    }
    public int Add(int a, int b, int c, int d)
    {
        int result = a + b + c + d;
        Console.WriteLine($"Result is {result}");
        return result;
    }
    public void Add(double a, double b)
    {
```

```
        double result = a + b;
        Console.WriteLine($"Result is {result}");
    }
}
```
Здесь представлены четыре разных версии метода Add, то есть определены четыре перегрузки данного метода.
Первые три версии метода отличаются по количеству параметров. Четвертая версия совпадает с первой по количеству параметров, но отличается по их типу. При этом достаточно, чтобы хотя бы один параметр отличался по типу. Поэтому это тоже допустимая перегрузка метода Add.
Также перегружаемые методы могут отличаться по используемым модификаторам. Например:
```
void Increment(ref int val)
{
    val++;
    Console.WriteLine(val);
}
void Increment(int val)
{
    val++;
    Console.WriteLine(val);
}
```
В данном случае обе версии метода Increment имеют одинаковый набор параметров одинакового типа, однако в первом случае параметр имеет модификатор ref. Поэтому обе версии метода будут корректными перегрузками метода Increment.
А отличие методов по возвращаемому типу или по имени параметров не является основанием для перегрузки.

Перегрузка операторов

В C#, подобно любому языку программирования, имеется готовый набор лексем, используемых для выполнения базовых операций над встроенными типами. Например, известно, что операция + может применяться к двум целым, чтобы дать их сумму:
```
// Операция + с целыми.
int a = 100;
int b = 240;
int c = a + b;          //c теперь равно 340
```
Здесь нет ничего нового, но задумывались ли вы когда-нибудь о том, что одна и та же операция + может применяться к большинству встроенных типов данных C#? Например, рассмотрим такой код:
```
// Операция + со строками.
string si = "Hello";
string s2 = " world!";
string s3 = si + s2; // s3 теперь содержит "Hello world!"
```
По сути, функциональность операции + уникальным образом базируются на представленных типах данных (строках или целых в данном случае). Когда операция + применяется к числовым типам, мы получаем арифметическую сумму операндов. Однако когда та же операция применяется к строковым типам, получается конкатенация строк.

Язык C# предоставляет возможность строить специальные классы и структуры, которые также уникально реагируют на один и тот же набор базовых лексем (вроде операции +). Имейте в виду, что абсолютно каждую встроенную операцию C# перегружать нельзя. Перегрузка операторов тесно связана с перегрузкой методов. Для перегрузки оператора служит ключевое слово operator, определяющее операторный метод, который, в свою очередь, определяет действие оператора относительно своего класса. Существуют две формы операторных методов (operator): одна - для унарных операторов, другая - для бинарных. Ниже приведена общая форма для каждой разновидности этих методов:

```
// Общая форма перегрузки унарного оператора.
public static возвращаемый_тип operator op(тип_параметра операнд)
{
// операции
}
// Общая форма перегрузки бинарного оператора.
public static возвращаемый_тип operator op(тип_параметра1 операнд1,
тип_параметра2 операнд2)
{
// операции
}
```

Здесь вместо op подставляется перегружаемый оператор, например + или /, а возвращаемый_тип обозначает конкретный тип значения, возвращаемого указанной операцией. Это значение может быть любого типа, но зачастую оно указывается такого же типа, как и у класса, для которого перегружается оператор. Такая корреляция упрощает применение перегружаемых операторов в выражениях. Для унарных операторов операнд обозначает передаваемый операнд, а для бинарных операторов то же самое обозначают операнд1 и операнд2. Обратите внимание на то, что операторные методы должны иметь оба спецификатора типа - public и static.

- Перегрузка бинарных операторов

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1
{
   class MyArr
   {
      // Координаты точки в трехмерном пространстве
      public int x, y, z;
      public MyArr(int x = 0, int y = 0, int z = 0)
      {
         this.x = x;
         this.y = y;
         this.z = z;
      }
      // Перегружаем бинарный оператор +
      public static MyArr operator +(MyArr obj1, MyArr obj2)
      {
         MyArr arr = new MyArr();
```

```
            arr.x = obj1.x + obj2.x;
            arr.y = obj1.y + obj2.y;
            arr.z = obj1.z + obj2.z;
            return arr;
        }
        // Перегружаем бинарный оператор -
        public static MyArr operator -(MyArr obj1, MyArr obj2)
        {
            MyArr arr = new MyArr();
            arr.x = obj1.x - obj2.x;
            arr.y = obj1.y - obj2.y;
            arr.z = obj1.z - obj2.z;
            return arr;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            MyArr Point1 = new MyArr(1, 12, -4);
            MyArr Point2 = new MyArr(0, -3, 18);
            Console.WriteLine("Координаты первой точки: " +
                Point1.x + " " + Point1.y + " " + Point1.z);
            Console.WriteLine("Координаты второй точки: " +
                Point2.x + " " + Point2.y + " " + Point2.z + "\n");

            MyArr Point3 = Point1 + Point2;
            Console.WriteLine("\nPoint1 + Point2 = "
                + Point3.x + " " + Point3.y + " " + Point3.z);
            Point3 = Point1 - Point2;
            Console.WriteLine("\nPoint1 - Point2 = "
                + Point3.x + " " + Point3.y + " " + Point3.z);

            Console.ReadLine();
        }
    }
}
```

- Перегрузка унарных операторов

Унарные операторы перегружаются таким же образом, как и бинарные. Главное отличие заключается, конечно, в том, что у них имеется лишь один операнд. Давайте модернизируем предыдущий пример, дополнив перегрузки операций ++, --, -:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1
{
```

```
class MyArr
{
    // Координаты точки в трехмерном пространстве
    public int x, y, z;
    public MyArr(int x = 0, int y = 0, int z = 0)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    // Перегружаем бинарный оператор +
    public static MyArr operator +(MyArr obj1, MyArr obj2)
    {
        MyArr arr = new MyArr();
        arr.x = obj1.x + obj2.x;
        arr.y = obj1.y + obj2.y;
        arr.z = obj1.z + obj2.z;
        return arr;
    }
    // Перегружаем бинарный оператор -
    public static MyArr operator -(MyArr obj1, MyArr obj2)
    {
        MyArr arr = new MyArr();
        arr.x = obj1.x - obj2.x;
        arr.y = obj1.y - obj2.y;
        arr.z = obj1.z - obj2.z;
        return arr;
    }
    // Перегружаем унарный оператор -
    public static MyArr operator -(MyArr obj1)
    {
        MyArr arr = new MyArr();
        arr.x = -obj1.x;
        arr.y = -obj1.y;
        arr.z = -obj1.z;
        return arr;
    }
    // Перегружаем унарный оператор ++
    public static MyArr operator ++(MyArr obj1)
    {
        obj1.x += 1;
        obj1.y += 1;
        obj1.z +=1;
        return obj1;
    }
    // Перегружаем унарный оператор --
    public static MyArr operator --(MyArr obj1)
    {
```

```
            obj1.x -= 1;
            obj1.y -= 1;
            obj1.z -= 1;
            return obj1;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyArr Point1 = new MyArr(1, 12, -4);
            MyArr Point2 = new MyArr(0, -3, 18);
            Console.WriteLine("Координаты первой точки: " +
                Point1.x + " " + Point1.y + " " + Point1.z);
            Console.WriteLine("Координаты второй точки: " +
                Point2.x + " " + Point2.y + " " + Point2.z + "\n");

            MyArr Point3 = Point1 + Point2;
            Console.WriteLine("\nPoint1 + Point2 = "
                + Point3.x + " " + Point3.y + " " + Point3.z);
            Point3 = Point1 - Point2;
            Console.WriteLine("Point1 - Point2 = "
                + Point3.x + " " + Point3.y + " " + Point3.z);
            Point3 = -Point1;
            Console.WriteLine("-Point1 = "
                + Point3.x + " " + Point3.y + " " + Point3.z);
            Point2++;
            Console.WriteLine("Point2++ = "
                + Point2.x + " " + Point2.y + " " + Point2.z);
            Point2--;
            Console.WriteLine("Point2-- = "
                + Point2.x + " " + Point2.y + " " + Point2.z);

            Console.ReadLine();
        }
    }
}
```

22.Индексаторы. Свойства. Неизменяемые поля. Константы.

Индексаторы

Индексаторы позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

Формальное определение индексатора:

```
возвращаемый_тип this [Тип параметр1, ...]
{
    get { ... }
    set { ... }
}
```

В отличие от свойств индексатор не имеет названия. Вместо него указывается ключевое слово this, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр.

Посмотрим на примере. Допустим, у нас есть класс Person, который представляет человека, и класс Company, который представляет некоторую компанию, где работают люди. Используем индексаторы для определения класса Company:

```
class Person
{
    public string Name { get;}
    public Person(string name) => Name=name;
}
class Company
{
    Person[] personal;
    public Company(Person[] people) => personal = people;
    // индексатор
    public Person this[int index]
    {
        get => personal[index];
        set => personal[index] = value;
    }
}
```

Для хранения персонала компании в классе определен массив personal, который состоит из объектов Person. Для доступа к этим объектам определен индексатор:

```
public Person this[int index]
```

Индексатор в принципе подобен стандартному свойству. Во-первых, для индексатора определяется тип в данном случае тип Person. Тип индексатора определяет, какие объекты будет получать и возвращать индексатор.

Во-вторых, для индексатора определен параметр int index, через который обращаемся к элементам внутри объекта Company.

Для возвращения объекта в индексаторе определен блок get:

```
get => personal[index];
```

Поскольку индексатор имеет тип Person, то в блоке get нам надо возвратить объект этого типа с помощью оператора return. Здесь мы можем определить разнообразную логику. В данном случае просто возвращаем объект из массива personal.

В блоке set, как и в обычном свойстве, получаем через параметр value переданный объект Person и сохраняем его в массив по индексу.

```
set => personal[index] = value;
```

После этого мы можем работать с объектом Company как с набором объектов Person:

```
var microsoft = new Company(new[]
{
    new Person("Tom"), new Person("Bob"), new Person("Sam"), new Person("Alice")
});
// получаем объект из индексатора
Person firstPerson = microsoft[0];
Console.WriteLine(firstPerson.Name);  // Tom
// переустанавливаем объект
microsoft[0] = new Person("Mike");
Console.WriteLine(microsoft[0].Name);  // Mike
```

Стоит отметить, что если индексатору будет передан некорректный индекс, который отсутствует в массиве person, то мы получим исключение, как и в случае обращения напрямую к элементам массива. В этом случае можно предусмотреть какую-то дополнительную логику. Например, проверять переданный индекс:

Индексы

Индексатор получает набор индексов в виде параметров. Однако индексы необязательно должны представлять тип int, устанавливаемые/возвращаемые значения необязательно хранить в массиве. Например, мы можем рассматривать объект как хранилище атрибутов/свойств и передавать имя атрибута в виде строки:

```
User tom = new User();
// устанавливаем значения
tom["name"] = "Tom";
tom["email"] = "tom@gmail.ru";
tom["phone"] = "+1234556767";
```

Применение нескольких параметров

Также индексатор может принимать несколько параметров. Допустим, у нас есть класс, в котором хранилище определено в виде двухмерного массива или матрицы:

```
class Matrix
{
    int[,] numbers = new int[,] { { 1, 2, 4 }, { 2, 3, 6 }, { 3, 4, 8 } };
    public int this[int i, int j]
    {
        get => numbers[i, j];
        set => numbers[i, j] = value;
    }
}
```

Теперь для определения индексатора используются два индекса - i и j.

Следует учитывать, что индексатор не может быть статическим и применяется только к экземпляру класса. Но при этом индексаторы могут быть виртуальными и абстрактными и могут переопределяться в производных классах.

Блоки get и set

Как и в свойствах, в индексаторах можно опускать блок get или set, если в них нет необходимости. Например, удалим блок set и сделаем индексатор доступным только для чтения.

Перегрузка индексаторов

Подобно методам индексаторы можно перегружать. В этом случае также индексаторы должны отличаться по количеству, типу или порядку используемых параметров.

Свойства

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.

Определение свойств

Стандартное описание свойства имеет следующий синтаксис:

```
 [модификаторы] тип_свойства название_свойства
{
   get { действия, выполняемые при получении значения свойства}
   set { действия, выполняемые при установке значения свойства}
}
```

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.

Рассмотрим пример:

```
Person person = new Person();
// Устанавливаем свойство - срабатывает блок Set
// значение "Tom" и есть передаваемое в свойство value
person.Name = "Tom";
// Получаем значение свойства и присваиваем его переменной - срабатывает блок Get
string personName = person.Name;
Console.WriteLine(personName);  // Tom
class Person
{
   private string name = "Undefined";
   public string Name
   {
      get
      {
         return name;    // возвращаем значение свойства
      }
      set
      {
         name = value;   // устанавливаем новое значение свойства
      }
   }
}
```

В программе мы можем обращаться к свойству, как к обычному полю. Если мы ему присваиваем какое-нибудь значение, то срабатывает блок set, а передаваемое значение

передается в параметр value. Если мы получаем значение свойства, то срабатывает блок get, который по сути возвращает значение переменной.
Возможно, может возникнуть вопрос, зачем нужны свойства, если мы можем в данной ситуации обходиться обычными полями класса? Но свойства позволяют вложить дополнительную логику, которая может быть необходима при установке или получении значения. Свойство позволяет опосредовать и контролировать доступ к данным объекта.

Свойства только для чтения и записи

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.

Вычисляемые свойства

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

```
Person tom = new("Tom", "Smith");
Console.WriteLine(tom.Name);    // Tom Smith
class Person
{
   string firstName;
   string lastName;
   public string Name
   {
      get { return  $"{firstName} {lastName}"; }
   }
   public Person(string firstName, string lastName)
   {
      this.firstName = firstName;
      this.lastName = lastName;
   }
}
```

Модификаторы доступа

Мы можем применять модификаторы доступа не только ко всему свойству, но и к отдельным блокам get и set.

При использовании модификаторов в свойствах следует учитывать ряд ограничений:

- Модификатор для блока set или get можно установить, если свойство имеет оба блока (и set, и get)
- Только один блок set или get может иметь модификатор доступа, но не оба сразу
- Модификатор доступа блока set или get должен быть более ограничивающим, чем модификатор доступа свойства. Например, если свойство имеет модификатор public, то блок set/get может иметь только модификаторы protected internal, internal, protected, private protected и private

Автоматические свойства

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:

```
class Person
```

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

   public Person(string name, int age)
   {
      Name = name;
      Age = age;
   }
}
```
На самом деле тут также создаются поля для свойств, только их создает не программист в коде, а компилятор автоматически генерирует при компиляции.
В чем преимущество автосвойств, если по сути они просто обращаются к автоматически создаваемой переменной, почему бы напрямую не обратиться к переменной без автосвойств? Дело в том, что в любой момент времени при необходимости мы можем развернуть автосвойство в обычное свойство, добавить в него какую-то определенную логику.
Стоит учитывать, что нельзя создать автоматическое свойство только для записи, как в случае со стандартными свойствами.

- Автосвойствам можно присвоить значения по умолчанию (инициализация автосвойств):

```
Person tom = new();
Console.WriteLine(tom.Name);    // Tom
Console.WriteLine(tom.Age);    // 37
class Person
{
   public string Name { get; set; } = "Tom";
   public int Age { get; set; } = 37;
}
```
И если мы не укажем для объекта Person значения свойств Name и Age, то будут действовать значения по умолчанию.

- Автосвойства также могут иметь модификаторы доступа:

```
class Person
{
   public string Name { private set; get;}
   public Person(string name) => Name = name;
}
```
Мы можем убрать блок set и сделать автосвойство доступным только для чтения. В этом случае для хранения значения этого свойства для него неявно будет создаваться поле с модификатором readonly, поэтому следует учитывать, что подобные get-свойства можно установить либо из конструктора класса, как в примере выше, либо при инициализации свойства:

```
class Person
{
   // через инициализацию свойства
   public string Name { get; } = "Tom";
   // через конструктор
   public Person(string name) => Name = name;
```

```
}
```
Блок init

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). Например:

```
Person person = new();
//person.Name = "Bob";    //! Ошибка - после инициализации изменить значение нельзя
Console.WriteLine(person.Name); // Undefined
public class Person
{
    public string Name { get; init; } = "Undefined";
}
```

В данном случае класс Person для свойства Name вместо сеттера использует оператор init. В итоге на строке

```
Person person = new();
```

предполагается создание объекта с инициализацией всех его свойств. В данном случае свойство Name получит в качестве значения строку "Undefined". Однако поскольку инициализация свойства уже произошла, то на строке

```
person.Name = "Bob";    // Ошибка
```

мы получим ошибку.

Как можно установить подобное свойство? Выше продемонстрирован один из способов - установка значения при определении свойства. Второй способ - через конструктор:

```
Person person = new("Tom");
Console.WriteLine(person.Name); // Tom
public class Person
{
    public Person(string name) => Name = name;
    public string Name { get; init; }
}
```

Третий способ - через инициализатор:

```
Person person = new() { Name = "Bob"};
Console.WriteLine(person.Name); // Bob
public class Person
{
    public string Name { get; init; } = "";
}
```

В принципе есть еще четвертый способ - установка через другое свойство с модификатором init:

```
var person = new Person() { Name = "Sam" };
Console.WriteLine(person.Name);     // Sam
Console.WriteLine(person.Email);    // Sam@gmail.com
public class Person
{
```

```
    string name = "";
    public string Name
    {
        get { return name; }
        init
        {
            name = value;
            Email = $"{value}@gmail.com";
        }
    }
    public string Email { get; init; } = "";
}
```

В данном случае в init-свойстве Name разворачивается в полное свойство, которое управляет полем для чтения name. Благодаря этому перед установкой значения свойства мы можем произвести некоторую предобработку. Кроме того, в выражении init устанавливается другое init-свойство - Email, которое для установки значения использует значение свойства Name - из имени получаем значение для электронного адреса.

