МУХАМЕДИЕВА Д.Т., ВАРЛАМОВА Л.П,
БАХРОМОВ С.А.

ПРОГРАММНАЯ ИНЖЕНЕРИЯ C# – ОСНОВЫ ПРОГРАММИРОВАНИЯ

УЧЕБНОЕ ПОСОБИЕ

МУХАМЕДИЕВА Д.Т., ВАРЛАМОВА Л.П,
БАХРОМОВ С.А.

ПРОГРАММНАЯ ИНЖЕНЕРИЯ C# – ОСНОВЫ ПРОГРАММИРОВАНИЯ

УЧЕБНОЕ ПОСОБИЕ

МИНИСТЕРСТВО ВЫСШЕГО И СРЕДНЕГО СПЕЦИАЛЬНОГО ОБРАЗОВАНИЯ РЕСПУБЛИКИ УЗБЕКИСТАН

НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ УЗБЕКИСТАНА ИМЕНИ МИРЗО УЛУГБЕКА

Мухамедиева Д.Т., Варламова Л.П.,
Бахромов С.А.

Учебное пособие

по курсу

«ПРОГРАММНАЯ ИНЖЕНЕРИЯ»

C# - основы программирования

Ташкент-2023 г.

Учебное пособие по курсу “Программная инженерия” составлено на основе образцовой и рабочей программ для студентов факультета “Прикладной математики и интеллектуальных систем” с целью обучения использованию языков программирования, разработке программного обеспечения и систем управления базами данных. Данный курс читается студентам на протяжении трех семестров. В первом семестре рассматриваются вопросы управления доступом к классам, рекурсии, инкапсуляции и наследования, создания интерфейсов и коллекций на базе языка программирования C#.

“Dasturiy injiniring” kursi uchun o‘quv qo‘llanma “Amaliy matematika va intellektual tizimlar” fakulteti talabalari uchun dasturlash tillaridan foydalanish, dasturiy ta’minot ishlab chiqish va ma’lumotlar bazasini boshqarish tizimlarini o‘rgatish maqsadida namunaviy va ishchi dastur asosida tuzilgan. . Ushbu kurs talabalarga uch semestr davomida o'qitiladi. Birinchi semestr sinflarga kirishni boshqarish, rekursiya, inkapsulyatsiya va meros, C# dasturlash tili asosida interfeyslar va kolleksiyalarni yaratish masalalarini qamrab oladi.

Авторы:

Д.Т.Мухамедиева - профессор Национального исследовательского университета «Ташкентский институт инженеров ирригации и механизации сельского хозяйства», д.т.н.

Л.П.Варламова - профессор кафедры вычислительной математики и информационных систем Национального университета Узбекистана имени Мирзо Улугбека, д.т.н.

С.А.Бахромов - доцент кафедры вычислительной математики и информационных систем Национального университета Узбекистана имени Мирзо Улугбека, к.т.н.

Рецензенты:

Матякубов А.С. Зав. кафедрой “Прикладная математика и компьютерный анализ” Национального университета Узбекистана

Якубов М.С. Профессор кафедры “Информационные технологии” Ташкентского университета информационных технологий

Учебное пособие рекомендовано к изданию на основании приказа Министерства высшего и среднего специального образования Республики Узбекистан от 09 сентября 2022 года №302.

Регистрационный номер 302-0382.

Содержание

Введение

Программный инженеринг - одна из важнейших и базовых отраслей современных информационных и коммуникационных технологий. Он играет важную роль в решении современных и важных проблем в этих областях науки.

Основная цель предмета «Программный инженеринг» - развить у студентов знания и навыки в области программной инженерии и описать примеры их практического применения.

Этот курс представляет собой широкое введение в процесс внедрения программной инженерии.

Предмет «Программная инженерия» 5330100 - Математическое и программное обеспечение информационных систем неразрывно связан с дисциплинами «Языки программирования», «Математика», «Структуры данных» и необходимо знать некоторые разделы этих дисциплин. Знания, полученные по предметам «Программная инженерия», «Безопасность компьютерных систем», «Базы данных и информационные системы», «Современные технологии в системах баз данных», «Интеллектуальные системы», «Информационные системы управления», «Аудит информационных систем», «Программное обеспечение для бизнеса. разработка программных приложений», «Разработка мобильных программных приложений ».

В проектах программирования эффективно используются различные предметные области программной инженерии.

Согласно программе в данной дисциплине изучается множество модельных вопросов, что позволяет каждому бакалавру, глубоко изучившему предмет, эффективно использовать знания и навыки, полученные в практической работе, исследованиях, а также в системе образования.

Данное пособие обеспечивает формирование знаний и умений в соответствии с государственным образовательным стандартом 5330100 «Математическое и программное обеспечение информационных систем», способствует формированию мировоззрения и системного мышления, обучает специалистов в области применения программных систем.

1. Управление доступом к классу.

Класс, концепция поля, создание классов и ссылки на них, работа с модификациями

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

Классы и структуры — это, по сути, шаблоны, по которым можно создавать объекты. Каждый объект содержит данные и методы, манипулирующие этими данными.

Общая форма определения класса

При определении класса объявляются данные, которые он содержит, а также код, оперирующий этими данными. Если самые простые классы могут содержать только код или только данные, то большинство настоящих классов содержит и то и другое.