Причем если при объявлении свойства указано значение, то в конструкторе мы можем его изменить. Значение, установленное в конструкторе, можно изменить в инициализаторе. Однако дальше процесс инициализации заканчивается. И значение не может быть изменено.

Сокращенная запись свойств

Как и методы, мы можем сокращать определения свойств. Поскольку блоки get и set представляют специальные методы, то как и обычные методы, если они содержат одну инструкцию, то мы их можем сократить с помощью оператора =>:

```
class Person
{
    string name;
    public string Name
    {
        get => name;
        set => name = value;
    }
}
```

Также можно сокращать все свойство в целом:

```
class Person
{
    string name;

    // эквивалентно public string Name { get { return name; } }
    public string Name => name;
}
```

Модификатор required

Модификатор required (добавлен в C# 11) указывает, что поле или свойства с этим модификатором обязательно должны быть инициализированы.

Причем не важно, устанавливаем эти свойства в конструкторе или инициализируем при определении, все равно надо использовать инициализатор для установки их значений.

Неизменяемые поля.Константы

Константы классы

Кроме полей класс может определять для хранения данных константы. В отличие от полей их значение устанавливается один раз непосредственно при их объявлении и впоследствии не может быть изменено. Кроме того, константы хранят некоторые данные, которые относятся не к одному объекту, а ко всему классу в целом. И для обращения к константам применяется не имя объекта, а имя класса:

```
Person tom = new Person();
tom.name = "Tom";
tom.Print();    // Person: Tom
Console.WriteLine(Person.type); // Person
// Person.type = "User";    // !Ошибка: изменить константу нельзя
class Person
{
    public const string type = "Person";
    public string name = "Undefined";
    public void Print() => Console.WriteLine($"{type}: {name}");
}
```

Стоит отметить, что константе сразу при ее определении необходимо присвоить значение.

Подобно обычным полям мы можем обращаться к константам класса внутри этого класса. Однако если мы хотим обратиться к константе вне ее класса, то для обращения необходимо использовались имя класса:

```
Console.WriteLine(Person.type); // Person
```

Таким образом, если необходимо хранить данные, которые относятся ко всему классу в целом, то можно использовать константы.

Поля для чтения и модификатор readonly

Поля для чтения представляют такие поля класса или структуры, значение которых нельзя изменить. Таким полям можно присвоить значение либо непосредственно при их объявлении, либо в конструкторе. В других местах программы присваивать значение таким полям нельзя, можно только считывать их значение.

Поле для чтения объявляется с ключевым словом readonly:

```
Person tom = new Person("Tom");
Console.WriteLine(tom.name);
//tom.name = "Sam"; // !Ошибка: нельзя изменить
class Person
{
    public readonly string name = "Undefined"; // можно так инициализировать
    public Person(string name)
    {
        this.name = name; // в конструкторе также можно присвоить значение полю для чтения
    }
    public void ChangeName(string otherName)
    {
        //this.name = otherName;    // так нельзя
    }
}
```

Сравнение констант

Константы должны быть определены во время компиляции, а поля для чтения могут быть определены во время выполнения программы.
Соответственно значение константы можно установить только при ее определении.
Поле для чтения можно инициализировать либо при его определении, либо в конструкторе класса.
Константы не могут использовать модификатор static, так как уже неявно являются статическими. Поля для чтения могут быть как статическими, так и не статическими.
Структуры для чтения
Кроме полей для чтения в C# можно определять структуры для чтения. Для этого они предваряются модификатором readonly.
Особенностью таких структур является то, что все их поля должны быть также полями для чтения:

```
readonly struct Person
{
    public readonly string name;
    public Person(string name)
    {
        this.name = name;
    }
}
```

То же самое касается и свойств, которые должны быть доступны только для чтения:

```
readonly struct Person
{
    public readonly string Name { get; } // указывать readonly необязательно
    public int Age { get; } // свойство только для чтения
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

Константа - это переменная, значение которой нельзя изменить.
Статическая - это переменная, которая не может использоваться вне области ее объявления. То есть, если это глобальная переменная, то она может использоваться только в файле, который ее объявляет. Если это переменная внутри функции, то ее можно использовать только внутри этой функции.

23.Перечисления. Флаги.

Перечисления

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:

```
enum название_перечисления
{
    // значения перечисления
    значение1,
    значение2,
    .....
    значениеN
}
```

После оператора enum идет название перечисления. И затем в фигурных скобках через запятую перечисляются константы перечисления.

Определим простейшее перечисление:

```
enum DayTime
{
    Morning,
    Afternoon,
    Evening,
    Night
}
```

Здесь определено перечисление DayTime, которое имеет четыре значения: Morning, Afternoon, Evening и Night

Каждое перечисление фактически определяет новый тип данных, с помощью которых мы также, как и с помощью любого другого типа, можем определять переменные, константы, параметры методов и т.д. В качестве значения переменной, константы и параметра метода, которые представляют перечисление, должна выступать одна из констант этого перечисления, например:

```
const DayTime dayTime = DayTime.Morning;
```

Далее в программе мы можем использовать подобные переменные/константы/параметры как и любые другие:

```
DayTime dayTime = DayTime.Morning;
if(dayTime == DayTime.Morning)
    Console.WriteLine("Доброе утро");
else
    Console.WriteLine("Привет");
enum DayTime
{
    Morning,
    Afternoon,
    Evening,
    Night
}
```

Хранение состояния

Зачастую переменная перечисления выступает в качестве хранилища состояния, в зависимости от которого производятся некоторые действия:

```
DayTime now = DayTime.Evening;
PrintMessage(now);                  // Добрый вечер
PrintMessage(DayTime.Afternoon);    // Добрый день
PrintMessage(DayTime.Night);        // Доброй ночи
void PrintMessage(DayTime dayTime)
{
    switch (dayTime)
    {
        case DayTime.Morning:
            Console.WriteLine("Доброе утро");
            break;
        case DayTime.Afternoon:
            Console.WriteLine("Добрый день");
            break;
        case DayTime.Evening:
            Console.WriteLine("Добрый вечер");
            break;
        case DayTime.Night:
            Console.WriteLine("Доброй ночи");
            break;
    }
}
enum DayTime
{
    Morning,
    Afternoon,
    Evening,
    Night
}
```

Здесь метод PrintMessage() в качестве параметра принимает значение типа перечисления DayTime и зависимости от этого значения выводит определенное приведение.

Тип и значения констант перечисления

Константы перечисления могут иметь тип. Тип указывается после названия перечисления через двоеточие:

```
enum Time : byte
{
    Morning,
    Afternoon,
    Evening,
    Night
}
```

Тип перечисления обязательно должен представлять целочисленный тип (byte, sbyte, short, ushort, int, uint, long, ulong). Если тип явным образом не указан, то по умолчанию используется тип int.

Тип влияет на значения, которые могут иметь константы. По умолчанию каждому элементу перечисления присваивается целочисленное значение, причем первый элемент будет иметь значение 0, второй - 1 и так далее. Например, возьмем выше определенное DayTime:

```
DayTime now = DayTime.Morning;
Console.WriteLine((int) now);  // 0
Console.WriteLine((int) DayTime.Night);  // 3
enum DayTime
{
    Morning,
    Afternoon,
    Evening,
    Night
}
```

Мы можем использовать операцию приведения, чтобы получить целочисленное значение константы перечисления:

```
 (int) DayTime.Night // 3
```

В то же время, несмотря на то, что каждая константа сопоставляется с определенным числом, мы НЕ можем присвоить ей числовое значение:

```
DayTime now = 2;    // ! Ошибка
```

Можно также явным образом указать значения элементов, либо указав значение первого элемента:

```
enum DayTime
{
    Morning = 3,    // каждый следующий элемент по умолчанию увеличивается на единицу
    Afternoon,      // этот элемент равен 4
    Evening,        // 5
    Night           // 6
}
```

Но можно и для всех элементов явным образом указать значения:

```
enum DayTime
{
    Morning = 2,
    Afternoon = 4,
    Evening = 8,
    Night = 16
}
```

При этом константы перечисления могут иметь одинаковые значения, либо даже можно присваивать одной константе значение другой константы:

```
enum DayTime
{
    Morning = 1,
    Afternoon = Morning,
    Evening = 2,
    Night = 2
}
```

Флаги

Битовые флаги

Предположим мы имеем некий объект, обладающий рядом булевых свойств. Мы, несомненно, можем описать их в виде нескольких самостоятельных полей типа bool. Данный подход является вполне приемлемым:

```
public class ClassWithBools
{
   public ConditionsBools Conditions { get; set; }
   public class ConditionsBools
   {
      public bool HasSomething { get; set; }
      public bool DoSomething { get; set; }
      public bool HasAnother { get; set; }
      public bool DoAnother { get; set; }
   }
}
```

Однако, есть иной способ определить эти атрибуты. Для этого мы можем использовать такую надстройку над enum-ом, как битовые флаги. Подобная конструкция представляет из себя обыкновенное enum-ом, обладающее атрибутом [Flags] и числовыми значениями, представляющими из себя степени двойки:

```
public class ClassWithFlags
{
   public ConditionsFlags Conditions { get; set; }
   [Flags]
   public enum ConditionsFlags
   {
      HasSomething = 1 << 0,  //0001 = 1
      DoSomething = 1 << 1,   //0010 = 2
      HasAnother = 1 << 2,    //0100 = 4
      DoAnother = 1 << 3      //1000 = 8
   }
}
```

Как же работать с бинарным флагом? В случае использования отдельных булевых переменных всё достаточно очевидно: обращайся к конкретной переменной и читай или же присваивай значение. Но в случае же работы битовыми флагами приходится использовать базовые двоичные операции: конъюнкция (И, &) и дизъюнкция (ИЛИ, |). Операция ИЛИ позволяет вернуть множество, содержащее в себе все подмножества, используемые в данной операции. Операция И позволяет вернуть подмножество пересечения множеств, используемые в данной операции.

Используя эти операции мы можем работать с флагами:

```
var classWithFlags = new ClassWithFlags();
classWithFlags.Conditions |= ConditionsFlags.DoAnother;                    //Присвоение значения
var hasClassDoAnother = (classWithFlags.Conditions & ConditionsFlags.DoAnother) != 0;
//Чтение значения
hasClassDoAnother = classWithFlags.Conditions.HasFlag(ConditionsFlags.DoAnother);
//Более привычное чтение
```

Как можно увидеть, работа с битовыми флагами является несколько нестандартной. Помимо этого неудобства, битовый флаг подразумевает, что состояние всегда определено и равно либо true, либо false, и не может быть равно null. Данная особенность

может являться недостатком, поскольку порой постановка задачи может подразумевать, что состояние может быть неизвестно.
Битовый флаг является достаточно специфичным типом данных. Его использование вместо структуры, обладающей булевыми переменными, зачастую является спорным, но иногда вполне оправданным. Одним из неочевидных преимуществ использования флагов является следующий момент: в случае использования битовых флагов нет необходимости обновлять структуру базы данных каждый раз, как появляется новое свойство, поскольку в базе данных подобные флаги зачастую хранятся в виде чисел.

24.Делегаты. События. Списки вызовов

Делегаты
Делегаты представляют такие объекты, которые указывают на методы. То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.
Определение делегатов
Для объявления делегата используется ключевое слово delegate, после которого идет возвращаемый тип, название и параметры. Например:
```
delegate void Message();
```
Рассмотрим применение этого делегата:
```
void Hello() => Console.WriteLine("Hello METANIT.COM");
```
Прежде всего сначала необходимо определить сам делегат:
```
delegate void Message(); // 1. Объявляем делегат
```
Для использования делегата объявляется переменная этого делегата:
```
Message mes; // 2. Создаем переменную делегата
```
Далее в делегат передается адрес определенного метода (в нашем случае метода Hello). Обратите внимание, что данный метод имеет тот же возвращаемый тип и тот же набор параметров (в данном случае отсутствие параметров), что и делегат.
```
mes = Hello; // 3. Присваиваем этой переменной адрес метода
```
Затем через делегат вызываем метод, на который ссылается данный делегат:
```
mes(); // 4. Вызываем метод
```
Вызов делегата производится подобно вызову метода.
При этом делегаты необязательно могут указывать только на методы, которые определены в том же классе, где определена переменная делегата. Это могут быть также методы из других классов и структур.
```
Message message1 = Welcome.Print;
Message message2 = new Hello().Display;
message1(); // Welcome
message2(); // Привет
delegate void Message();
class Welcome
{
    public static void Print() => Console.WriteLine("Welcome");
}
class Hello
{
    public void Display() => Console.WriteLine("Привет");
}
```
Место определения делегата

Если мы определяем делегат в прогаммах верхнего уровня (top-level program), которую по умолчанию представляет файл Program.cs начиная с версии C# 10, как в примере выше, то, как и другие типы, делегат определяется в конце кода. Но в принцие делегат можно определять внутри класса:

```
class Program
{
    delegate void Message(); // 1. Объявляем делегат
    static void Main()
    {
        Message mes;           // 2. Создаем переменную делегата
        mes = Hello;           // 3. Присваиваем этой переменной адрес метода
        mes();                 // 4. Вызываем метод

        void Hello() => Console.WriteLine("Hello METANIT.COM");
    }
}
```

Либо вне класса:

```
delegate void Message(); // 1. Объявляем делегат
class Program
{
    static void Main()
    {
        Message mes;           // 2. Создаем переменную делегата
        mes = Hello;           // 3. Присваиваем этой переменной адрес метода
        mes();                 // 4. Вызываем метод

        void Hello() => Console.WriteLine("Hello METANIT.COM");
    }
}
```

Параметры и результат делегата

Рассмотрим определение и применение делегата, который принимает параметры и возвращает результат:

```
Operation operation = Add;      // делегат указывает на метод Add
int result = operation(4, 5);   // фактически Add(4, 5)
Console.WriteLine(result);      // 9
operation = Multiply;           // теперь делегат указывает на метод Multiply
result = operation(4, 5);       // фактически Multiply(4, 5)
Console.WriteLine(result);      // 20
int Add(int x, int y) => x + y;
int Multiply(int x, int y) => x * y;
delegate int Operation(int x, int y);
```

В данном случае делегат Operation возвращает значение типа int и имеет два параметра типа int. Поэтому этому делегату соответствует любой метод, который возвращает значение типа int и принимает два параметра типа int. В данном случае это методы Add и Multiply. То есть мы можем присвоить переменной делегата любой из этих методов и вызывать.

Поскольку делегат принимает два параметра типа int, то при его вызове необходимо передать значения для этих параметров: operation(4,5).

Присвоение ссылки на метод

Выше переменной делегата напрямую присваивался метод. Есть еще один способ - создание объекта делегата с помощью конструктора, в который передается нужный метод:

```
Operation operation1 = Add;
Operation operation2 = new Operation(Add);
int Add(int x, int y) => x + y;
delegate int Operation(int x, int y);
```

Оба способа равноценны.

Добавление методов в делегат

В примерах выше переменная делегата указывала на один метод. В реальности же делегат может указывать на множество методов, которые имеют ту же сигнатуру и возвращаемые тип. Все методы в делегате попадают в специальный список - список вызова или invocation list. И при вызове делегата все методы из этого списка последовательно вызываются. И мы можем добавлять в этот список не один, а несколько методов. Для добавления методов в делегат применяется операция +=:

```
Message message = Hello;
message += HowAreYou;  // теперь message указывает на два метода
message();             // вызываются оба метода - Hello и HowAreYou
void Hello() => Console.WriteLine("Hello");
void HowAreYou() => Console.WriteLine("How are you?");
delegate void Message();
```

В данном случае в список вызова делегата message добавляются два метода - Hello и HowAreYou. И при вызове message вызываются сразу оба этих метода.

Однако стоит отметить, что в реальности будет происходить создание нового объекта делегата, который получит методы старой копии делегата и новый метод, и новый созданный объект делегата будет присвоен переменной message.

При добавлении делегатов следует учитывать, что мы можем добавить ссылку на один и тот же метод несколько раз, и в списке вызова делегата тогда будет несколько ссылок на один и то же метод. Соответственно при вызове делегата добавленный метод будет вызываться столько раз, сколько он был добавлен.

Подобным образом мы можем удалять методы из делегата с помощью операций -=.

При удалении методов из делегата фактически будет создаваться новый делегат, который в списке вызова методов будет содержать на один метод меньше.

Стоит отметить, что при удалении метода может сложиться ситуация, что в делегате не будет методов, и тогда переменная будет иметь значение null. Поэтому в данном случае переменная определена не просто как переменная типа Message, а именно Message?, то есть типа, который может представлять как делегат Message, так и значение null.

При удалении следует учитывать, что если делегат содержит несколько ссылок на один и тот же метод, то операция -= начинает поиск с конца списка вызова делегата и удаляет только первое найденное вхождение. Если подобного метода в списке вызова делегата нет, то операция -= не имеет никакого эффекта.

Объединение делегатов

Делегаты можно объединять в другие делегаты. Например:

```
Message mes1 = Hello;
Message mes2 = HowAreYou;
Message mes3 = mes1 + mes2; // объединяем делегаты
mes3(); // вызываются все методы из mes1 и mes2
```

```
void Hello() => Console.WriteLine("Hello");
void HowAreYou() => Console.WriteLine("How are you?");
delegate void Message();
```

В данном случае объект mes3 представляет объединение делегатов mes1 и mes2. Объединение делегатов значит, что в список вызова делегата mes3 попадут все методы из делегатов mes1 и mes2. И при вызове делегата mes3 все эти методы одновременно будут вызваны.

Вызов делегата

В примерах выше делегат вызывался как обычный метод. Если делегат принимал параметры, то при его вызове для параметров передавались необходимые значения:

```
Message mes = Hello;
mes();
Operation op = Add;
int n = op(3, 4);
Console.WriteLine(n);
void Hello() => Console.WriteLine("Hello");
int Add(int x, int y) => x + y;
delegate int Operation(int x, int y);
delegate void Message();
```

Другой способ вызова делегата представляет метод Invoke():

```
Message mes = Hello;
mes.Invoke(); // Hello
Operation op = Add;
int n = op.Invoke(3, 4);
Console.WriteLine(n);   // 7
void Hello() => Console.WriteLine("Hello");
int Add(int x, int y) => x + y;
delegate int Operation(int x, int y);
delegate void Message();
```

Если делегат принимает параметры, то в метод Invoke передаются значения для этих параметров.

Следует учитывать, что если делегат пуст, то есть в его списке вызова нет ссылок ни на один из методов (то есть делегат равен Null), то при вызове такого делегата мы получим исключение, как, например, в следующем случае:

Поэтому при вызове делегата всегда лучше проверять, не равен ли он null. Либо можно использовать метод Invoke и оператор условного null:

```
Message? mes = null;
mes?.Invoke();        // ошибки нет, делегат просто не вызывается
Operation? op = Add;
op -= Add;          // делегат op пуст
int? n = op?.Invoke(3, 4);   // ошибки нет, делегат просто не вызывается, а n = null
```

Обобщенные делегаты

Делегаты, как и другие типы, могут быть обобщенными, например:

```
Operation<decimal, int> squareOperation = Square;
decimal result1 = squareOperation(5);
Console.WriteLine(result1);  // 25
Operation<int, int> doubleOperation = Double;
int result2 = doubleOperation(5);
```

```
Console.WriteLine(result2);  // 10
decimal Square(int n) => n * n;
int Double(int n) => n + n;
delegate T Operation<T, K>(K val);
```

Здесь делегат Operation типизируется двумя параметрами типов. Параметр T представляет тип возвращаемого значения. А параметр K представляет тип передаваемого в делегат параметра. Таким образом, этому делегату соответствует метод, который принимает параметр любого типа и возвращает значение любого типа.

Делегаты как параметры методов

Также делегаты могут быть параметрами методов. Благодаря этому один метод в качестве параметров может получать действия - другие методы. Например:

```
DoOperation(5, 4, Add);         // 9
DoOperation(5, 4, Subtract);    // 1
DoOperation(5, 4, Multiply);    // 20

void DoOperation(int a, int b, Operation op)
{
   Console.WriteLine(op(a,b));
}
int Add(int x, int y) => x + y;
int Subtract(int x, int y) => x - y;
int Multiply(int x, int y) => x * y;

delegate int Operation(int x, int y);
```

Возвращение делегатов из метода

Также делегаты можно возвращать из методов. То есть мы можем возвращать из метода какое-то действие в виде другого метода. Например:

```
Operation operation = SelectOperation(OperationType.Add);
Console.WriteLine(operation(10, 4));    // 14

operation = SelectOperation(OperationType.Subtract);
Console.WriteLine(operation(10, 4));    // 6

operation = SelectOperation(OperationType.Multiply);
Console.WriteLine(operation(10, 4));    // 40

Operation SelectOperation(OperationType opType)
{
   switch (opType)
   {
      case OperationType.Add: return Add;
      case OperationType.Subtract: return Subtract;
      default: return Multiply;
   }
}
int Add(int x, int y) => x + y;
int Subtract(int x, int y) => x - y;
int Multiply(int x, int y) => x * y;
```

```
enum OperationType
{
    Add, Subtract, Multiply
}
delegate int Operation(int x, int y);
```

События

События сигнализируют системе о том, что произошло определенное действие. И если нам надо отследить эти действия, то как раз мы можем применять события.

Например, возьмем следующий класс, который описывает банковский счет:

```
class Account
{
    // сумма на счете
    public int Sum { get; private set; }
    // в конструкторе устанавливаем начальную сумму на счете
    public Account(int sum) => Sum = sum;
    // добавление средств на счет
    public void Put(int sum) => Sum += sum;
    // списание средств со счета
    public void Take(int sum)
    {
        if (Sum >= sum)
        {
            Sum -= sum;
        }
    }
}
```

В конструкторе устанавливаем начальную сумму, которая хранится в свойстве Sum. С помощью метода Put мы можем добавить средства на счет, а с помощью метода Take, наоборот, снять деньги со счета. Попробуем использовать класс в программе - создать счет, положить и снять с него деньги:

```
Account account = new Account(100);
account.Put(20);    // добавляем на счет 20
Console.WriteLine($"Сумма на счете: {account.Sum}");
account.Take(70);   // пытаемся снять со счета 70
Console.WriteLine($"Сумма на счете: {account.Sum}");
account.Take(180);  // пытаемся снять со счета 180
Console.WriteLine($"Сумма на счете: {account.Sum}");
```

Все операции работают как и положено. Но что если мы хотим уведомлять пользователя о результатах его операций. Мы могли бы, например, для этого изменить метод Put следующим образом:

```
public void Put(int sum)
{
    Sum += sum;
    Console.WriteLine($"На счет поступило: {sum}");
}
```

Казалось, теперь мы будем извещены об операции, увидев соответствующее сообщение на консоли. Но тут есть ряд замечаний. На момент определения класса мы можем точно не знать, какое действие мы хотим произвести в методе Put в ответ на добавление денег.

Это может вывод на консоль, а может быть мы захотим уведомить пользователя по email или sms. Более того мы можем создать отдельную библиотеку классов, которая будет содержать этот класс, и добавлять ее в другие проекты. И уже из этих проектов решать, какое действие должно выполняться. Возможно, мы захотим использовать класс Account в графическом приложении и выводить при добавлении на счет в графическом сообщении, а не консоль. Или нашу библиотеку классов будет использовать другой разработчик, у которого свое мнение, что именно делать при добавлении на счет. И все эти вопросы мы можем решить, используя события.

Определение и вызов событий

События объявляются в классе с помощью ключевого слова event, после которого указывается тип делегата, который представляет событие:

```
delegate void AccountHandler(string message);
event AccountHandler  Notify;
```

В данном случае вначале определяется делегат AccountHandler, который принимает один параметр типа string. Затем с помощью ключевого слова event определяется событие с именем Notify, которое представляет делегат AccountHandler. Название для события может быть произвольным, но в любом случае оно должно представлять некоторый делегат.

Определив событие, мы можем его вызвать в программе как метод, используя имя события:

```
Notify("Произошло действие");
```

Поскольку событие Notify представляет делегат AccountHandler, который принимает один параметр типа string - строку, то при вызове события нам надо передать в него строку.

Однако при вызове событий мы можем столкнуться с тем, что событие равно null в случае, если для его не определен обработчик. Поэтому при вызове события лучше его всегда проверять на null. Например, так:

```
if(Notify !=null) Notify("Произошло действие");
```

Или так:

```
Notify?.Invoke("Произошло действие");
```

В этом случае поскольку событие представляет делегат, то мы можем его вызвать с помощью метода Invoke(), передав в него необходимые значения для параметров.

Объединим все вместе и создадим и вызовем событие:

```
class Account
{
    public delegate void AccountHandler(string message);
    public event AccountHandler? Notify;              // 1.Определение события
    public Account(int sum) => Sum = sum;
    public int Sum { get; private set; }
    public void Put(int sum)
    {
        Sum += sum;
        Notify?.Invoke($"На счет поступило: {sum}");   // 2.Вызов события
    }
    public void Take(int sum)
    {
        if (Sum >= sum)
        {
```

```
            Sum -= sum;
            Notify?.Invoke($"Со счета снято: {sum}");   // 2.Вызов события
        }
        else
        {
            Notify?.Invoke($"Недостаточно денег на счете. Текущий баланс: {Sum}"); ;
        }
    }
}
```

Добавление обработчика события

С событием может быть связан один или несколько обработчиков. Обработчики событий - это именно то, что выполняется при вызове событий. Нередко в качестве обработчиков событий применяются методы. Каждый обработчик событий по списку параметров и возвращаемому типу должен соответствовать делегату, который представляет событие.

Для добавления обработчика события применяется операция +=:

```
Notify += обработчик события;
```

Определим обработчики для события Notify, чтобы получить в программе нужные уведомления:

```
Account account = new Account(100);
account.Notify += DisplayMessage;   // Добавляем обработчик для события Notify
account.Put(20);    // добавляем на счет 20
Console.WriteLine($"Сумма на счете: {account.Sum}");
account.Take(70);   // пытаемся снять со счета 70
Console.WriteLine($"Сумма на счете: {account.Sum}");
account.Take(180);  // пытаемся снять со счета 180
Console.WriteLine($"Сумма на счете: {account.Sum}");
void DisplayMessage(string message) => Console.WriteLine(message);
```

В данном случае в качестве обработчика используется метод DisplayMessage, который соответствует по списку параметров и возвращаемому типу делегату AccountHandler. В итоге при вызове события Notify?.Invoke() будет вызываться метод DisplayMessage, которому для параметра message будет передаваться строка, которая передается в Notify?.Invoke(). В DisplayMessage просто выводим полученное от события сообщение, но можно было бы определить любую логику.

Если бы в данном случае обработчик не был бы установлен, то при вызове события Notify?.Invoke() ничего не происходило, так как событие Notify было бы равно null.

Консольный вывод программы:

```
На счет поступило: 20
Сумма на счете: 120
Со счета снято: 70
Сумма на счете: 50
Недостаточно денег на счете. Текущий баланс: 50
Сумма на счете: 50
```

В качестве обработчиков могут использоваться не только обычные методы, но также делегаты, анонимные методы и лямбда-выражения. Использование делегатов и методов:

```
Account acc = new Account(100);
// установка делегата, который указывает на метод DisplayMessage
acc.Notify += new Account.AccountHandler(DisplayMessage);
```

```
// установка в качестве обработчика метода DisplayMessage
acc.Notify += DisplayMessage;       // добавляем обработчик DisplayMessage
acc.Put(20);    // добавляем на счет 20
void DisplayMessage(string message) => Console.WriteLine(message);
```

В данном случае разницы между двумя обработчиками никакой не будет.

Установка в качестве обработчика анонимного метода:

```
Account acc = new Account(100);
acc.Notify += delegate (string mes)
{
   Console.WriteLine(mes);
};
acc.Put(20);
```

Установка в качестве обработчика лямбда-выражения:

```
Account account = new Account(100);
account.Notify += message => Console.WriteLine(message);
account.Put(20);
```

Списки вызовов

Сильная сторона делегатов состоит в том, что они позволяют делегировать выполнение некоторому коду извне. И на момент написания программы мы можем не знать, что за код будет выполняться. Мы просто вызываем делегат. А какой метод будет непосредственно выполняться при вызове делегата, будет решаться потом.

Рассмотрим подробный пример. Пусть у нас есть класс, описывающий счет в банке:

```
public class Account
{
   int sum; // Переменная для хранения суммы
   // через конструктор устанавливается начальная сумма на счете
   public Account(int sum) => this.sum = sum;
   // добавить средства на счет
   public void Add(int sum) => this.sum += sum;
   // взять деньги с счета
   public void Take(int sum)
   {
      // берем деньги, если на счете достаточно средств
      if (this.sum >=sum) this.sum -= sum;
   }
}
```

В переменной sum хранится сумма на счете. С помощью конструктора устанавливается начальная сумма на счете. Метод Add() служит для добавления на счет, а метод Take - для снятия денег со счета.

Допустим, в случае вывода денег с помощью метода Take нам надо как-то уведомлять об этом самого владельца счета и, может быть, другие объекты. Если речь идет о консольной программе, и класс будет применяться в том же проекте, где он создан, то мы можем написать просто:

```
public class Account
{
   int sum;
   public Account(int sum) => this.sum = sum;
   public void Add(int sum) => this.sum += sum;
```

```
    public void Take(int sum)
    {
        if (this.sum >= sum)
        {
            this.sum -= sum;
            Console.WriteLine($"Со счета списано {sum} у.е.");
        }
    }
}
```

Но что если наш класс планируется использовать в других проектах, например, в графическом приложении на Windows Forms или WPF, в мобильном приложении, в веб-приложении. Там строка уведомления

```
Console.WriteLine($"Со счета списано {sum} у.е.");
```

не будет иметь большого смысла.

И делегаты позволяют делегировать определение действия из класса во внешний код, который будет использовать этот класс.

Изменим класс, применив делегаты:

```
// Объявляем делегат
public delegate void AccountHandler(string message);
public class Account
{
    int sum;
    // Создаем переменную делегата
    AccountHandler? taken;
    public Account(int sum) => this.sum = sum;
    // Регистрируем делегат
    public void RegisterHandler(AccountHandler del)
    {
        taken = del;
    }
    public void Add(int sum) => this.sum += sum;
    public void Take(int sum)
    {
        if (this.sum >= sum)
        {
            this.sum -= sum;
            // вызываем делегат, передавая ему сообщение
            taken?.Invoke($"Со счета списано {sum} у.е.");
        }
        else
        {
            taken?.Invoke($"Недостаточно средств. Баланс: {this.sum} у.е.");
        }
    }
}
```

Для делегирования действия здесь определен делегат AccountHandler. Этот делегат соответствует любым методам, которые имеют тип void и принимают параметр типа string.

```
public delegate void AccountHandler(string message);
```
В классе Account определяем переменную taken, которая представляет этот делегат:
```
AccountHandler? taken;
```
Теперь надо связать эту переменную с конкретным действием, которое будет выполняться. Мы можем использовать разные способы для передачи делегата в класс. В данном случае определяется специальный метод RegisterHandler, который передается в переменную taken реальное действие:
```
public void RegisterHandler(AccountHandler del)
{
    taken = del;
}
```
Таким образом, делегат установлен, и теперь его можно вызывать. Вызов делегата производится в методе Take:
```
public void Take(int sum)
{
    if (this.sum >= sum)
    {
        this.sum -= sum;
        // вызываем делегат, передавая ему сообщение
        taken?.Invoke($"Со счета списано {sum} у.е.");
    }
    else
    {
        taken?.Invoke($"Недостаточно средств. Баланс: {this.sum} у.е.");
    }
}
```
Поскольку делегат AccountHandler в качестве параметра принимает строку, то при вызове переменной taken() мы можем передать в этот вызов конкретное сообщение. В зависимости от того, произошло снятие денег или нет, в вызов делегата передаются разные сообщения.

То есть фактически вместо делегата будут выполняться действия, которые переданы делегату в методе RegisterHandler. Причем опять же подчеркну, при вызове делегата мы не значем, что это будут действия. Здесь мы только передаем в эти действия сообщение об успешно или неудачном снятии.

В результате, если мы создаем консольное приложение, мы можем через делегат выводить сообщение на консоль. Если мы создаем графическое приложение Windows Forms или WPF, то можно выводить сообщение в виде графического окна. А можно не просто выводить сообщение. А, например, записать при списании информацию об этом действии в файл или отправить уведомление на электронную почту. В общем любыми способами обработать вызов делегата. И способ обработки не будет зависеть от класса Account.

Добавление и удаление методов в делегате

Хотя в примере наш делегат принимал адрес на один метод, в действительности он может указывать сразу на несколько методов. Кроме того, при необходимости мы можем удалить ссылки на адреса определенных методов, чтобы они не вызывались при вызове делегата. Итак, изменим в классе Account метод RegisterHandler и добавим новый метод UnregisterHandler, который будет удалять методы из списка методов делегата:
```
public delegate void AccountHandler(string message);
```

```
public class Account
{
    int sum;
    AccountHandler? taken;
    public Account(int sum) => this.sum = sum;
    // Регистрируем делегат
    public void RegisterHandler(AccountHandler del)
    {
        taken += del;
    }
    // Отмена регистрации делегата
    public void UnregisterHandler(AccountHandler del)
    {
        taken -= del; // удаляем делегат
    }
    public void Add(int sum) => this.sum += sum;
    public void Take(int sum)
    {
        if (this.sum >= sum)
        {
            this.sum -= sum;
            taken?.Invoke($"Со счета списано {sum} у.е.");
        }
        else
            taken?.Invoke($"Недостаточно средств. Баланс: {this.sum} у.е.");
    }
}
```

В первом методе объединяет делегаты taken и del в один, который потом присваивается переменной taken. Во втором методе из переменной taken удаляется делегат del.

25.Атрибуты. Сериализация

Атрибуты

Атрибуты в .NET представляют специальные инструменты, которые позволяют встраивать в сборку дополнительные метаданные. Атрибуты могут применяться как ко всему типу (классу, интерфейсу и т.д.), так и к отдельным его частям (методу, свойству и т.д.). Основу атрибутов составляет класс System.Attribute, от которого образованы все остальные классы атрибутов. В .NET имеется множество встроенных классов атрибутов. И также мы можем создавать свои собственные классы атрибутов, которые будут определять метаданные других типов.

Допустим, нам надо проверять пользователя на соответствие некоторым возрастным ограничениям. Создадим свой атрибут, который будет хранить пороговое значение возраста, с которого разрешены некоторые действия:

```
class AgeValidationAttribute : Attribute
{
    public int Age { get;}
    public AgeValidationAttribute() { }
    public AgeValidationAttribute(int age) => Age = age;
}
```

По сути это обычный класс, унаследованный от System.Atribute. В нем определено два конструктора: с параметром и без. В качестве параметра второй конструктор атрибута принимает некий пороговый возраст и сохраняет его в свойстве.

Теперь применим его к некоторому классу:

```
 [AgeValidation(18)]
public class Person
{
    public string Name { get;}
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

Данный класс Person применяет атрибут. Для этого имя атрибута указывается в квадратных скобках непосредственно перед определением класса. Причем суффикс Attribute указывать необязательно. Обе записи [AgeValidation(18)] и [AgeValidationAttribute(18)] будут равноправны.