Вообще говоря, данные содержатся в членах данных, определяемых классом, а код — в функциях-членах. Следует сразу же подчеркнуть, что в C# предусмотрено несколько разновидностей членов данных и функций-членов:

Данные-члены

Данные-члены — это те члены, которые содержат данные класса — поля, константы, события. Данные-члены могут быть статическими (static). Член класса является членом экземпляра, если только он не объявлен явно как static. Давайте рассмотрим виды этих данных:

Поля (field)

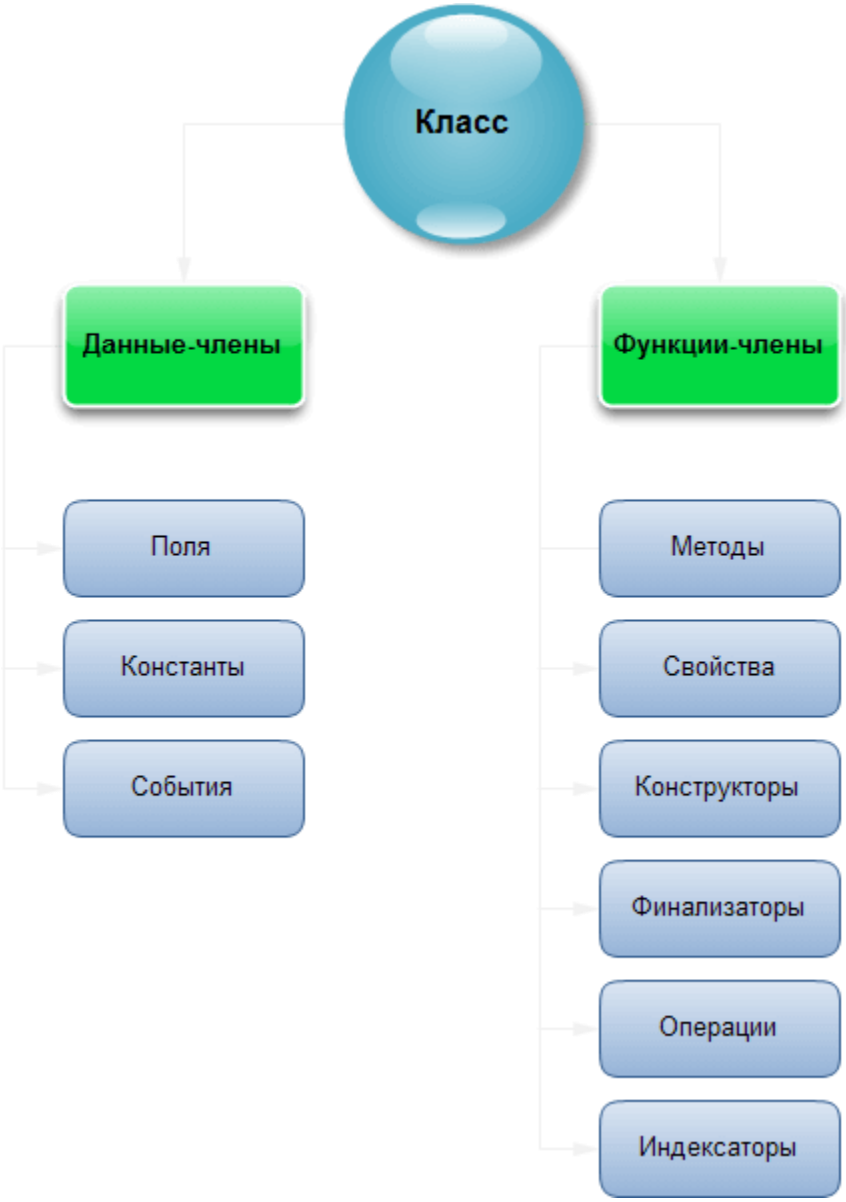
Это любые переменные, ассоциированные с классом.

Константы

Константы могут быть ассоциированы с классом тем же способом, что и переменные. Константа объявляется с помощью ключевого слова const. Если она объявлена как public, то в этом случае становится доступной извне класса.

События

Это члены класса, позволяющие объекту уведомлять вызывающий код о том, что случилось нечто достойное упоминания, например, изменение свойства класса либо некоторое взаимодействие с пользователем. Клиент может иметь код, известный как обработчик событий, реагирующий на них.



Функции-члены

Функции-члены — это члены, которые обеспечивают некоторую функциональность для манипулирования данными класса. Они включают методы, свойства, конструкторы, финализаторы, операции и индексаторы:

Методы (method)

Это функции, ассоциированные с определенным классом. Как и данные-члены, по умолчанию они являются членами экземпляра. Они могут быть объявлены статическими с помощью модификатора static.

Свойства (property)

Это наборы функций, которые могут быть доступны клиенту таким же способом, как общедоступные поля класса. В C# предусмотрен специальный синтаксис для реализации чтения и записи свойств для классов, поэтому писать собственные методы с именами, начинающимися на Set и Get, не понадобится. Поскольку не существует какого-то отдельного синтаксиса для свойств, который отличал бы их от нормальных функций, создается иллюзия объектов как реальных сущностей, предоставляемых клиентскому коду.

Конструкторы (constructor)

Это специальные функции, вызываемые автоматически при инициализации объекта. Их имена совпадают с именами классов, которым они принадлежат, и они не имеют типа возврата. Конструкторы полезны для инициализации полей класса.