Если конструктор атрибута предусматривает использование параметров (public AgeValidationAttribute(int age)), то после имени атрибута мы можем указать значения для параметров конструктора. В данном случае передается значение для параметра age. То есть фактически мы говорим, что в AgeValidationAttribute свойство Age будет иметь значение 18.

Теперь получим атрибут класса Person и используем его для проверки объектов данного класса:

```
Person tom = new Person("Tom", 35);
Person bob = new Person("Bob", 16);
bool tomIsValid = ValidateUser(tom);    // true
bool bobIsValid = ValidateUser(bob);    // false
```

```
Console.WriteLine($"Результат валидации Тома: {tomIsValid}");
Console.WriteLine($"Результат валидации Боба: {bobIsValid}");
bool ValidateUser(Person person)
{
   Type type = typeof(Person);
   // получаем все атрибуты класса Person
   object[] attributes = type.GetCustomAttributes(false);
   // проходим по всем атрибутам
   foreach (Attribute attr in attributes)
   {
      // если атрибут представляет тип AgeValidationAttribute
      if (attr is AgeValidationAttribute ageAttribute)
         // возвращаем результат проверки по возрасту
         return person.Age >= ageAttribute.Age;
   }
   return true;
}
class AgeValidationAttribute : Attribute
{
   public int Age { get;}
   public AgeValidationAttribute() { }
   public AgeValidationAttribute(int age) => Age = age;
}
[AgeValidation(18)]
public class Person
{
   public string Name { get;}
   public int Age { get; set; }
   public Person(string name, int age)
   {
      Name = name;
      Age = age;
   }
}
```

В данном случае в методе ValidateUser через параметр получаем некоторый объект Person и с помощью метода GetCustomAttributes вытаскиваем из типа Person все атрибуты. Далее берем из атрибутов атрибут AgeValidationAttribute при его наличии (ведь мы можем его и не применять к классу) и проверям допустимость возраста пользователя. Если пользователь прошел проверку по возрасту, то возвращаем true, иначе возвращаем false. Если атрибут не применяется, возвращаем true.

Ограничение применения атрибута

С помощью атрибута AttributeUsage можно ограничить типы, к которым будет применяться атрибут. Например, мы хотим, чтобы выше определенный атрибут мог применяться только к классам:

```
 [AttributeUsage(AttributeTargets.Class)]
class AgeValidationAttribute : Attribute
{
//.....................................
```

```
}
```
Ограничение задает перечисление AttributeTargets, которое может принимать еще ряд значений:
All: используется всеми типами
Assembly: атрибут применяется к сборке
Constructor: атрибут применяется к конструктору
Delegate: атрибут применяется к делегату
Enum: применяется к перечислению
Event: атрибут применяется к событию
Field: применяется к полю типа
Interface: атрибут применяется к интерфейсу
Method: применяется к методу
Property: применяется к свойству
Struct: применяется к структуре
С помощью логической операции ИЛИ можно комбинировать эти значения. Например, пусть атрибут может применяться к классам и структурам:
```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
```
Сериализация

Сериализация представляет процесс преобразования какого-либо объекта в поток байтов. После преобразования мы можем этот поток байтов или записать на диск или сохранить его временно в памяти. А при необходимости можно выполнить обратный процесс - десериализацию, то есть получить из потока байтов ранее сохраненный объект.

Атрибут Serializable

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом Serializable:
```
[Serializable]
class Person
{
    public string Name { get; set; }
    public int Year { get; set; }

    public Person(string name, int year)
    {
        Name = name;
        Year = year;
    }
}
```
При отстутствии данного атрибута объект Person не сможет быть сериализован, и при попытке сериализации будет выброшено исключение SerializationException.
Сериализация применяется к свойствам и полям класса. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом NonSerialized:
```
 [Serializable]
class Person
{
    public string Name { get; set; }
    public int Year { get; set; }

    [NonSerialized]
```

```
    public string accNumber;
    public Person(string name, int year, string acc)
    {
        Name = name;
        Year = year;
        accNumber = acc;
    }
}
```

При наследовании подобного класса, следует учитывать, что атрибут Serializable автоматически не наследуется. И если мы хотим, чтобы производный класс также мог бы быть сериализован, то опять же мы применяем к нему атрибут:

```
 [Serializable]
class Worker : Person
```

Формат сериализации

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

- бинарный
- SOAP
- xml
- JSON

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

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

```
public interface IFormatter
{
    SerializationBinder Binder { get; set;}
    StreamingContext Context { get; set;}
    ISurrogateSelector SurrogateSelector  { get; set;}
    object Deserialize (Stream serializationStream);
    void Serialize (Stream serializationStream, object graph);
}
```

Хотя классы BinaryFormatter и SoapFormatter по-разному реализуют данный интерфейс, но общий функционал будет тот же: для сериализации будет использоваться метод Serialize, который в качестве параметров принимает поток, куда помещает сериализованные данные (например, бинарный файл), и объект, который надо сериализовать. А для десериализации будет применяться метод Deserialize, который в качестве параметра принимает поток с сериализованными данными.

Класс XmlSerializer не реализует интерфейс IFormatter и по функциональности в целом несколько отличается от BinaryFormatter и SoapFormatter, но и он также предоставляет для сериализации метод Serialize, а для десериализации Deserialize. И в этом плане очень легко при необходимости перейти от одного способа сериализации к другому.

```
Person person1 = new Person("Tom", 29);
Person person2 = new Person("Bill", 25);
// массив для сериализации
Person[] people = new Person[] { person1, person2 };
```

```
BinaryFormatter formatter = new BinaryFormatter();
using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))
{
   // сериализуем весь массив people
   formatter.Serialize(fs, people);

   Console.WriteLine("Объект сериализован");
}
// десериализация
using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))
{
   Person[] deserilizePeople = (Person[])formatter.Deserialize(fs);

   foreach (Person p in deserilizePeople)
   {
      Console.WriteLine($"Имя: {p.Name} --- Возраст: {p.Age}");
   }
```

26.Шаблоные типы. Nullable-типы.

Обобщения

Кроме обычных типов фреймворк .NET также поддерживает обобщенные типы (generics), а также создание обобщенных методов.

Мы можем определить свойство Id как свойство типа object. Так как тип object является универсальным типом, от которого наследуется все типы, соответственно в свойствах подобного типа мы можем сохранить и строки, и числа:

```
class Person
{
   public object Id { get;}
   public string Name { get;}
   public Person(object id, string name)
   {
      Id = id;
      Name = name;
   }
}
```

Затем этот класс можно было использовать для создания пользователей в программе:

```
Person tom = new Person(546, "Tom");
Person bob = new Person("abc123", "Bob");
int tomId = (int)tom.Id;
string bobId = (string) bob.Id;
Console.WriteLine(tomId);   // 546
Console.WriteLine(bobId);   // abc123
```

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

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

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

```
Person tom = new Person(546, "Tom");
string tomId = (string)tom.Id;  // !Ошибка  - Исключение InvalidCastException
Console.WriteLine(tomId);   // 546
```

Мы можем не знать, какой именно объект представляет Id, и при попытке получить число в данном случае мы столкнемся с исключением InvalidCastException. Причем с исключением мы столкнемся на этае выполнения программы.

Для решения этих проблем в язык C# была добавлена поддержка обобщенных типов (также часто называют универсальными типами). Обобщенные типы позволяют указать конкретный тип, который будет использоваться. Поэтому определим класс Person как обощенный:

```
class Person<T>
{
    public T Id { get; set; }
    public string Name { get; set; }
    public Person(T id, string name)
    {
        Id = id;
        Name = name;
    }
}
```

Угловые скобки в описании class Person<T> указывают, что класс является обобщенным, а тип T, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву T, это может быть и любая другая буква или набор символов. Причем сейчас 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.

Например, вместо параметра T можно использовать объект int, то есть число, представляющее номер пользователя.

Поскольку класс Person является обобщенным, то при определении переменной после названия типа в угловых скобках необходимо указать тот тип, который будет использоваться вместо универсального параметра T.

При попытке передать для параметра id значение другого типа мы получим ошибку компиляции:

```
Person<int> tom = new Person<int>("546", "Tom");  // ошибка компиляции
```

А при получении значения из Id нам больше не потребуется операция приведения типов и распаковка тоже применяться не будет:

```
int tomId = tom.Id;      // распаковка не нужна
```

Тем самым мы избежим проблем с типобезопасностью. Таким образом, используя обобщенный вариант класса, мы снижаем время на выполнение и количество потенциальных ошибок.
При этом универсальный параметр также может представлять обобщенный тип:

```
// класс компании
class Company<P>
{
    public P CEO { get; set; }  // президент компании
    public Company(P ceo)
    {
        CEO = ceo;
    }
}
class Person<T>
{
    public T Id { get;}
    public string Name { get;}
    public Person(T id, string name)
    {
        Id = id;
        Name = name;
    }
}
```

Здесь класс компании определяет свойство CEO, которое хранит президента компании.
Статические поля обобщенных классов
При типизации обобщенного класса определенным типом будет создаваться свой набор статических членов. Например, в классе Person определено следующее статическое поле:

```
class Person<T>
{
    public static T? code;
    public T Id { get; set; }
    public string Name { get; set; }
    public Person(T id, string name)
    {
        Id = id;
        Name = name;
    }
}
```

Теперь типизируем класс двумя типами int и string:

```
Person<int> tom = new Person<int>(546, "Tom");
Person<int>.code = 1234;

Person<string> bob = new Person<string>("abc", "Bob");
Person<string>.code = "meta";

Console.WriteLine(Person<int>.code);       // 1234
Console.WriteLine(Person<string>.code);   // meta
```

В итоге для Person<string> и для Person<int> будет создана своя переменная code.

Использование нескольких универсальных параметров

Обобщения могут использовать несколько универсальных параметров одновременно, которые могут представлять одинаковые или различные типы:

```
class Person<T, K>
{
   public T Id { get;}
   public K Password { get; set; }
   public string Name { get;}
   public Person(T id, K password, string name)
   {
      Id = id;
      Name = name;
      Password = password;
   }
}
```

Обобщенные методы

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

```
int x = 7;
int y = 25;
Swap<int>(ref x, ref y); // или так Swap(ref x, ref y);
Console.WriteLine($"x={x}    y={y}");   // x=25   y=7
string s1 = "hello";
string s2 = "bye";
Swap<string>(ref s1, ref s2); // или так Swap(ref s1, ref s2);
Console.WriteLine($"s1={s1}    s2={s2}"); // s1=bye   s2=hello
void Swap<T>(ref T x, ref T y)
{
   T temp = x;
   x = y;
   y = temp;
}
```

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

Null и значимые типы

В отличие от ссылочных типов переменным/параметрам значимых типов нельзя напрямую присвоить значение null. Тем не менее нередко бывает удобно, чтобы переменная/параметр значимого типа могли принимать значение null, Например, получаем числовое значение из базы данных, которое в бд может отсутствовать. То есть, если значение в базе данных есть - получим число, если нет - то null.

Чтобы присвоения переменной или параметру значимого типа значения null, эти переменная/параметр значимого типа должны представлять тип nullable. Для этого после названия типа указывается знак вопроса ?

```
int? val = null;
Console.WriteLine(val);
```

Здесь переменная val представляет не просто тип int, а тип int? - тип, переменные/параметры которого могут принимать как значения типа int, так и значение

null. В данном случае мы передаем ей значение null. Но также можно передать и значение типа int:

```
int? val = null;
IsNull(val);    // null
val = 22;
IsNull(val);    // 22
void IsNull(int? obj)
{
   if (obj == null) Console.WriteLine("null");
   else Console.WriteLine(obj);
}
```

Однако если переменная/параметр представляет значимый не nullable-тип, то присвоить им значение null не получится:

```
int val = null;  // ! ошибка, переменная val НЕ представляет тип nullable
```

Стоит отметить, что фактически запись ? для значимых типов является упрощенной формой использования структуры System.Nullable<T>. Параметр T в угловых скобках представляет универсальный параметр, вместо которого в программе подставляется конкретный тип данных. Следующие виды определения переменных будут эквивалентны:

```
int? number1 = 5;
Nullable<int> number2 = 5;
```

Свойства Value и HasValue и метод GetValueOrDefault

Структура Nullable<T> имеет два свойства:

- Value - значение объекта
- HasValue: возвращает true, если объект хранит некоторое значение, и false, если объект равен null.

Мы можем использовать эти свойства для проверки наличия и получения значения:

```
PrintNullable(5);       // 5
PrintNullable(null);    // параметр равен null
void PrintNullable(int? number)
{
   if (number.HasValue)
   {
      Console.WriteLine(number.Value);
      // аналогично
      Console.WriteLine(number);
   }
   else
   {
      Console.WriteLine("параметр равен null");
   }
}
```

Однако если мы попробуем получить через свойство Value значение переменной, которая равна null, то мы столкнемся с ошибкой:

```
int? number = null;
Console.WriteLine(number.Value);    // ! Ошибка
Console.WriteLine(number);          // Ошибки нет - просто ничего не выведет
```

Также структура Nullable<T> имеет метод GetValueOrDefault(). Он возвращает значение переменной/параметра, если они не равны null. Если они равны null, то возвращается значение по умолчанию. Значение по умолчанию можно передать в метод. Если в метод не передается данных, то возвращается значение по умолчанию для данного типа данных (например, для числовых данных это число 0).

```
int? number = null; // если значения нет, метод возвращает значение по умолчанию
Console.WriteLine(number.GetValueOrDefault());      // 0  - значение по умолчанию для числовых типов
Console.WriteLine(number.GetValueOrDefault(10));    // 10
number = 15;    // если значение задано, оно возвращается методом
Console.WriteLine(number.GetValueOrDefault());    // 15
Console.WriteLine(number.GetValueOrDefault(10));  // 15
```

Преобразование значимых nullable-типов

Рассмотрим возможные преобразования:

- явное преобразование от T? к T

```
int? x1 = null;
if(x1.HasValue)
{
   int x2 = (int)x1;
   Console.WriteLine(x2);
}
```

- неявное преобразование от T к T?

```
int x1 = 4;
int? x2 = x1;
Console.WriteLine(x2);
```

- неявные расширяющие преобразования от V к T?

```
int x1 = 4;
long? x2 = x1;
Console.WriteLine(x2);
```

- явные сужающие преобразования от V к T?

```
long x1 = 4;
int? x2 = (int?)x1;
```

- Подобным образом работают явные сужающие преобразования от V? к T?

```
long? x1 = 4;
int? x2 = (int?)x1;
```

- явные сужающие преобразования от V? к T

```
long? x1 = null;
if (x1.HasValue)
{
   int x2 = (int)x1;
}
```

Операции с nullable-типами

nullable-типы поддерживают тот же набор операций, что и их не-nullable двойники. Но следует учитывать, что если в операции участвует nullable-тип, то результатом также будет значение nullable-типа

```
int? x = 5;
int z = x + 7;          // нельзя
int? w = x + 7;         // можно
```

int d = x.Value + 7; // можно
В арифметических операциях, если один из операндов равен null, то результатом операции также будет null:
int? x = null;
int? w = x + 7; // w = null
В операциях сравнения, если хотя бы один из операндов равен null, то возвращается false

27.Неуправляемый код. Указатели. Маршалинг.

Управляемый код?
Код, который написан с целью получения услуг управляемой среды выполнения, таких как CLR (Common Language Runtime) в .NET Framework, известен как управляемый код. Он всегда реализуется управляемой средой выполнения вместо непосредственного выполнения операционной системой. Управляемая среда выполнения предоставляет различные типы сервисов, таких как сборка мусора, проверка типов, обработка исключений, проверка границ и т.д. для автоматического программирования без вмешательства программиста. Он также обеспечивает выделение памяти, безопасность типов и т.д. для кода. Приложение написано на таких языках, как Java, C #, VB.Net И т.д. всегда предназначены для служб среды выполнения для управления выполнением, и код, написанный на этих типах языков, известен как управляемый код.
В случае .NET Framework компилятор всегда компилирует управляющий код на промежуточном языке (MSIL), а затем создает исполняемый файл. Когда программист запускает исполняемый файл, то компилятор Just In Time CLR компилирует промежуточный язык в машинный код, специфичный для базовой архитектуры. Здесь этот процесс происходит в управляемой среде выполнения runtime, поэтому эта среда отвечает за работу кода.
Каковы преимущества использования управляемого кода?

- Это повышает безопасность приложения, например, при использовании среды выполнения автоматически проверяет буферы памяти на предмет переполнения буфера.
- Он автоматически реализует сборку мусора.
- Он также обеспечивает проверку типов во время выполнения / динамическую проверку типов.
- Он также обеспечивает проверку ссылок, что означает, что он проверяет, указывает ли ссылка на допустимый объект или нет, а также проверяет, не дублируются ли они.

Каковы недостатки управляемого кода?
Основным недостатком управляемого языка является то, что вам не разрешено выделять память напрямую, или вы не можете получить низкоуровневый доступ к архитектуре процессора.
Неуправляемый код
Код, который непосредственно выполняется операционной системой, известен как неуправляемый код. Он всегда ориентирован на архитектуру процессора и зависит от архитектуры компьютера. Когда этот код компилируется, он всегда стремится получить определенную архитектуру и всегда выполняться на этой платформе, другими словами, всякий раз, когда вы хотите выполнить тот же код для другой архитектуры, вам приходится перекомпилировать этот код снова в соответствии с этой архитектурой. Он всегда компилируется в машинный код, специфичный для данной архитектуры.

В неуправляемом коде распределением памяти, безопасностью типов и т.д. управляет разработчик. Из-за этого возникает несколько проблем, связанных с памятью, таких как переполнение буфера, утечка памяти, переопределение указателя и т.д. Исполняемые файлы неуправляемого кода обычно представляют собой двоичные образы, код x86, который загружается непосредственно в память. Приложения, написанные на VB 6.0, C, C ++ и т.д.,всегда находятся в неуправляемом коде.