Финализаторы (finalizer)

Вызываются, когда среда CLR определяет, что объект больше не нужен. Они имеют то же имя, что и класс, но с предшествующим символом тильды. Предсказать точно, когда будет вызван финализатор, невозможно.

Операции (operator)

Это простейшие действия вроде + или -. Когда вы складываете два целых числа, то, строго говоря, применяете операцию + к целым. Однако C# позволяет указать, как существующие операции будут работать с пользовательскими классами (так называемая перегрузка операции).

Индексаторы (indexer)

Позволяют индексировать объекты таким же способом, как массив или коллекцию.

Класс создается с помощью ключевого слова class. Ниже приведена общая форма определения простого класса, содержащая только переменные экземпляра и методы:

```
class имя_класса {
        // Объявление переменных экземпляра.
        доступ тип переменная1;
        доступ тип переменная2;
        //...
        доступ тип переменнаяN;

        // Объявление методов.
        доступ возвращаемый_тип метод1 (параметры) {
                // тело метода
        }
        доступ возвращаемый_тип метод2 (параметры) {
                // тело метода
        }
        //. . .
        доступ возвращаемый_тип методN(параметры) {
                // тело метода
        }
}
```

Обратите внимание на то, что перед каждым объявлением переменной и метода указывается доступ. Это спецификатор доступа, например public, определяющий порядок доступа к данному члену класса. Члены класса могут быть как закрытыми (private) в пределах класса, так открытыми (public), т.е. более доступными. Спецификатор доступа определяет тип разрешенного доступа. Указывать спецификатор доступа не обязательно, но если он

отсутствует, то объявляемый член считается закрытым в пределах класса. Члены с закрытым доступом могут использоваться только другими членами их класса.

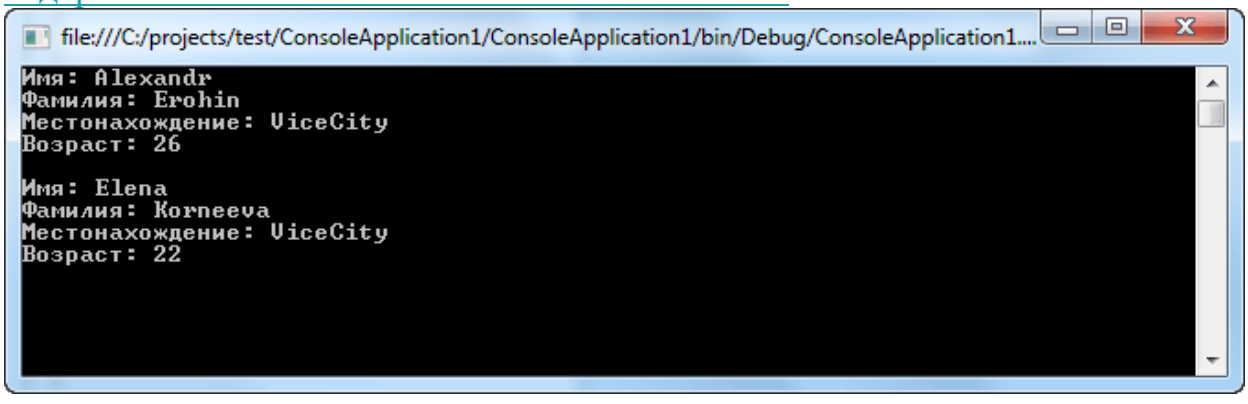
Давайте разберем пример создания класса, описывающего характеристики пользователя:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class UserInfo
    {
        // Поля класса
        public string Name, Family, Adress;
        public byte Age;

        // Метод, выводящий в консоль контактную информацию
        public void writeInConsoleInfo(string name, string family, string adress, byte age)
        {
            Console.WriteLine("Имя: {0}\nФамилия: {1}\nМестонахождение: {2}\nВозраст: {3}\n", name, family, adress, age);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Создаем объект типа UserInfo
            UserInfo myInfo = new UserInfo();

            myInfo.Name = "Alexandr";
            myInfo.Family = "Erohin";
            myInfo.Adress = "ViceCity";
            myInfo.Age = 26;

            // Создадим новый экземпляр класса UserInfo
            UserInfo myGirlFriendInfo = new UserInfo();

            myGirlFriendInfo.Name = "Elena";
            myGirlFriendInfo.Family = "Korneeva";
```

```
            myGirlFriendInfo.Adress = "ViceCity";
            myGirlFriendInfo.Age = 22;

            // Выведем информацию в консоль
            myInfo.writeInConsoleInfo(myInfo.Name, myInfo.Family,
myInfo.Adress, myInfo.Age);

myGirlFriendInfo.writeInConsoleInfo(myGirlFriendInfo.Name,myGirlFriendInfo.
Family,myGirlFriendInfo.Adress,myGirlFriendInfo.Age);