Каковы преимущества использования неуправляемого кода?

- Он предоставляет программисту низкоуровневый доступ.
- Он также обеспечивает прямой доступ к оборудованию.
- Это позволяет программисту обходить некоторые параметры и ограничения, которые используются платформой управляемого кода.

Каковы недостатки неуправляемого кода?

- Это не обеспечивает безопасность приложения.
- Из-за доступа к распределению памяти возникают проблемы, связанные с памятью, такие как переполнение буфера памяти и т.д.
- Ошибки и исключения также обрабатываются программистом.
- Он не фокусируется на сборке мусора.

Небезопасный - код, который может находиться за пределами проверяемого подмножества CIL.(испол-ие указателей)

Неуправляемый - код, который не управляется средой выполнения и, следовательно, не виден GC.

Указатели

Указатели позволяют получить доступ к определенной ячейке памяти и произвести определенные манипуляции со значением, хранящимся в этой ячейке.

В языке C# указатели редко используются, однако в некоторых случаях можно прибегать к ним для оптимизации приложений. Код, применяющий указатели, еще называют небезопасным (unsafe) кодом. Однако это не значит, что он представляет какую-то опасность. Просто при работе с ним все действия по использованию памяти, в том числе по ее очистке, ложится целиком на нас, а не на среду CLR. И с точки зрения CLR такой код не безопасен, так как среда не может проверить данный код, поэтому повышается вероятность различного рода ошибок.

Чтобы использовать небезопасный код в C#, надо первым делом указать проекту, что он будет работать с небезопасным кодом. Для этого надо установить в настройках проекта соответствующий флаг - в меню Project (Проект) найти Свойства проекта. Затем в меню Build установить флажок Unsafe code (Небезопасный код).

Теперь мы можем приступать к работе с небезопасным кодом и указателями.

Ключевое слово unsafe

Блок кода или метод, в котором используются указатели, помечается ключевым словом unsafe:

```
// блок кода, использующий указатели
unsafe
{

}
```

Метод, использующий указатели:

```
unsafe void Test()
{
```

```
}
```
Также с помощью unsafe можно объявлять структуры и классы:
```
unsafe struct State
{

}

unsafe class Person
{

}
```
Операции * и &

Ключевой при работе с указателями является операция *, которую еще называют операцией разыменовывания. Операция разыменовывания позволяет получить или установить значение по адресу, на который указывает указатель. Для получения адреса переменной применяется операция &:
```
unsafe
{
    int* x; // определение указателя
    int y = 10; // определяем переменную

    x = &y; // указатель x теперь указывает на адрес переменной y
    Console.WriteLine(*x); // 10

    y = y + 20;     // меняем значение
    Console.WriteLine(*x);// 30

    *x = 50;
    Console.WriteLine(y); // переменная y=50
}
```
При объявлении указателя указываем тип int* x; - в данном случае объявляется указатель на целое число. Но кроме типа int можно использовать и другие: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal или bool. Также можно объявлять указатели на типы enum, структуры и другие указатели.

Выражение x = &y; позволяет нам получить адрес переменной y и установить на него указатель x. До этого указатель x не на что не указывал.

После этого все операции с y будут влиять на значение, получаемое через указатель x и наоборот, так как они указывают на одну и ту же область в памяти.

Для получения значения, которое хранится в области памяти, на которую указывает указатель x, используется выражение *x.

Получение адреса

Используя преобразование указателя к целочисленному типу, можно получить адрес памяти, на который указывает указатель:
```
int* x; // определение указателя
int y = 10; // определяем переменную
x = &y; // указатель x теперь указывает на адрес переменной y
// получим адрес переменной y
ulong addr = (ulong)x;
```

```
Console.WriteLine($"Адрес переменной y: {addr}");
```
Для получения адреса используется преобразование в тип uint, long или ulong. Так как значение адреса - это целое число, а на 32-разрядных системах диапазон адресов 0 до 4 000 000 000, а адрес можно получить в переменную uint/int. Соответственно на 64-разрядных системах диапазон доступных адресов гораздо больше, поэтому в данном случае лучше использовать ulong, чтобы избежать ошибки переполнения.

Указатель на другой указатель

Объявление и использование указателя на указатель:

```
unsafe
{
    int* x; // определение указателя
    int y = 10; // определяем переменную

    x = &y; // указатель x теперь указывает на адрес переменной y
    int** z = &x; // указатель z теперь указывает на адрес, указателя x
    **z = **z + 40; // изменение указателя z повлечет изменение переменной y
    Console.WriteLine(y); // переменная y=50
    Console.WriteLine(**z); // переменная **z=50
}
```

Маршалинг

Маршалирование — это процесс преобразования типов, когда они должны пересекать управляемый и машинный код.

Необходимость в маршалинге вызвана различием типов в управляемом и неуправляемом коде. Например, в управляемом коде имеется String, а в неуправляемом строки могут иметь различный формат: Юникод, отличный от Юникода, с конечным символом NULL, ASCII и т. д. По умолчанию подсистема P/Invoke пытается выбрать правильное решение в зависимости от реакции на событие по умолчанию. Однако в ситуациях, когда требуется дополнительный контроль, можно применить атрибут MarshalAs, чтобы указать ожидаемый тип на стороне неуправляемого кода. Например, если строку нужно передать в виде строки ANSI с конечным символом NULL, это можно сделать следующим образом:

```
 [DllImport("somenativelibrary.dll")]
static extern int MethodA([MarshalAs(UnmanagedType.LPStr)] string parameter);
```

Другим аспектом маршалинга типов является способ передачи структуры в неуправляемый метод. Например, некоторые неуправляемые методы требуют указания структуры в качестве параметра. В таких случаях необходимо создать соответствующую структуру или соответствующий класс в управляемом коде и использовать в качестве параметра. Однако простого определения класса недостаточно, необходимо также указать маршалировщику, как сопоставлять поля в классе с неуправляемой структурой. Здесь пригодится атрибут StructLayout.

```
 [DllImport("kernel32.dll")]
static extern void GetSystemTime(SystemTime systemTime);
[StructLayout(LayoutKind.Sequential)]
class SystemTime {
    public ushort Year;
    public ushort Month;
    public ushort DayOfWeek;
    public ushort Day;
```

```
    public ushort Hour;
    public ushort Minute;
    public ushort Second;
    public ushort Millisecond;
}
public static void Main(string[] args) {
    SystemTime st = new SystemTime();
    GetSystemTime(st);
    Console.WriteLine(st.Year);
}
```

В приведенном выше примере представлен код вызова функции GetSystemTime(). Интерес представляет строка 3. Атрибут указывает, что поля класса должны последовательно сопоставляться со структурой на другой (неуправляемой) стороне. Это означает, что имена полей не важны, важен только их порядок, так как он должен соответствовать неуправляемой структуре, которая показана в приведенном ниже примере:

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

28.Динамическое связывание. dynamic

Связывание — это процесс разрешения, или соотнесения идентификаторов типов (членов типа, операций) с логическими частями программы, представляющими соответствующие типы. Как правило это процесс происходит во время компиляции, когда компилятор находит в исходном коде идентификаторы типов и соотносит их с соответствующей логикой.
Динамическое связывание позволяет отложить этот процесс с момента компиляции до момента выполнения. Динамическое связывание может быть полезно, когда на момент компиляции компилятор не может знать о существовании какого-либо типа (члена типа, функции), но мы можем быть точно уверены, что на момент выполнения программы соответствующий тип будет существовать. Такая ситуация может возникнуть при взаимодействии, например, с динамическими языками, COM или в сценариях использующих отражение (reflection).
Динамический тип объявляется с использованием ключевого слова dynamic:

```
dynamic d = GetSomeObject();
d.Quack();
```

Динамический тип дает указание компилятору не выполнять связывания. Мы предполагаем, что динамический тип d будет иметь метод Quack, но на момент компиляции мы не можем этого гарантировать. Так как d — динамический тип, компилятор отложит связывание для него и его метода Quack до момента выполнения.
Статическое связывание (Static Binding) и динамическое связывание (Dynamic Binding)
Пример связывания — сопоставление имени с конкретной функцией при компиляции выражения. Чтобы скомпилировать следующее выражение, компиляторы необходимо найти реализацию метода с именем Quack:

```
d.Quack();
```

Предположим, что d имеет статический тип Duck:

```
Duck d = ...
d.Quack();
```

В простейшем случае компилятор выполнит связывание, найдя не принимающий параметров метод с именем Quack в классе Duck. Если такой метод найти не удастся, компилятор расширит поиск до методов принимающих необязательные параметры, методов базового класса для класса Duck и методов расширений, принимающих Duck в качестве первого параметра. Если совпадений не будет найдено, произойдет ошибка компиляции. Независимо от того, какой метод будет привязан, суть состоит в том, что связывание будет завершено компилятором. При этом связывание полностью зависит от того, известны ли на момент компиляции все необходимые типы. Такое связывание называется статическим.
Теперь заменим статический тип d на object:

```
object d = ...
d.Quack();
```

Вызов метода Quack вызовет ошибку при компиляции, т.к. хотя объект сохраненный в d может и содержать метод Quack, компилятор не может об этом знать, поскольку он располагает только информацией, основанной на типе переменной, а в данном случае это object.
Теперь заменим статический тип d на dynamic:

```
dynamic d = ...
d.Quack();
```

Тип dynamic схож с типом object — он также не несет никакой информации о реальном типе объекта. Разница состоит в том, что его можно использовать теми способами, о которых во время компиляции еще ничего не известно. Динамический объект связывается во время выполнения и основывается он на типе, который приобретает во время выполнения, а не на том, который ему задан при компиляции. Когда компилятор встречает динамически связанное выражение (любое выражение, содержащее значение с типом dynamic), он просто упаковывает выражение таким образом, чтобы связывание можно было осуществить позже, во время выполнения.
Во время выполнения, если динамический объект реализует интерфейс IDynamicMetaObjectProvider, этот интерфейс используется для осуществления связывания. В противном случае связывание осуществляется так, как если бы реальный тип динамического объекта уже был бы известен компилятору. Эти два альтернативных способа называются пользовательское связывание (custom binding) и языковое связывание (language binding).
Пользовательское связывание (Custom Binding)
Пользовательское связывание происходит, когда динамически объект реализует интерфейс IDynamicMetaObjectProvider (IDMOP). Типы, написанные на C#, могут реализовывать интерфейс IDMOP и это может быть полезно, но ,как правило, объекты реализующие данный интерфейс импортируются из динамических языков, реализованных в .NET на Dynamic Language Runtime (DLR), например IronPython или IronRuby. Объекты из этих языков автоматически реализуют интерфейс IDMOP и поэтому автоматически контролируют все операции выполняемые с ними.

```
using System;
using System.Dynamic;
public class Test
{
   static void Main()
   {
      dynamic d = new Duck();
      d.Quack();
      d.Waddle();
   }
}
public class Duck : DynamicObject
{
   public override bool TryInvokeMember (
      InvokeMemberBinder binder,
      object[] args,
      out object result)
   {
      Console.WriteLine (binder.Name + " was called");
      result = null;
      return true;
   }
}
```

Класс Duck не содержит метода Quack, вместо этого он использует пользовательское связывание, чтобы перехватывать и интерпретировать вызов всех методов.

Языковое связывание (Language Binding)

Языковое связывание происходит, когда динамический объект не реализует интерфейс IDynamicMetaObjectProvider. Языковое связывание может быть полезно при работе с не полностью разработанными типами или неотъемлемыми ограничениями системы типов .NET. Например, при работе с числовыми типами всегда возникает проблема с тем, что они не имеют общего интерфейса. Поэтому методы и операторы, взаимодействующие с числовыми типами могут использовать динамическое связывание, чтобы обойти эту проблему:

```
static dynamic Mean (dynamic x, dynamic y)
{
   return (x + y) / 2;
}
static void Main()
{
   int x = 3, y = 4;
   Console.WriteLine (Mean (x, y));
}
```

Выгода очевидна — нет необходимости дублировать код для каждого числового типа. Однако такой подход менее безопасен, чем статическое связывание: ошибки связывания будут возникать не во время компиляции, а при выполнении программы. Однако динамическое связывание более безопасно чем отражение (reflection), т.к. не отменяет правила доступности членов.

Динамическое связывание отрицательно влияет на производительность. Правда DLR имеет встроенный механизм кэширования повторных вызовов, позволяя оптимизировать использование динамических выражений в циклах.

RuntimeBinderException

Если динамическое связывание выполнить не удается, выбрасывается исключение RuntimeBinderException:

```
dynamic d = 5;
d.Hello(); // выбрасывается RuntimeBinderException
```

dynamic и object

Типы dynamic и object практически идентичны. Например, следующее выражение при выполнении вернет true:

```
typeof (dynamic) == typeof (object)
```

Этот принцип распространяется на составные типы и массивы:

```
typeof (List<dynamic>) == typeof (List<object>)
typeof (dynamic[]) == typeof (object[])
```

Как и ссылка типа object, ссылка типа dynamic может ссылаться на объекты любого типа:

```
dynamic x = "hello";
Console.WriteLine (x.GetType().Name); // String
x = 123; // Не вызовет ошибки несмотря на использование той же переменной
Console.WriteLine (x.GetType().Name); // Int32
```

Структурно нет никаких различий между ссылкой типа object и ссылкой типа dynamic. Динамическая ссылка просто позволяет выполнять динамические операции над объектом, на который она ссылается. Можно преобразовывать object в dynamic для выполнения любых динамических операций над object:

```
object o = new System.Text.StringBuilder();
dynamic d = o;
```

```
d.Append ("hello");
Console.WriteLine (o); // hello
```

Преобразование dynamic

Тип dynamic может быть автоматически преобразован в и из любого другого типа. Чтобы преобразование увенчалось успехом, необходимо чтобы тип, который будет привязан динамическому объекту при выполнении, мог автоматически преобразовывать в/из целевого типа (тип, в/из которого осуществляется преобразование):

```
int i = 7;
dynamic d = i;
long l = d; // OK
short j = d; // исключение RuntimeBinderException
```

var и dynamic

var и тип dynamic на первый взгляд очень схожи, но на самом деле они несут разный смысл: var указывает компилятору на необходимость вычислить тип самостоятельно (при компиляции), а dynamic предписывает вычислить тип во время выполнения.

```
dynamic x = "hello"; // Статический тип - dynamic
var y = "hello"; // Статический тип - string
int i = x; // Ошибка при выполнении
int j = y; // Ошибка при компиляции
```

Динамические выражения

Поля, свойства, методы, события, конструкторы, индексаторы, операторы и преобразования — все могут быть вызваны динамически. Динамические выражения также как статические должны возвращать результат. Если попытать вместо результата вернуть void, будет зафиксирована ошибка, но не при компиляции как для статических выражений, а во время выполнения.

Выражения, принимающие динамические операнды, тоже являются динамическими:

```
dynamic x = 2;
var y = x * 3; // Статический тип y - dynamic
```

Но из этого правила есть два очевидных исключения: во-первых, приведение динамического выражения к статическому типу вернет статическое выражение; во-вторых, вызов конструктора всегда вернет статическое выражение, даже если он вызван со статическим аргументом. Кроме того, есть еще несколько случаев, когда выражение, содержащее динамический аргумент, будет статическим (например, передача индекса в массив, создание делегата).

Идентификация перегрузки динамических членов

Случай использования типа dynamic — применение динамического получателя (receiver): когда динамический объект используется как получатель для вызова динамической функции:

```
dynamic x = ...;
x.Foo (123); // x - получатель
```

Однако это не единственный вариант применения динамического связывания: аргументы методов также могут связываться динамически. Эффект от вызова функции с динамическим аргументом — возможность перенести выбор варианта перегруженного метода с момента компиляции до выполнения программы:

```
class Program
{
   static void Foo (int x) { Console.WriteLine ("1"); }
   static void Foo (string x) { Console.WriteLine ("2"); }
```

```
    static void Main()
    {
        dynamic x = 5;
        dynamic y = "watermelon";
        Foo (x); // 1
        Foo (y); // 2
    }
}
```

Выбор варианта перегруженного метода во время выполнения программы также называется множественной отсылкой (multiple dispatch) и используется в реализации паттерна гость (visitor).

Если не используется динамический получатель, компилятор может статически проверить, увенчается ли динамический вызов успехом или нет: он проверяет что функция с указанным названием и числом аргументов существует. Если подходящей функции найдено не будет, возникнет ошибка компиляции.

Если функция вызывается с несколькими аргументами, одни из которых статические, а другие динамические, то будет использовано смешанное связывание: для статических методов — статическое (во время компиляции), для динамических — динамическое (во время выполнения).

Невызываемые функции

Некоторые функции не могут быть вызваны динамически:

- методы расширения (можно вызвать как статические методы)
- члены интерфейса
- базовые члены, скрытые производным классом

Причина этого ограничения в том, что для динамического связывания необходима информация, во-первых, об имени вызываемой функции, и, во-вторых, об объекте, через который эта функция вызывается. Однако во всех трех невызываемых сценариях задействован дополнительный тип, известный только во время компиляции. Определить этот тип динамически возможности нет.

Для методов расширений этим дополнительным типом является класс в котором определены методы расширения. Этот класс указывается в директиве using в исходном коде, автоматически определяется компилятором и после компиляции исчезает). При вызове членов через интерфейс, дополнительный тип связан с приведение типов. Схожая ситуация с вызовом скрытых членов базового класса: дополнительный тип возникает в результате приведения типов или использования ключевого слова base, а во время выполнения программы этот тип недоступен.

29.Многопоточные приложения. Внедрение программного потока. Оператор lock.