            Console.ReadLine();
        }
    }

}
```

В данном примере определяется новый пользовательский класс UserInfo, который содержит 4 поля и 1 метод, которые являются открытыми (т.е. содержат модификатор доступа public). В методе Main() создаются два экземпляра этого класса: myInfo и myGirlFriendInfo. Затем инициализируются поля данных экземпляров и вызывается метод writeInConsoleInfo().

Прежде чем двигаться дальше, рассмотрим следующий основополагающий принцип: у каждого объекта имеются свои копии переменных экземпляра, определенных в его классе. Следовательно, содержимое переменных в одном объекте может отличаться от их содержимого в другом объекте. Между обоими объектами не существует никакой связи, за исключением того факта, что они являются объектами одного и того же типа. Так, если имеются два объекта типа UserInfo, то у каждого из них своя копия переменных Name, Family, Age и Adress, а их содержимое в обоих объектах может отличаться:



Класс Object

C# --- Руководство по C# --- Класс Object

В C# предусмотрен специальный класс object, который неявно считается базовым классом для всех остальных классов и типов, включая и

типы значений. Иными словами, все остальные типы являются производными от object. Это, в частности, означает, что переменная ссылочного типа object может ссылаться на объект любого другого типа. Кроме того, переменная типа object может ссылаться на любой массив, поскольку в C# массивы реализуются как объекты. Формально имя object считается в C# еще одним обозначением класса System.Object, входящего в библиотеку классов для среды .NET Framework.

Практическое значение этого в том, что помимо методов и свойств, которые вы определяете, также появляется доступ к множеству общедоступных и защищенных методов-членов, которые определены в классе Object. Эти методы присутствуют во всех определяемых классах.

Методы System.Object

Ниже перечислены все методы данного класса:

ToString()

Метод ToString() возвращает символьную строку, содержащую описание того объекта, для которого он вызывается. Кроме того, метод ToString() автоматически вызывается при выводе содержимого объекта с помощью метода WriteLine(). Этот метод переопределяется во многих классах, что позволяет приспосабливать описание к конкретным типам объектов, создаваемых в этих классах.

Применяйте этот метод, когда нужно получить представление о содержимом объекта — возможно, в целях отладки. Он предлагает очень ограниченные средства форматирования данных. Например, даты в принципе могут быть отображены в огромном разнообразии форматов, но DateTime.ToString() не оставляет никакого выбора в этом отношении. Если нужно более сложное строковое представление, которое, например, принимает во внимание установленные предпочтения или местные стандарты, то понадобится реализовать интерфейс IFormattable.

GetHashCode()

Этот метод используется, когда объект помещается в структуру данных, известную как карта (map), которая также называется хеш-таблицей или словарем. Применяется классами, которые манипулируют этими структурами, чтобы определить, куда именно в структуру должен быть помещен объект. Если вы намерены использовать свой класс как ключ словаря, то должны переопределить GetHashCode(). Существуют достаточно строгие требования относительно того, как нужно реализовывать перегрузку.

Хеш-код можно использовать в любом алгоритме, где хеширование применяется в качестве средства доступа к хранимым объектам. Следует, однако, иметь в виду, что стандартная реализация метода GetHashCode() не пригодна на все случаи применения.

Equals() и ReferenceEquals()

По умолчанию метод Equals (object) определяет, ссылается ли вызывающий объект на тот же самый объект, что и объект, указываемый в качестве аргумента этого метода, т.е. он определяет, являются ли обе ссылки

одинаковыми. Метод Equals (object) возвращает логическое значение true, если сравниваемые объекты одинаковы, в противном случае — логическое значение false. Он может быть также переопределен в создаваемых классах. Это позволяет выяснить, что же означает равенство объектов для создаваемого класса. Например, метод Equals (object) можно определить таким образом, чтобы в нем сравнивалось содержимое двух объектов.

Как несложно догадаться, учитывая существование трех различных методов сравнения объектов, среда .NET использует довольно сложную схему определения эквивалентности объектов. Следует учитывать и использовать тонкие различия между этими тремя методами и операцией сравнения ==. Кроме того, также существуют ограничения, регламентирующие, как следует переопределять виртуальную версию Equals() с одним параметром, если вы решитесь на это — поскольку некоторые базовые классы из пространства имен System.Collections вызывают этот метод и ожидают от него определенного поведения.

Finalize()

Назначение этого метода в C# примерно соответствует деструкторам C++, и он вызывается при сборке мусора для очистки ресурсов, занятых ссылочным объектом. Реализация Finalize() из Object на самом деле ничего не делает и игнорируется сборщиком мусора. Обычно переопределять Finalize() необходимо, если объект владеет неуправляемыми ресурсами, которые нужно освободить при его уничтожении. Сборщик мусора не может сделать это напрямую, потому что он знает только об управляемых ресурсах, поэтому полагается на финализацию, определенную вами.

GetType()

Этот метод возвращает экземпляр класса, унаследованный от System.Type. Этот объект может предоставить большой объем информации о классе, членом которого является ваш объект, включая базовый тип, методы, свойства и т.п. System.Type также представляет собой стартовую точку технологии рефлексии .NET.

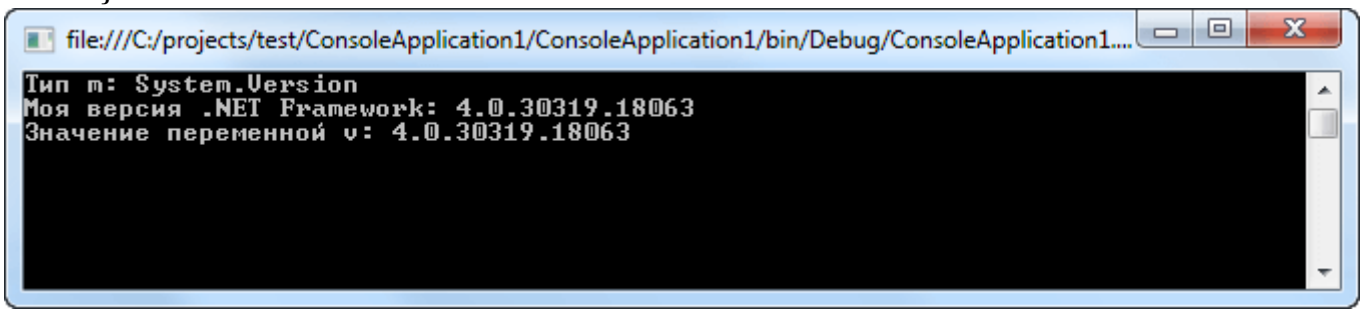
Clone()

Этот метод создает копию объекта и возвращает ссылку на эту копию (а в случае типа значения — ссылку на упаковку). Отметим, что при этом выполняется неглубокое копирование, т.е. копируются все типы значений в классе. Если же класс включает в себя члены ссылочных типов, то копируются только ссылки, а не объекты, на которые они указывают. Этот метод является защищенным, а потому не может вызываться для копирования внешних объектов. К тому же он не виртуальный, а потому переопределять его реализацию нельзя.

Давайте рассмотрим применение некоторых из этих методов на конкретном примере:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;

namespace ConsoleApplication1
{
  class Program
   {
      static void Main(string[] args)
      {
         var m = Environment.Version;
               Console.WriteLine("Тип m: "+m.GetType());
         string s = m.ToString();
         Console.WriteLine("Моя версия .NET Framework: " + s);
         Version v = (Version)m.Clone();
         Console.WriteLine("Значение переменной v: "+v);
         Console.ReadLine();
      }
   }
}
```

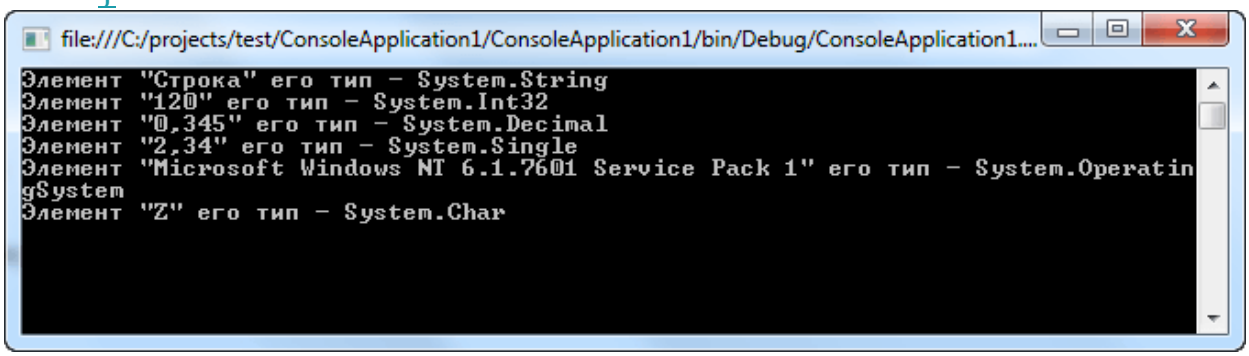


Класс object как универсальный тип данных

Если object является базовым классом для всех остальных типов и упаковка значений простых типов происходит автоматически, то класс object можно вполне использовать в качестве "универсального" типа данных. Для примера рассмотрим программу, в которой сначала создается массив типа object, элементам которого затем присваиваются значения различных типов данных:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
  class Program
   {
      static void Main(string[] args)
      {
         var myOS = Environment.OSVersion;
```

```
            object[] myArr = { "Строка", 120, 0.345m, 2.34f, myOS, 'Z' };

            foreach (object obj in myArr)
                Console.WriteLine("Элемент \"{0}\" его тип - {1}",obj,obj.GetType());

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



Как показывает данный пример, по ссылке на объект класса object можно обращаться к данным любого типа, поскольку в переменной ссылочного типа object допускается хранить ссылку на данные всех остальных типов. Следовательно, в массиве типа object из рассматриваемого здесь примера можно сохранить данные практически любого типа. В развитие этой идеи можно было бы, например, без особого труда создать класс стека со ссылками на объекты класса object. Это позволило бы хранить в стеке данные любого типа.

Несмотря на то что универсальный характер класса object может быть довольно эффективно использован в некоторых ситуациях, было бы ошибкой думать, что с помощью этого класса стоит пытаться обойти строго соблюдаемый в C# контроль типов. Вообще говоря, целое значение следует хранить в переменной типа int, строку — в переменной ссылочного типа string и т.д.

А самое главное, что начиная с версии 2.0 для программирования на C# стали доступными подлинно обобщенные типы данных — обобщения. Внедрение обобщений позволило без труда определять классы и алгоритмы, автоматически обрабатывающие данные разных типов, соблюдая типовую безопасность. Благодаря обобщениям отпала необходимость пользоваться классом object как универсальным типом данных при создании нового кода. Универсальный характер этого класса лучше теперь оставить для применения в особых случаях.