Одним из ключевых аспектов в современном программировании является многопоточность. Ключевым понятием при работе с многоопоточностью является поток. Поток предствляет некоторую часть кода программы. При выполнении программы каждому потоку выделяется определенный квант времени. И при помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому, к примеру, клиент-серверные приложения (и не только они) практически не мыслимы без многопоточности.
Основной функционал для использования потоков в приложении сосредоточен в пространстве имен System.Threading. В нем определен класс, представляющий отдельный поток - класс Thread.
Класс Thread определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства класса:
ExecutionContext: позволяет получить контекст, в котором выполняется поток

- IsAlive: указывает, работает ли поток в текущий момент
- IsBackground: указывает, является ли поток фоновым
- Name: содержит имя потока
- ManagedThreadId: возвращает числовой идентификатор текущего потока
- Priority: хранит приоритет потока - значение перечисления ThreadPriority:
    - Lowest
    - BelowNormal
    - Normal
    - AboveNormal
    - Highest

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

- ThreadState возвращает состояние потока - одно из значений перечисления ThreadState:
    - Aborted: поток остановлен, но пока еще окончательно не завершен
    - AbortRequested: для потока вызван метод Abort, но остановка потока еще не произошла
    - Background: поток выполняется в фоновом режиме
    - Running: поток запущен и работает (не приостановлен)
    - Stopped: поток завершен
    - StopRequested: поток получил запрос на остановку
    - Suspended: поток приостановлен
    - SuspendRequested: поток получил запрос на приостановку
    - Unstarted: поток еще не был запущен

- WaitSleepJoin: поток заблокирован в результате действия методов Sleep или Join

В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение Unstarted. Запустив поток, мы изменим его статус на Running. Вызвав метод Sleep, статус изменится на WaitSleepJoin.

Кроме того статическое свойство CurrentThread класса Thread позволяет получить текущий поток.

В программе на C# есть как минимум один поток - главный поток, в котором выполняется метод Main.

Также класс Thread определяет ряд методов для управления потоком. Основные из них:

- Статический метод GetDomain возвращает ссылку на домен приложения
- Статический метод GetDomainID возвращает id домена приложения, в котором выполняется текущий поток
- Статический метод Sleep останавливает поток на определенное количество миллисекунд
- Метод Interrupt прерывает поток, который находится в состоянии WaitSleepJoin
- Метод Join блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
- Метод Start запускает поток

Для создания потока применяется один из конструкторов класса Thread:

- Thread(ThreadStart): в качестве параметра принимает объект делегата ThreadStart, который представляет выполняемое в потоке действие
- Thread(ThreadStart, Int32): в дополнение к делегату ThreadStart принимает числовое значение, которое устанавливает размер стека, выделяемого под данный поток
- Thread(ParameterizedThreadStart): в качестве параметра принимает объект делегата ParameterizedThreadStart, который представляет выполняемое в потоке действие
- Thread(ParameterizedThreadStart, Int32): вместе с делегатом ParameterizedThreadStart принимает числовое значение, которое устанавливает размер стека для данного потока

Вне зависимости от того, какой конструктор будет применяться для создания, нам надо определить выполняемое в потоке действие. Рассмотрим использование делегата ThreadStart. Этот делегат представляет действие, которое не принимает никаких параметров и не возвращает никакого значения:

```
public delegate void ThreadStart();
```

То есть под этот делегат нам надо определить метод, который имеет тип void и не принимает никаких параметров. Примеры определения потоков:

```
Thread myThread1 = new Thread(Print);
Thread myThread2 = new Thread(new ThreadStart(Print));
Thread myThread3 = new Thread(()=>Console.WriteLine("Hello Threads"));
void Print() => Console.WriteLine("Hello Threads");
```

Для запуска нового потока применяется метод Start класса Thread:

```
using System.Threading;
// создаем новый поток
Thread myThread1 = new Thread(Print);
Thread myThread2 = new Thread(new ThreadStart(Print));
Thread myThread3 = new Thread(()=>Console.WriteLine("Hello Threads"));
```

```
myThread1.Start();  // запускаем поток myThread1
myThread2.Start();  // запускаем поток myThread2
myThread3.Start();  // запускаем поток myThread3
void Print() => Console.WriteLine("Hello Threads");
```

Преимуществом потоком является то, что они могут выполняться одновременно. Например:

```
using System.Threading;
// создаем новый поток
Thread myThread = new Thread(Print);
// запускаем поток myThread
myThread.Start();
// действия, выполняемые в главном потоке
for (int i = 0; i < 5; i++)
{
    Console.WriteLine($"Главный поток: {i}");
    Thread.Sleep(300);
}
// действия, выполняемые во втором потокке
void Print()
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine($"Второй поток: {i}");
        Thread.Sleep(400);
    }
}
```

Здесь новый поток будет производить действия, определенные в методе Print, то есть выводить числа от 0 до 4 на консоль. Причем после каждого вывода производится задержка на 400 миллисекунд.

Кроме того, в главном потоке производим аналогичные действия - выводим на консоль числа от 0 до 4 с задержкой в 300 миллисекунд.

Таким образом, в нашей программе будут работать одновременно главный поток, представленный методом Main, и второй поток, в котором выполняется метод Print. Как только все потоки отработают, программа завершит свое выполнение. В итоге мы получим следующий консольный вывод:

```
Главный поток: 0
Второй поток: 0
Главный поток: 1
Второй поток: 1
Главный поток: 2
Второй поток: 2
Главный поток: 3
Второй поток: 3
Главный поток: 4
Второй поток: 4
```

Подобным образом мы можем создать и запускать и три, и четыре, и целый набор новых потоков, которые смогут решать те или иные задачи.

Оператор lock

Нередко в потоках используются некоторые разделяемые ресурсы, общие для всей программы. Это могут быть общие переменные, файлы, другие ресурсы. Например:

```
int x = 0;
// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";   // устанавливаем имя для каждого потока
    myThread.Start();
}
void Print()
{
    x = 1;
    for (int i = 1; i < 6; i++)
    {
        Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
        x++;
        Thread.Sleep(100);
    }
}
```

Здесь у нас запускаются пять потоков, которые вызывают метод Print и которые работают с общей переменной x. И мы предполагаем, что метод выведет все значения x от 1 до 5. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым. Например, в моем случае я получил следующий консольный вывод (он может в каждом конкретном случае различаться):

```
Поток 1: 1
Поток 5: 1
Поток 4: 1
Поток 2: 1
Поток 3: 1
Поток 1: 6
Поток 5: 7
Поток 3: 7
Поток 2: 7
Поток 4: 9
Поток 1: 11
Поток 4: 11
Поток 2: 11
Поток 3: 14
Поток 5: 11
Поток 1: 16
Поток 2: 16
Поток 3: 16
Поток 5: 18
Поток 4: 16
Поток 1: 21
Поток 5: 21
```

Поток 3: 21
Поток 2: 21
Поток 4: 21

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово lock. Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. Остальный потоки помещаются в очередь ожидания и ждут, пока текущий поток не освободит данный блок кода. В итоге с помощью lock мы можем переделать предыдущий пример следующим образом:

```
int x = 0;
object locker = new();  // объект-заглушка
// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";
    myThread.Start();
}
void Print()
{
    lock (locker)
    {
        x = 1;
        for (int i = 1; i < 6; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
        }
    }
}
```

Для блокировки с ключевым словом lock используется объект-заглушка, в данном случае это переменная locker. Обычно это переменная типа object. И когда выполнение доходит до оператора lock, объект locker блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток. После окончания работы блока кода, объект locker освобождается и становится доступным для других потоков.

В этом случае консольный вывод будет более упорядоченным

Поток 1: 1
Поток 1: 2
Поток 1: 3
Поток 1: 4
Поток 1: 5
Поток 5: 1
Поток 5: 2
Поток 5: 3
Поток 5: 4
Поток 5: 5

Поток 3: 1
Поток 3: 2
Поток 3: 3
Поток 3: 4
Поток 3: 5
Поток 2: 1
Поток 2: 2
Поток 2: 3
Поток 2: 4
Поток 2: 5
Поток 4: 1
Поток 4: 2
Поток 4: 3
Поток 4: 4
Поток 4: 5

30.Асинхронные функции. Await и async.

Нередко программа выполняет такие операции, которые могут занять продолжительное время, например, обращение к сетевым ресурсам, чтение-запись файлов, обращение к базе данных и т.д. Такие операции могут серьезно нагрузить приложение. Особенно это актуально в графических (десктопных или мобильных) приложениях, где продолжительные операции могут блокировать интерфейс пользователя и негативно повлиять на желание пользователя работать с программой, или в веб-приложениях, которые должны быть готовы обслуживать тысячи запросов в секунду. В синхронном приложении при выполнении продолжительных операций в основном потоке этот поток просто бы блокировался на время выполнения операции. И чтобы продолжительные операции не блокировали общую работу приложения, в C# можно задействовать асинхронность.
Асинхронность позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы и при этом более экономно использовать потоки. Асинхронные методы выполняются в отдельных потоках. Однако при выполнении продолжительной операции поток асинхронного метода возвратится в пул потоков и будет использоваться для других задач. А когда продолжительная операция завершит свое выполнение, для асинхронного метода опять выделяется поток из пула потоков, и асинхронный метод продолжает свою работу.
Ключевыми для работы с асинхронными вызовами в C# являются два оператора: async и await, цель которых - упростить написание асинхронного кода. Они используются вместе для создания асинхронного метода.
Асинхронный метод обладает следующими признаками:

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

Асинхронный метод, как и обычный, может использовать любое количество параметров или не использовать их вообще. Однако асинхронный метод не может определять параметры с модификаторами out, ref и in.
Также стоит отметить, что слово async, которое указывается в определении метода, НЕ делает автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений await.
Рассмотрим простейший пример определения и вызова асинхронного метода:

```
await PrintAsync();   // вызов асинхронного метода
Console.WriteLine("Некоторые действия в методе Main");
void Print()
{
    Thread.Sleep(3000);     // имитация продолжительной работы
    Console.WriteLine("Hello METANIT.COM");
}
// определение асинхронного метода
async Task PrintAsync()
{
    Console.WriteLine("Начало метода PrintAsync"); // выполняется синхронно
    await Task.Run(() => Print());                 // выполняется асинхронно
    Console.WriteLine("Конец метода PrintAsync");
}
```

Здесь прежде всего определен обычный метод Print, который просто выводит некоторую строку на консоль. Для имитации долгой работы в нем используется задержка на 3 секунд с помощью метода Thread.Sleep(). То есть условно Print - это некоторый метод, который выполняет некоторую продолжительную операцию. В реальном приложении это могло бы быть обращение к базе данных или чтение-запись файлов, но для упрощения понимания он просто выводит строку на консоль.
Также здесь определен асинхронный метод PrintAsync(). Асинхронным он является потому, что имеет в определении перед возвращаемым типом модификатор async, его возвращаемым типом является Task, и в теле метода определено выражение await. Стоит отметить, что явным образом метод PrintAsync не возвращает никакого объекта Task, однако поскольку в теле метода применяется выражение await, то в качестве возвращаемого типа можно использовать тип Task.
Оператор await предваряет выполнение задачи, которая будет выполняться асинхронно. В данном случае подобная операция представляет выполнение метода Print:

```
await Task.Run(()=>Print());
```

По негласным правилам в названии асинхроннных методов принято использовать суффикс Async - PrintAsync(), хотя в принципе это необязательно делать.
И затем в программе (в данном случае в методе Main) вызывается этот асинхронный метод.

```
await PrintAsync();   // вызов асинхронного метода
```

Посмотрим, какой у программы будет консольный вывод:

```
Начало метода PrintAsync
Hello METANIT.COM
Конец метода PrintAsync
Некоторые действия в методе Main
```

Разберем поэтапно, что здесь происходит:

Запускается программа, а точнее метод Main, в котором вызывается асинхронный метод PrintAsync.
Метод PrintAsync начинает выполняться синхронно вплоть до выражения await.
Console.WriteLine("Начало метода PrintAsync"); // выполняется синхронно
Выражение await запускает асинхронную задачу Task.Run(()=>Print())
Пока выполняется асинхронная задача Task.Run(()=>Print()) (а она может выполняться довольно продожительное время), выполнение кода возвращается в вызывающий метод - то есть в метод Main.
Когда асинхронная задача завершила свое выполнение (в случае выше - вывела строку через три секунды), продолжает работу асинхронный метод PrintAsync, который вызвал асинхронную задачу.
После завершения метода PrintAsync продолжает работу метод Main.

Асинхронный метод Main

Стоит учитывать, что оператор await можно применять только в методе, который имеет модификатор async. И если мы в методе Main используем оператор await, то метод Main тоже должен быть определен как асинхронный.

Задержка асинхронной операции и Task.Delay

В асинхронных методах для остановки метода на некоторое время можно применять метод Task.Delay(). В качестве параметра он принимает количество миллисекунд в виде значения int, либо объект TimeSpan, который задает время задержки.

```
await Task.Delay(3000);     // имитация продолжительной работы
```

Причем метод Task.Delay сам по себе представляет асинхронную операцию, поэтому к нему применяется оператор await.

Преимущества асинхронности

Выше приведенные примеры являются упрощением, и вряд ли их можно считать показательным. Рассмотрим другой пример:

```
PrintName("Tom");
PrintName("Bob");
PrintName("Sam");
void PrintName(string name)
{
    Thread.Sleep(3000);     // имитация продолжительной работы
    Console.WriteLine(name);
}
```

Данный код является синхронным и выполняет последовательно три вызова метода PrintName. Поскольку для имитации продолжительной работы в методе установлена задержка на три секунды, то общее выполнение программы займет не менее 9 секунд. Так как каждый последующий вызов PrintName будет ждать пока завершится предыдущий.

Изменим в программе синхронный метод PrintName на асинхронный:

```
await PrintNameAsync("Tom");
await PrintNameAsync("Bob");
await PrintNameAsync("Sam");
// определение асинхронного метода
async Task PrintNameAsync(string name)
{
    await Task.Delay(3000);     // имитация продолжительной работы
    Console.WriteLine(name);
```

```
}
```
Вместо метода PrintName теперь вызывается три раза PrintNameAsync. Для имитации продолжительной работы в методе установлена задержка на 3 секунды с помощью вызова Task.Delay(3000). И поскольку при вызове каждого метода применяется оператор await, который останавливает выполнение до завершения асинхронного метода, то общее выполнение программы опять же займет не менее 9 секунд. Тем не менее теперь выполнение асинхронных операций не блокирует основной поток.

Теперь оптимизируем программу:

```
var tomTask = PrintNameAsync("Tom");
var bobTask = PrintNameAsync("Bob");
var samTask = PrintNameAsync("Sam");
await tomTask;
await bobTask;
await samTask;
// определение асинхронного метода
async Task PrintNameAsync(string name)
{
    await Task.Delay(3000);     // имитация продолжительной работы
    Console.WriteLine(name);
}
```

В данном случае задачи фактически запускаются при определении. А оператор await применяется лишь тогда, когда нам нужно дождаться завершения асинхронных операций - то есть в конце программы. И в этом случае общее выполнение программы займет не менее 3 секунд, но гораздо меньше 9 секунд.

Определение асинхронного лямбда-выражения

Асинхронную операцию можно определить не только с помощью отдельного метода, но и с помощью лямбда-выражения:

```
// асинхронное лямбда-выражение
Func<string, Task> printer = async (message) =>
{
    await Task.Delay(1000);
    Console.WriteLine(message);
};

await printer("Hello World");
await printer("Hello METANIT.COM");
```

31.Расширяющие методы. Вложенные типы.

Расширяющие методы

Методы расширения (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем, поскольку у нас нет доступа к исходному коду. Либо если мы не можем использовать стандартный механизм наследования, например, если классы определенны с модификатором sealed.

Например, нам надо добавить для типа string новый метод:

```
string s = "Привет мир";
char c = 'и';
int i = s.CharCount(c);
Console.WriteLine(i);
public static class StringExtension
{
    public static int CharCount(this string str, char c)
    {
        int counter = 0;
        for (int i = 0; i < str.Length; i++)
        {
            if (str[i] == c)
                counter++;
        }
        return counter;
    }
}
```

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

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

Затем у всех строк мы можем вызвать данный метод:

```
int i = s.CharCount(c);
```

Причем нам уже не надо указывать первый параметр. Значения для остальных параметров передаются в обычном порядке.

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

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

Вложенные типы

Вложенные типы (Nested Types) — это типы объявленные внутри области видимости другого типа:

```
public class TopLevel
{
   public class Nested { } // Вложенный класс
   public enum Color { Red, Blue, Tan } // Вложенное перечисление
}
```

Вложенные типы имеют ряд особенностей:

- ✓ Вложенные типы имеют доступ к частным (private) членам родительского типа, а также ко всему, к чему имеет доступ сам родительский тип.
- ✓ Вложенный тип можно объявлять с любым модификатором доступа, а не только public и internal.
- ✓ По умолчанию вложенные типы имеют доступ private, а не internal.
- ✓ Для доступа к вложенным типам из-вне родительского типа, необходимо указывать имя родительского типа (как для доступа к статичным членам).
- ✓ TopLevel.Color color = TopLevel.Color.Red;
- ✓ Все типы могут быть вложенными, а вот родительскими могут выступать только классы и структуры.

32.Регулярные выражения: основные понятия.