Создание объектов

C# --- Руководство по C# --- Создание объектов

Для объявления объекта произвольного типа используется следующая конструкция:

<тип класса> имя переменной = new <тип класса>();

Например:

InfoUser myinfo = new InfoUser();

Эта строка объявления выполняет три функции. Во-первых, объявляется переменная myinfo, относящаяся к типу класса InfoUser. Сама эта переменная не является объектом, а лишь переменной, которая может ссылаться на объект. Во-вторых, создается конкретная, физическая, копия объекта. Это делается с помощью оператора new. И наконец, переменной myinfo присваивается ссылка на данный объект. Таким образом, после выполнения анализируемой строки объявленная переменная myinfo ссылается на объект типа InfoUser.

Оператор new динамически (т.е. во время выполнения) распределяет память для объекта и возвращает ссылку на него, которая затем сохраняется в переменной. Следовательно, в C# для объектов всех классов должна быть динамически распределена память.

То обстоятельство, что объекты классов доступны по ссылке, объясняет, почему классы называются ссылочными типами. Главное отличие типов значений от ссылочных типов заключается в том, что именно содержит переменная каждого из этих типов. Так, переменная типа значения содержит конкретное значение, а ссылочная переменная содержит не сам объект, а лишь ссылку на него.

Переменные ссылочного типа и присваивание

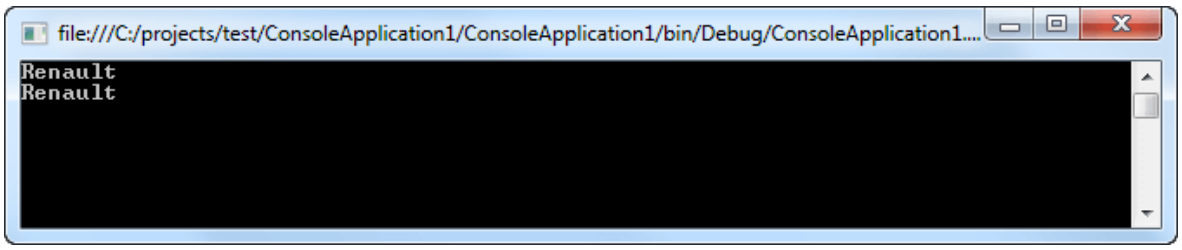
В операции присваивания переменные ссылочного типа действуют иначе, чем переменные типа значения, например типа int. Когда одна переменная типа значения присваивается другой, ситуация оказывается довольно простой. Переменная, находящаяся в левой части оператора присваивания, получает копию значения переменной, находящейся в правой части этого оператора.

Когда же одна переменная ссылки на объект присваивается другой, то ситуация несколько усложняется, поскольку такое присваивание приводит к тому, что переменная, находящаяся в левой части оператора присваивания, ссылается на тот же самый объект, на который ссылается переменная, находящаяся в правой части этого оператора. Сам же объект не копируется. В силу этого отличия присваивание переменных ссылочного типа может привести к несколько неожиданным результатам.

Когда переменная Car1 присваивается переменой Car2, то в конечном итоге переменная Car2 просто ссылается на тот же самый объект, что и переменная Car1. Следовательно, этим объектом можно оперировать с помощью переменной Car1 или Car2. Несмотря на то что обе переменные, Car1 и Car2, ссылаются на один и тот же объект, они никак иначе не связаны друг с другом.

В качестве примера рассмотрим следующий фрагмент кода:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1
{
    class autoCar
    {
        public string marka;
    }
    class Program
    {
        static void Main(string[] args)
        {
            autoCar Car1 = new autoCar();
            autoCar Car2 = Car1;
            Car1.marka = "Renault";
            Console.WriteLine(Car1.marka);
            Console.WriteLine(Car2.marka);
            Console.ReadLine();
        }
    }
}
```



Инициализаторы объектов

Инициализаторы объектов предоставляют способ создания объекта и инициализации его полей и свойств. Если используются инициализаторы объектов, то вместо обычного вызова конструктора класса указываются имена полей или свойств, инициализируемых первоначально задаваемым значением. Следовательно, синтаксис инициализатора объекта предоставляет альтернативу явному вызову конструктора класса. Синтаксис инициализатора объекта используется главным образом при создании анонимных типов в LINQ-выражениях. Но поскольку инициализаторы объектов можно, а иногда и нужно использовать в именованном классе, то ниже представлены основные положения об инициализации объектов.

Ниже приведена общая форма синтаксиса инициализации объектов:

new имя_класса {имя = выражение, имя = выражение, ...}

где имя обозначает имя поля или свойства, т.е. доступного члена класса, на который указывает имя_класса. А выражениеобозначает инициализирующее выражение, тип которого, конечно, должен соответствовать типу поля или свойства.

Инициализаторы объектов обычно не используются в именованных классах, хотя это вполне допустимо. Вообще, при обращении с именованными классами используется синтаксис вызова обычного конструктора.

Пример использования инициализаторов объекта:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class autoCar
    {
        public string marka;
        public short year;
    }

    class Program
    {
        static void Main(string[] args)
        {
```

```
            // используем инициализаторы
            autoCar myCar = new autoCar { marka = "Renault", year = 2004 };
            Console.ReadLine();
        }
    }
}
```

Методы

C# --- Руководство по C# --- Методы

Следует отметить, что официальная терминология C# делает различие между функциями и методами. Согласно этой терминологии, понятие "функция-член" включает не только методы, но также другие члены, не являющиеся данными, класса или структуры. Сюда входят индексаторы, операции, конструкторы, деструкторы, а также — возможно, несколько неожиданно — свойства. Они контрастируют с данными-членами: полями, константами и событиями.