Классы StringBuilder и String предоставляют достаточную функциональность для работы со строками. Однако .NET предлагает еще один мощный инструмент - регулярные выражения. Регулярные выражения представляют эффективный и гибкий метод по обработке больших текстов, позволяя в то же время существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками.
Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен System.Text.RegularExpressions. А центральным классом при работе с регулярными выражениями является класс Regex. Например, у нас есть некоторый текст и нам надо найти в нем все словоформы какого-нибудь слова. С классом Regex это сделать очень просто:

```
using System.Text.RegularExpressions;
string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";
Regex regex = new Regex(@"туп(\w*)");
MatchCollection matches = regex.Matches(s);
if (matches.Count > 0)
{
   foreach (Match match in matches)
      Console.WriteLine(match.Value);
}
else
{
   Console.WriteLine("Совпадений не найдено");
}
```

Здесь мы находим в искомой строке все словоформы слова "туп". В конструктор объекта Regex передается регулярное выражение для поиска. Далее мы разберем некоторые элементы синтаксиса регулярных выражений, а пока достаточно знать, что выражение туп(\w*) обозначает, найти все слова, которые имеют корень "туп" и после которого может стоять различное количество символов. Выражение \w означает алфавитно-цифровой символ, а звездочка после выражения указывает на неопределенное их количество - их может быть один, два, три или вообще не быть.
Метод Matches класса Regex принимает строку, к которой надо применить регулярные выражения, и возвращает коллекцию найденных совпадений.
Каждый элемент такой коллекции представляет объект Match. Его свойство Value возвращает найденное совпадение.

Параметр RegexOptions

Класс Regex имеет ряд конструкторов, позволяющих выполнить начальную инициализацию объекта. Две версии конструкторов в качестве одного из параметров принимают перечисление RegexOptions. Некоторые из значений, принимаемых данным перечислением:

- ✓ Compiled: при установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение
- ✓ CultureInvariant: при установке этого значения будут игнорироваться региональные различия
- ✓ IgnoreCase: при установке этого значения будет игнорироваться регистр
- ✓ IgnorePatternWhitespace: удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака #

- ✓ Multiline: указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы "^" и "$" совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста
- ✓ RightToLeft: приписывает читать строку справа налево
- ✓ Singleline: при данном режиме символ "." соответствует любому символу, в том числе последовательности "\n", которая осуществляет переход на следующую строку

Например:
Regex regex = new Regex(@"туп(\w*)", RegexOptions.IgnoreCase);
При необходимости можно установить несколько параметров:
Regex regex = new Regex(@"туп(\w*)", RegexOptions.Compiled | RegexOptions.IgnoreCase);
Синтаксис регулярных выражений
Рассмотрим вкратце некоторые элементы синтаксиса регулярных выражений:

- ✓ ^: соответствие должно начинаться в начале строки (например, выражение @"^пр\w*" соответствует слову "привет" в строке "привет мир")
- ✓ $: конец строки (например, выражение @"\w*ир$" соответствует слову "мир" в строке "привет мир", так как часть "ир" находится в самом конце)
- ✓ .: знак точки определяет любой одиночный символ (например, выражение "м.р" соответствует слову "мир" или "мор")
- ✓ *: предыдущий символ повторяется 0 и более раз
- ✓ +: предыдущий символ повторяется 1 и более раз
- ✓ ?: предыдущий символ повторяется 0 или 1 раз
- ✓ \s: соответствует любому пробельному символу
- ✓ \S: соответствует любому символу, не являющемуся пробелом
- ✓ \w: соответствует любому алфавитно-цифровому символу
- ✓ \W: соответствует любому не алфавитно-цифровому символу
- ✓ \d: соответствует любой десятичной цифре
- ✓ \D : соответствует любому символу, не являющемуся десятичной цифрой

Мы можем не только задать поиск по определенным типам символов - пробелы, цифры, но и задать конкретные символы, которые должны входить в регулярное выражение. Например, перепишем пример с номером телефона и явно укажем, какие символы там должны быть:
string s = "456-435-2318";
Regex regex = new Regex("[0-9]{3}-[0-9]{3}-[0-9]{4}");
В квадратных скобках задается диапазон символов, которые должны в данном месте встречаться. В итоге данный и предыдущий шаблоны телефонного номера будут эквивалентны.
Также можно задать диапазон для алфавитных символов: Regex regex = new Regex("[a-v]{5}"); - данное выражение будет соответствовать любому сочетанию пяти символов, в котором все символы находятся в диапазоне от a до v.
Можно также указать отдельные значения: Regex regex = new Regex(@"[2]*-[0-9]{3}-\d{4}");. Это выражение будет соответствовать, например, такому номеру телефона "222-222-2222" (так как первые числа двойки)
С помощью операции | можно задать альтернативные символы, например:
Regex regex = new Regex(@"(2|3){3}-[0-9]{3}-\d{4}");
То есть первые три цифры могут содержать только двойки или тройки. Такой шаблон будет соответствовать, например, строкам "222-222-2222" и "323-435-2318". А вот строка

"235-435-2318" уже не подпадает под шаблон, так как одной из трех первых цифр является цифра 5.
Итак, у нас такие символы, как *, + и ряд других используются в качестве специальных символов. И возникает вопрос, а что делать, если нам надо найти, строки, где содержится точка, звездочка или какой-то другой специальный символ? В этом случае нам надо просто экранировать эти символы слешем:

```
Regex regex = new Regex(@"(2|3){3}\.[0-9]{3}\.\d{4}");
// этому выражению будет соответствовать строка "222.222.2222"
```

Проверка на соответствие строки формату

Нередко возникает задача проверить корректность данных, введенных пользователем. Это может быть проверка электронного адреса, номера телефона, Класс Regex предоставляет статический метод IsMatch, который позволяет проверить входную строку с шаблоном на соответствие:

```
using System.Text.RegularExpressions;
string pattern = @"^(?("")(""[^""]+?""@)|(([0-9a-z]((\.(?!\.))|[-
!#\$%&'\*\+/=\?\^`\{\}\|~\w])*)(?<=[0-9a-z])@))" +
            @"(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-z][-\w]*[0-9a-z]*\.)+[a-z0-9]{2,17}))$";
var data = new string[]
{
   "tom@gmail.com",
   "+12345678999",
   "bob@yahoo.com",
   "+13435465566",
   "sam@yandex.ru",
   "+43743989393"
};

Console.WriteLine("Email List");
for(int i = 0; i < data.Length; i++)
{
   if (Regex.IsMatch(data[i], pattern, RegexOptions.IgnoreCase))
   {
      Console.WriteLine(data[i]);
   }
}
```

Переменная pattern задает регулярное выражение для проверки адреса электронной почты.
Далее в цикле мы проходим по массиву строк и определяем, какие строки соответствуют этому шаблону, то есть представляют валидный адрес электронной почты. Для проверки соответствия строки шаблону используется метод IsMatch: Regex.IsMatch(data[i], pattern, RegexOptions.IgnoreCase). Последний параметр указывает, что регистр можно игнорировать. И если строка соответствует шаблону, то метод возвращает true.

Замена и метод Replace

Класс Regex имеет метод Replace, который позволяет заменить строку, соответствующую регулярному выражению, другой строкой:

```
string text = "Мама  мыла  раму. ";
string pattern = @"\s+";
string target = " ";
```

```
Regex regex = new Regex(pattern);
string result = regex.Replace(text, target);
Console.WriteLine(result);
```
Данная версия метода Replace принимает два параметра: строку с текстом, где надо выполнить замену, и сама строка замены. Так как в качестве шаблона выбрано выражение "\s+ (то есть наличие одного и более пробелов), метод Replace проходит по всему тексту и заменяет несколько подряд идущих пробелов ординарными.

33.Анонимные методы. Лямбда-выражения. Неименованные функции. Замыкания.

Анонимные методы

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

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

```
delegate(параметры)
{
    // инструкции
}
```
Например:
```
MessageHandler handler = delegate (string mes)
{
    Console.WriteLine(mes);
};
handler("hello world!");
delegate void MessageHandler(string message);
```
Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата, как в данном случае переменная handler представляет анонимный метод. И через эту переменную делегата можно вызвать данный анонимный метод.

Другой пример анонимных методов - передача в качестве аргумента для параметра, который представляет делегат:
```
ShowMessage("hello!", delegate (string mes)
{
    Console.WriteLine(mes);
});
static void ShowMessage(string message, MessageHandler handler)
{
    handler(message);
}
delegate void MessageHandler(string message);
```
Если анонимный метод использует параметры, то они должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе опустить параметры:
```
MessageHandler handler = delegate
{
    Console.WriteLine("анонимный метод");
```

```
};
handler("hello world!");    // анонимный метод
delegate void MessageHandler(string message);
```
То есть если анонимный метод содержит параметры, они обязательно должны соответствовать параметрам делегата. Либо анонимный метод вообще может не содержать никаких параметров, тогда он соответствует любому делегату, который имеет тот же тип возвращаемого значения.
При этом параметры анонимного метода не могут быть опущены, если один или несколько параметров определены с модификатором out.
Также, как и обычные методы, анонимные могут возвращать результат:
```
Operation operation = delegate (int x, int y)
{
   return x + y;
};
int result = operation(4, 5);
Console.WriteLine(result);       // 9
delegate int Operation(int x, int y);
```
При этом анонимный метод имеет доступ ко всем переменным, определенным во внешнем коде:
```
int z = 8;
Operation operation = delegate (int x, int y)
{
   return x + y + z;
};
int result = operation(4, 5);
Console.WriteLine(result);       // 17
delegate int Operation(int x, int y);
```
В каких ситуациях используются анонимные методы? Когда нам надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется. В частности, их можно использовать для обработки событий, которые будут рассмотрены далее.

Лямбда-выражения

Лямбда-выражения представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы.
Лямбда-выражения имеют следующий синтаксис: слева от лямбда-оператора => определяется список параметров, а справа блок выражений, использующий эти параметры:
```
 (список_параметров) => выражение
```
С точки зрения типа данных лямбда-выражение представляет делегат. Например, определим простейшее лямбда-выражение:
```
Message hello = () => Console.WriteLine("Hello");
hello();       // Hello
delegate void Message();
```
В данном случае переменная hello представляет делегат Message - то есть некоторое действие, которое ничего не возвращает и не принимает никаких параметров. В качестве значения этой переменной присваивается лямбда-выражение. Это лямбда-выражение должно соответствовать делегату Message - оно тоже не принимает никаких параметров,

поэтому слева от лямбда-оператора идут пустые скобки. А справа от лямбда-оператора идет выполняемое выражение - Console.WriteLine("Hello")
Затем в программе можно вызывать эту переменную как метод.
Если лямбда-выражение содержит несколько действий, то они помещаются в фигурные скобки:

```
Message hello = () =>
{
   Console.Write("Hello ");
   Console.WriteLine("World");
};
hello();      // Hello World
```

Выше мы определили переменную hello, которая представляет делегат Message. Но начиная с версии C# 10 мы можем применять неявную типизацию (определение переменной с помощью оператора var) при определении лямбда-выражения:

```
var hello = () => Console.WriteLine("Hello");
hello();      // Hello
```

Но какой тип в данном случае представляет переменная hello? При неявной типизации компилятор сам пытается сопоставить лямбда-выражение на основе его опеределения с каким-нибудь делегатом. Например, выше определенное лямбда-выражение hello по умолчанию компилятор будет рассматривать как переменную встроенного делегата Action, который не принимает никаких параметров и ничего не возвращает.

Параметры лямбды

При определении списка параметров мы можем не указывать для них тип данных:

```
Operation sum = (x, y) => Console.WriteLine($"{x} + {y} = {x + y}");
sum(1, 2);      // 1 + 2 = 3
sum(22, 14);    // 22 + 14 = 36
delegate void Operation(int x, int y);
```

В данном случае компилятор видит, что лямбда-выражение sum представляет тип Operation, а значит оба параметра лямбды представляют тип int. Поэтому никак проблем не возникнет.
Однако если мы применяем неявную типизацию, то у компилятора могут возникнуть трудности, чтобы вывести тип делегата для лямбда-выражения, например, в следующем случае

```
var sum = (x, y) => Console.WriteLine($"{x} + {y} = {x + y}");   // ! Ошибка
```

В этом случае можно указать тип параметров

```
var sum = (int x, int y) => Console.WriteLine($"{x} + {y} = {x + y}");
sum(1, 2);      // 1 + 2 = 3
sum(22, 14);    // 22 + 14 = 36
```

Если лямбда имеет один параметр, для которого не требуется указывать тип данных, то скобки можно опустить:

```
PrintHandler print = message => Console.WriteLine(message);
print("Hello");        // Hello
print("Welcome");      // Welcome
delegate void PrintHandler(string message);
```

Возвращение результата

Лямбда-выражение может возвращать результат. Возвращаемый результат можно указать после лямбда-оператора:

```
var sum = (int x, int y) => x + y;
```

```
int sumResult = sum(4, 5);                // 9
Console.WriteLine(sumResult);             // 9
Operation multiply = (x, y) => x * y;
int multiplyResult = multiply(4, 5);      // 20
Console.WriteLine(multiplyResult);        // 20
delegate int Operation(int x, int y);
```

Если лямбда-выражение содержит несколько выражение, тогда нужно использовать оператор return, как в обычных методах:

```
var subtract = (int x, int y) =>
{
   if (x > y) return x - y;
   else return y - x;
};
int result1 = subtract(10, 6);  // 4
Console.WriteLine(result1);     // 4
int result2 = subtract(-10, 6);  // 16
Console.WriteLine(result2);     // 16
```

Добавление и удаление действий в лямбда-выражении

Поскольку лямбда-выражение представляет делегат, то как и в делегат, в переменную, которая представляет лямбда-выражение можно добавлять методы и другие лямбды:

```
var hello = () => Console.WriteLine("METANIT.COM");
var message = () => Console.Write("Hello ");
message += () => Console.WriteLine("World"); // добавляем анонимное лямбда-выражение
message += hello;   // добавляем лямбда-выражение из переменной hello
message += Print;   // добавляем метод
message();
Console.WriteLine("--------------"); // для разделения вывода
message -= Print;   // удаляем метод
message -= hello;   // удаляем лямбда-выражение из переменной hello
message?.Invoke();  // на случай, если в message больше нет действий
void Print() => Console.WriteLine("Welcome to C#");
```

```
Hello World
METANIT.COM
Welcome to C#
--------------
Hello World
```

Лямбда-выражение как аргумент метода

Как и делегаты, лямбда-выражения можно передавать параметрам метода, которые представляют делегат:

```
int[] integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// найдем сумму чисел больше 5
int result1 = Sum(integers, x => x > 5);
Console.WriteLine(result1); // 30
// найдем сумму четных чисел
int result2 = Sum(integers, x => x % 2 == 0);
Console.WriteLine(result2);  //20
int Sum(int[] numbers, IsEqual func)
{
```

```
    int result = 0;
    foreach (int i in numbers)
    {
        if (func(i))
            result += i;
    }
    return result;
}
delegate bool IsEqual(int x);
```

Метод Sum принимает в качестве параметра массив чисел и делегат IsEqual и возвращает сумму чисел массива в виде объекта int. В цикле проходим по всем числам и складываем их. Причем складываем только те числа, для которых делегат IsEqual func возвращает true. То есть делегат IsEqual здесь фактически задает условие, которому должны соответствовать значения массива. Но на момент написания метода Sum нам неизвестно, что это за условие.

При вызове метода Sum ему передается массив и лямбда-выражение:

```
int result1 = Sum(integers, x => x > 5);
```

То есть параметр x здесь будет представлять число, которое передается в делегат:

```
if (func(i))
```

А выражение x > 5 представляет условие, которому должно соответствовать число. Если число соответствует этому условию, то лямбда-выражение возвращает true, а переданное число складывается с другими числами.

Неименованные функции(это и есть анонимные методы и лямбды)

Замыкания

Замыкание (closure) представляет объект функции, который запоминает свое лексическое окружение даже в том случае, когда она выполняется вне своей области видимости. Технически замыкание включает три компонента:

- ✓ внешняя функция, которая определяет некоторую область видимости и в которой определены некоторые переменные и параметры - лексическое окружение
- ✓ переменные и параметры (лексическое окружение), которые определены во внешней функции
- ✓ вложенная функция, которая использует переменные и параметры внешней функции

В языке C# реализовать замыкания можно разными способами - с помощью локальных функций и лямбда-выражений.

Рассмотрим создание замыканий через локальные функции:

```
var fn = Outer();   // fn = Inner, так как метод Outer возвращает функцию Inner
// вызываем внутреннюю функцию Inner
fn();   // 6
fn();   // 7
fn();   // 8
Action Outer()  // метод или внешняя функция
{
    int x = 5;  // лексическое окружение - локальная переменная
    void Inner()    // локальная функция
    {
        x++;        // операции с лексическим окружением
        Console.WriteLine(x);
```

```
    }
    return Inner;   // возвращаем локальную функцию
}
```
Здесь метод Outer в качестве возвращаемого типа имеет тип Action, то есть метод возвратить функцию, которая не принимает параметров и имеет тип void.
```
Action Outer()
```
Внутри метода Outer определена переменная x - это и есть лексическое окружение для внутренней функции:
```
int x = 5;
```
Также внутри метода Outer определена внутренняя функция - локальная функция Inner, которая обращается к своему лексическому окружению - переменной x - увеличивает ее значение на единицу и выводит на консоль.
Эта локальная функция возвращается методом Outer:
```
return Inner;
```
В программе вызываем метод Outer и получаем в переменную fn локальную функцию Inner:
```
var fn = Outer();
```
Переменная fn и представляет собой замыкание, то есть объединяет две вещи: функцию и окружение, в котором функция была создана. И несмотря на то, что мы получили локальную функцию и можем ее вызывать вне ее метода, в котором она определена, тем не менее она запомнила свое лексическое окружение и может к нему обращаться и изменять, что мы увидим по консольному выводу:
```
fn();   // 6
fn();   // 7
fn();   // 8
```
Реализация с помощью лямбда-выражений

С помощью лямбд можно сократить определение замыкания:
```
var outerFn = () =>
{
    int x = 10;
    var innerFn = () => Console.WriteLine(++x);
    return innerFn;
};
var fn = outerFn();   // fn = innerFn, так как outerFn возвращает innerFn
// вызываем innerFn
fn();   // 11
fn();   // 12
fn();   // 13
```
Применение параметров

Кроме внешних переменных к лексическому окружению также относятся параметры окружающего метода. Рассмотрим использование параметров:
```
var fn = Multiply(5);
Console.WriteLine(fn(5));   // 25
Console.WriteLine(fn(6));   // 30
Console.WriteLine(fn(7));   // 35
Operation Multiply(int n)
{
    int Inner(int m)
```

```
    {
        return n * m;
    }
    return Inner;
}
delegate int Operation(int n);
```

Здесь внешняя функция - метод Multiply возвращает функцию, которая принимает число int и возвращает число int. Для этого определен делегат Operation, который будет представлять возвращаемый тип:

```
delegate int Operation(int n);
```

Хотя также можно было бы использовать встроенный делегат Func<int, int>.

Вызов метода Multiply() возвращает локальную функцию, которая соответствует сигнатуре делегата Operation:

```
int Inner(int m)
{
    return n * m;
}
```

Эта функция запоминает окружение, в котором она была создана, в частности, значение параметра n. Кроме того, сама принимает параметр и возвращает произведение параметров n и m.

В итоге при вызове метода Multiply определяется переменная fn, которая получает локальную функцию Inner и ее лексическое окружение - значение параметра n:

```
var fn = Multiply(5);
```

В данном случае параметр n равен 5.

При вызове локальной функции, например, в случае:

```
Console.WriteLine(fn(6));   // 30
```

Число 6 передается для параметра m локальной функции, которая возвращает произведение n и m, то есть 5 * 6 = 30.

Также можно было бы сократить весь этот код с помощью лямбд:

```
var multiply = (int n) => (int m) => n * m;
var fn = multiply(5);
Console.WriteLine(fn(5));   // 25
Console.WriteLine(fn(6));   // 30
Console.WriteLine(fn(7));   // 35
```

34.Анонимные типы. Неявно типизированные локальные переменные и массивы.

Анонимные типы

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

```
var user = new { Name = "Tom", Age = 34 };
Console.WriteLine(user.Name);
```

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

При этом во время компиляции компилятор сам будет создавать для него имя типа и использовать это имя при обращении к объекту.

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

Если в программе используются несколько объектов анонимных типов с одинаковым набором свойств, то для них компилятор создаст одно определение анонимного типа:

```
var user = new { Name = "Tom", Age = 34 };
var student = new { Name = "Alice", Age = 21 };
var manager = new { Name = "Bob", Age = 26, Company = "Microsoft" };
Console.WriteLine(user.GetType().Name); // <>f__AnonymousType0'2
Console.WriteLine(student.GetType().Name); // <>f__AnonymousType0'2
Console.WriteLine(manager.GetType().Name); // <>f__AnonymousType1'3
```

Здесь user и student будут иметь одно и то же определение анонимного типа. Однако подобные объекты нельзя преобразовать к какому-нибудь другому типу, например, классу, даже если он имеет подобный набор свойств.

Следует учитывать, что свойства анонимного объекта доступны для установки только в инициализаторе. Вне инициализатора присвоить им значение мы не можем. Поэтому, например, в следующем случае мы столкнемся с ошибкой:

```
var student = new { Name = "Alice", Age = 21 };
student.Age = 32;   // ! Ошибка
```

Кроме использованной выше формы инициализации, когда мы присваиваем свойствам некоторые значения, также можно использовать инициализаторы с проекцией (projection initializers), когда мы можем передать в инициализатор некоторые идиентификаторы, имена которых будут использоваться как названия свойств:

```
Person tom = new Person("Tom");
int age = 34;
var student = new { tom.Name, age }; // инициализатор с проекцией
Console.WriteLine(student.Name);
Console.WriteLine(student.age);
class Person
{
    public string Name { get; set; }
    public Person(string name) => Name = name;
}
```

В данном случае определение анонимного объекта фактически будет идентично следующему:

```
var student = new { Name = tom.Name, age = age};
```

Названия свойств и переменных (Name и age) будут использоваться в качесте названий свойств объекта.
Также можно определять массивы объектов анонимных типов:

```
var people = new[]
{
   new {Name="Tom"},
   new {Name="Bob"}
};
foreach(var p in people)
{
   Console.WriteLine(p.Name);
}
```

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

Неявно типизированные локальные переменные и массивы

Язык C# — это язык со статической (строгой) типизацией. Тип переменной определяется один раз на этапе объявления переменной и не может изменяться в последствии.
Неявно типизированная переменная — это переменная, тип которой выводится компилятором, исходя из выражения, стоящего справа от переменной. Например,

```
var i = 1; //тип переменной int
var s = "string"; //тип переменной string
```

Чтобы убедиться, что переменным i и s действительно присвоен соответствующий их значениям тип, достаточно в Visual Studio навести курсор на ключевое слово var:
неявно типизированная переменная

Неявно типизированная переменная

Изменить тип переменной мы не можем. Например, следующий код приведет к ошибке:

```
var s = "string"; //тип переменной string
s = 2;
```

Ошибка CS0029 Не удается неявно преобразовать тип «int» в «string».
Таким образом, использование неявно типизированных переменных в C# это не более, чем синтаксическое удобство (хотя и несколько сомнительное), которое позволяет сократить количество ввода. Ни смены типа переменной в процессе выполнения приложения, ни каких-либо других преимуществ, в данном случае, мы не получаем.

Где могут использоваться неявно типизированные переменные

Использование неявно типизированных переменных имеет свои ограничения. Например, мы можем использовать неявно типизированные переменные в качестве локальных переменных в методах. Также, мы можем использовать их в циклах:

```
for (var i = 1; i < 10; i++)
{
   Console.WriteLine(i);
}
```

В данном случае, справа от var i стоит число 1, что позволяет компилятору вывести тип (int) и выполнить цикл. В цикле foreach также допустимо использование var:

```
var list = new List<string>();
list.Add("Привет");
list.Add("мир");
foreach (var item in list)
{
   Console.WriteLine(item);
}
```

Здесь в цикле тип переменной item определен, исходя из типа элементов в списке (string).

Где запрещено использование неявно типизированных переменных

Неявно типизированные переменны запрещено использовать:

1) в качестве возвращаемого методом результата:

```
public var Method()
{
}
```

2) в параметрах методов:

```
public void Method(var i)
{
}
```

3) в свойствах классов:

```
public class MyClass
{
   public var Name { get; set; }
}
```

Во всех приведенных выше случаях мы получим ошибку:

Контекстное ключевое слово «var» может использоваться только в объявлении локальной переменной или в скрипте.

4) в выражениях, где тип переменной невозможно определить

Например, в выражении ниже:

```
var n = null;
```

в итоге, получим ошибку:

Ошибка CS0815 Не удается присвоить неявно типизированной переменной.

35.Операторы преобразования. explicit, implicit.

Иногда объект определенного класса требуется использовать в выражении, включающем в себя данные других типов. В одних случаях для этой цели оказывается пригодной перегрузка одного или более операторов, а в других случаях — обыкновенное преобразование типа класса в целевой тип. Для подобных ситуаций в C# предусмотрена специальная разновидность операторного метода, называемая оператором преобразования. Такой оператор преобразует объект исходного класса в другой тип. Операторы преобразования помогают полностью интегрировать типы классов в среду программирования на C#, разрешая свободно пользоваться классами вместе с другими типами данных, при условии, что определен порядок преобразования в эти типы. Существуют две формы операторов преобразования: явная и неявная. Ниже они представлены в общем виде:
public static explicit operator целевой_тип(исходный_тип v) {return значение;}
public static implicit operator целевой_тип(исходный_тип v) {return значение; }
где целевой_тип обозначает тот тип, в который выполняется преобразование; исходный_тип — тот тип, который преобразуется; значение — конкретное значение, приобретаемое классом после преобразования. Операторы преобразования возвращают данные, имеющие целевой_тип, причем указывать другие возвращаемые типы данных не разрешается.
Если оператор преобразования указан в неявной форме (implicit), то преобразование вызывается автоматически, например, в том случае, когда объект используется в выражении вместе со значением целевого типа. Если же оператор преобразования указан в явной форме (explicit), то преобразование вызывается в том случае, когда выполняется приведение типов. Для одних и тех же исходных и целевых типов данных нельзя указывать оператор преобразования одновременно в явной и неявной форме.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1
{
   class UserInfo
   {
      public string Name, Family;
      public byte Age;

      public UserInfo(string Name, string Family, byte Age)
      {
         this.Name = Name;
         this.Family = Family;
         this.Age = Age;
      }

      // Явное преобразование типа UserInfo к string
      public static explicit operator string(UserInfo obj)
      {
         return "Информация о пользователе: " + obj.Name + " " + obj.Family + " ("
+obj.Age+" лет)";
```

```
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      UserInfo ui1 = new UserInfo(Name: "Alexandr", Family: "Erohin", Age: 26);
      string s = (string)ui1;

      Console.WriteLine(s);

      Console.ReadLine();
    }
  }
}
```

Перегрузка операторов преобразования C#

Оператор неявного преобразования применяется автоматически в следующих случаях: когда в выражении требуется преобразование типов; методу передается объект; осуществляется присваивание и производится явное приведение к целевому типу. С другой стороны, можно создать оператор явного преобразования, вызываемый только тогда, когда производится явное приведение типов. В таком случае оператор явного преобразования не вызывается автоматически, как показано в примере.

На операторы преобразования накладывается ряд следующих ограничений:

- ✓ Исходный или целевой тип преобразования должен относиться к классу, для которого объявлено данное преобразование. В частности, нельзя переопределить преобразование в тип int, если оно первоначально указано как преобразование в тип double.
- ✓ Нельзя указывать преобразование в класс object или же из этого класса.
- ✓ Для одних и тех же исходных и целевых типов данных нельзя указывать одновременно явное и неявное преобразование.
- ✓ Нельзя указывать преобразование базового класса в производный класс.
- ✓ Нельзя указывать преобразование в интерфейс или же из него.

Помимо указанных выше ограничений, имеется ряд рекомендаций, которыми обычно руководствуются при выборе операторов явного или неявного преобразования. Несмотря на все преимущества неявных преобразований, к ним следует прибегать только в тех случаях, когда преобразованию не свойственны ошибки. Во избежание подобных ошибок неявные преобразования должны быть организованы только в том случае, если удовлетворяются следующие условия. Во-первых, информация не теряется, например, в результате усечения, переполнения или потери знака. И во-вторых, преобразование не приводит к исключительной ситуации. Если же неявное преобразование не удовлетворяет этим двум условиям, то следует выбрать явное преобразование.

36.Интегрированный язык запросов LINQ.

LINQ (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс IEnumerable (например, стандартные коллекции, массивы), набор данных DataSet, документ XML. Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

Существует несколько разновидностей LINQ:

- ✓ LINQ to Objects: применяется для работы с массивами и коллекциями
- ✓ LINQ to Entities: используется при обращении к базам данных через технологию Entity Framework
- ✓ LINQ to XML: применяется при работе с файлами XML
- ✓ LINQ to DataSet: применяется при работе с объектом DataSet
- ✓ Parallel LINQ (PLINQ): используется для выполнения параллельных запросов

Основная часть функциональности LINQ сосредоточена в пространстве имен System.LINQ. В проектах под .NET 6 данное пространство имен подключается по умолчанию.

В чем же удобство LINQ? Посмотрим на простейшем примере. Выберем из массива строки, которые начинаются на определенную букву, например, букву "T", и отсортируем полученный список:

```
string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };
// создаем новый список для результатов
var selectedPeople = new List<string>();
// проходим по массиву
foreach (string person in people)
{
    // если строка начинается на букву T, добавляем в список
    if (person.ToUpper().StartsWith("T"))
        selectedPeople.Add(person);
}
// сортируем список
selectedPeople.Sort();
foreach (string person in selectedPeople)
    Console.WriteLine(person);
```

Для отфильтрованных строк создается специальный список. Затем в цикле проходим по всем элементам массива и, если они соответствуют условию (начинаются на букву T), то добавляем их в этот список. Затем сортируем список по возрастанию. И в конце элементы полученного списка выводим на консоль:

```
Tim
Tom
Tomas
```

Хотя подобный подход вполне работает, однако LINQ позволяет значительно сократить код с помощью интуитивно понятного и краткого синтаксиса.

Для работы с колекциями можно использовать два способа:

- ✓ Операторы запросов LINQ
- ✓ Методы расширений LINQ

Операторы запросов LINQ

Операторы запросов LINQ в каком-то роде частично напоминают синтаксис запросов SQL, поэтому если вы работали когда-нибудь с sql-запросами, то будет проще понять

общую концепцию. Итак, изменим предыдущий пример, применив операторы запросов LINQ:

```
string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };
// создаем новый список для результатов
var selectedPeople = from p in people // передаем каждый элемент из people в переменную p
                     where p.ToUpper().StartsWith("T") //фильтрация по критерию
                     orderby p  // упорядочиваем по возрастанию
                     select p; // выбираем объект в создаваемую коллекцию

foreach (string person in selectedPeople)
   Console.WriteLine(person);
```

Прежде всего, как мы видим, код стал меньше и проще, а результат будет тем же. В принципе все выражение можно было бы записать в одну строку:

```
var selectedPeople = from p in people where p.ToUpper().StartsWith("T") orderby p  select p;
```

Но для более понятной логической разбивки я поместил каждое отдельное подвыражение на отдельной строке.

Простейшее определение запроса LINQ выглядит следующим образом:

```
from переменная in набор_объектов
select переменная;
```

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

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

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

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

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

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

Методы расширения LINQ

Кроме стандартного синтаксиса from .. in .. select для создания запроса LINQ мы можем применять специальные методы расширения, которые определены для интерфейса IEnumerable. Как правило, эти методы реализуют ту же функциональность, что и операторы LINQ типа where или orderby.

Например:

```
string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };
var selectedPeople = people.Where(p => p.ToUpper().StartsWith("T")).OrderBy(p => p);
foreach (string person in selectedPeople)
   Console.WriteLine(person);
```

Запрос people.Where(p=>p.ToUpper().StartsWith("T")).OrderBy(p => p) будет аналогичен предыдущему. Он состоит из цепочки методов Where и OrderBy. В качестве аргумента эти методы принимают делегат или лямбда-выражение.

И хотя ряд действий мы можем реализовать как с помощью операторов запросов LINQ, так и с помощью методов расширений LINQ, но не каждый метод расширения имеет аналог среди операторов LINQ. И в этом случае можно сочетать оба подхода. Например, используем стандартный синтаксис linq и метод расширения Count(), который возвращает количество элементов в выборке:

```
int number = (from p in people where p.ToUpper().StartsWith("T") select p).Count();
Console.WriteLine(number); // 3
```

Список используемых методов расширения LINQ

- ✓ Select: определяет проекцию выбранных значений
- ✓ Where: определяет фильтр выборки
- ✓ OrderBy: упорядочивает элементы по возрастанию
- ✓ OrderByDescending: упорядочивает элементы по убыванию
- ✓ ThenBy: задает дополнительные критерии для упорядочивания элементов возрастанию
- ✓ ThenByDescending: задает дополнительные критерии для упорядочивания элементов по убыванию
- ✓ Join: соединяет две коллекции по определенному признаку
- ✓ Aggregate: применяет к элементам последовательности агрегатную функцию, которая сводит их к одному объекту
- ✓ GroupBy: группирует элементы по ключу
- ✓ ToLookup: группирует элементы по ключу, при этом все элементы добавляются в словарь
- ✓ GroupJoin: выполняет одновременно соединение коллекций и группировку элементов по ключу
- ✓ Reverse: располагает элементы в обратном порядке
- ✓ All: определяет, все ли элементы коллекции удовлятворяют определенному условию
- ✓ Any: определяет, удовлетворяет хотя бы один элемент коллекции определенному условию
- ✓ Contains: определяет, содержит ли коллекция определенный элемент
- ✓ Distinct: удаляет дублирующиеся элементы из коллекции
- ✓ Except: возвращает разность двух коллекцию, то есть те элементы, которые создаются только в одной коллекции
- ✓ Union: объединяет две однородные коллекции
- ✓ Intersect: возвращает пересечение двух коллекций, то есть те элементы, которые встречаются в обоих коллекциях
- ✓ Count: подсчитывает количество элементов коллекции, которые удовлетворяют определенному условию
- ✓ Sum: подсчитывает сумму числовых значений в коллекции
- ✓ Average: подсчитывает среднее значение числовых значений в коллекции
- ✓ Min: находит минимальное значение
- ✓ Max: находит максимальное значение
- ✓ Take: выбирает определенное количество элементов
- ✓ Skip: пропускает определенное количество элементов

- ✓ TakeWhile: возвращает цепочку элементов последовательности, до тех пор, пока условие истинно
- ✓ SkipWhile: пропускает элементы в последовательности, пока они удовлетворяют заданному условию, и затем возвращает оставшиеся элементы
- ✓ Concat: объединяет две коллекции
- ✓ Zip: объединяет две коллекции в соответствии с определенным условием
- ✓ First: выбирает первый элемент коллекции
- ✓ FirstOrDefault: выбирает первый элемент коллекции или возвращает значение по умолчанию
- ✓ Single: выбирает единственный элемент коллекции, если коллекция содержит больше или меньше одного элемента, то генерируется исключение
- ✓ SingleOrDefault: выбирает единственный элемент коллекции. Если коллекция пуста, возвращает значение по умолчанию. Если в коллекции больше одного элемента, генерирует исключение
- ✓ ElementAt: выбирает элемент последовательности по определенному индексу
- ✓ ElementAtOrDefault: выбирает элемент коллекции по определенному индексу или возвращает значение по умолчанию, если индекс вне допустимого диапазона
- ✓ Last: выбирает последний элемент коллекции
- ✓ LastOrDefault: выбирает последний элемент коллекции или возвращает значение по умолчанию