Объявление методов

В C# определение метода состоит из любых модификаторов (таких как спецификация доступности), типа возвращаемого значения, за которым следует имя метода, затем список аргументов в круглых скобках и далее - тело метода в фигурных скобках:

```
[модификаторы] тип_возврата ИмяМетода([параметры])
{
// Тело метода
}
```

Каждый параметр состоит из имени типа параметра и имени, по которому к нему можно обратиться в теле метода. Вдобавок, если метод возвращает значение, то для указания точки выхода должен использоваться оператор возврата return вместе с возвращаемым значением.

Если метод не возвращает ничего, то в качестве типа возврата указывается **void**, поскольку вообще опустить тип возврата невозможно. Если же он не принимает аргументов, то все равно после имени метода должны присутствовать пустые круглые скобки. При этом включать в тело метода оператор возврата не обязательно — метод возвращает управление автоматически по достижении закрывающей фигурной скобки.

Возврат из метода и возврат значения

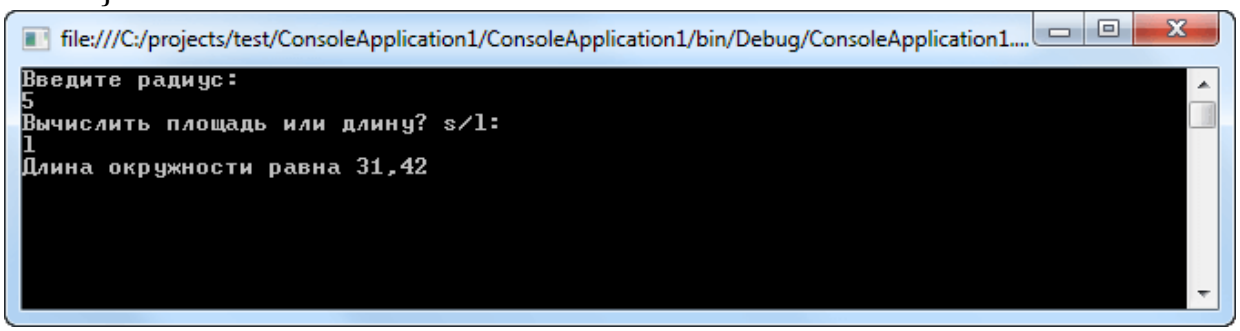
В целом, возврат из метода может произойти при двух условиях. Во-первых, когда встречается фигурная скобка, закрывающая тело метода. И во-вторых, когда выполняется оператор **return**. Имеются две формы оператора return: одна — для методов типа void (*возврат из метода*), т.е. тех методов, которые не возвращают значения, а другая — для методов, возвращающих конкретные значения (*возврат значения*).

Давайте рассмотрим пример:

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   class MyMathOperation
   {
      public double r;
      public string s;

      // Возвращает площадь круга
      public double sqrCircle()
      {
         return Math.PI * r * r;
      }

      // Возвращает длину окружности
      public double longCircle()
      {
         return 2 * Math.PI * r;
      }

      public void writeResult()
      {
         Console.WriteLine("Вычислить площадь или длину? s/l:");
         s = Console.ReadLine();
         s = s.ToLower();
         if (s == "s")
         {
            Console.WriteLine("Площадь круга равна
{0:#.###}",sqrCircle());
            return;
         }
         else if (s == "l")
         {
            Console.WriteLine("Длина окружности равна
{0:#.##}",longCircle());
            return;
         }
         else
         {
            Console.WriteLine("Вы ввели не тот символ");
         }
      }
```

```
        }

        class Program
        {
            static void Main(string[] args)
            {
                Console.WriteLine("Введите радиус: ");
                string radius = Console.ReadLine();

                MyMathOperation newOperation = new MyMathOperation { r = double.Parse(radius) };
                newOperation.writeResult();

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



Использование параметров

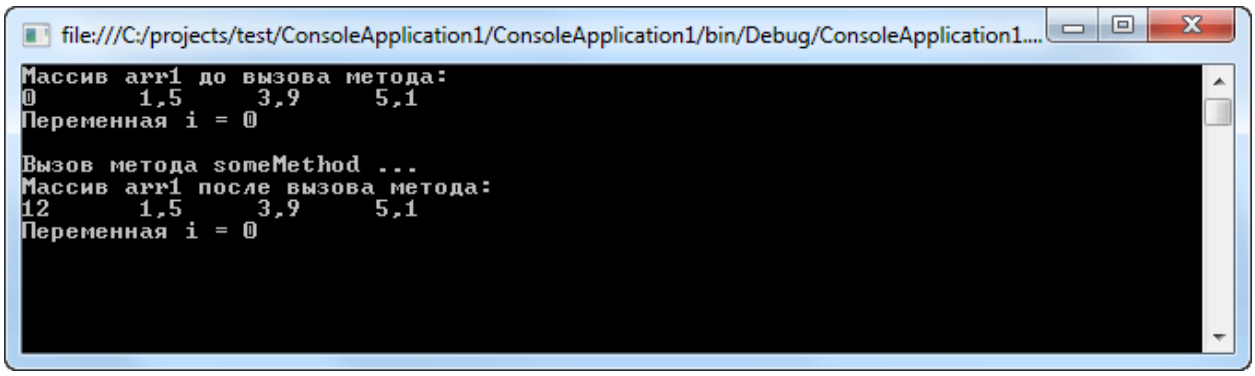
При вызове метода ему можно передать одно или несколько значений. Значение, передаваемое методу, называется *аргументом*. А переменная, получающая аргумент, называется формальным параметром, или просто *параметром*. Параметры объявляются в скобках после имени метода. Синтаксис объявления параметров такой же, как и у переменных. А областью действия параметров является тело метода. За исключением особых случаев передачи аргументов методу, параметры действуют так же, как и любые другие переменные.

В общем случае параметры могут передаваться методу либо по значению, либо по ссылке. Когда переменная передается по ссылке, вызываемый метод получает саму переменную, поэтому любые изменения, которым она подвергнется внутри метода, останутся в силе после его завершения. Но если переменная передается по значению, вызываемый метод получает копию этой переменной, а это значит, что все изменения в ней по завершении метода будут утеряны. Для сложных типов данных передача по ссылке более эффективна из-за большого объема данных, который приходится копировать при передаче по значению.

Давайте рассмотрим пример:

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class myClass
    {
        public void someMethod(double[] myArr, int i )
        {
            myArr[0] = 12.0;
            i = 12;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            double[] arr1 = { 0, 1.5, 3.9, 5.1 };
            int i = 0;
            Console.WriteLine("Массив arr1 до вызова метода: ");
            foreach (double d in arr1)
                Console.Write("{0}\t",d);
            Console.WriteLine("\nПеременная i = {0}\n",i);

            Console.WriteLine("Вызов метода someMethod ...");
            myClass ss = new myClass();
            ss.someMethod(arr1,i);
            Console.WriteLine("Массив arr1 после вызова метода:");
            foreach (double d in arr1)
                Console.Write("{0}\t",d);
            Console.WriteLine("\nПеременная i = {0}\n",i);

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



Обратите внимание, что значение i осталось неизменным, но измененные значения в myArr также изменились в исходном массиве arr1, так как массивы являются ссылочными типами.

Поведение строк также отличается. Дело в том, что строки являются неизменными (изменение значения строки приводит к созданию совершенно новой строки), поэтому строки не демонстрируют поведение, характерное для ссылочных типов. Любые изменения, проведенные в строке внутри метода, не влияют на исходную строку.

Конструкторы

C# --- Руководство по C# --- Конструкторы

Конструктор инициализирует объект при его создании. У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу. Но у конструкторов нет возвращаемого типа, указываемого явно. Ниже приведена общая форма конструктора:

доступ имя_класса(список_параметров) {
// тело конструктора
}

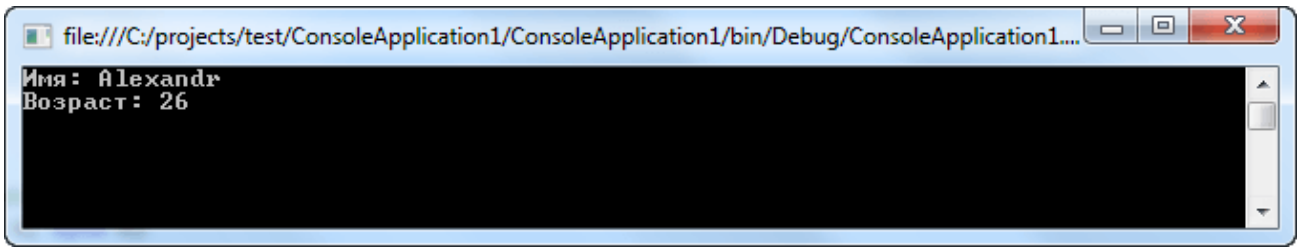
Как правило, конструктор используется для задания первоначальных значений переменных экземпляра, определенных в классе, или же для выполнения любых других установочных процедур, которые требуются для создания полностью сформированного объекта. Кроме того, *доступ* обычно представляет собой модификатор доступа типа public, поскольку конструкторы зачастую вызываются в классе. А *список_параметров* может быть как пустым, так и состоящим из одного или более указываемых параметров.

Каждый класс C# снабжается конструктором по умолчанию, который при необходимости может быть переопределен. По определению такой конструктор никогда не принимает аргументов. После размещения нового объекта в памяти конструктор по умолчанию гарантирует установку всех полей в соответствующие стандартные значения. Если вы не удовлетворены такими присваиваниями по умолчанию, можете переопределить конструктор по умолчанию в соответствии со своими нуждами.

Конструктор также может принимать один или несколько параметров. В конструктор параметры вводятся таким же образом, как и в метод. Для этого достаточно объявить их в скобках после имени конструктора.

Давайте рассмотрим применение конструкторов на примере:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1
{
   class MyClass
   {
      public string Name;
      public byte Age;
      // Создаем параметрический конструктор
      public MyClass(string s, byte b)
      {
         Name = s;
         Age = b;
      }
      public void reWrite()
      {
         Console.WriteLine("Имя: {0}\nВозраст: {1}", Name, Age);
      }
   }

   class Program
   {
      static void Main(string[] args)
      {
         MyClass ex = new MyClass("Alexandr", 26);
         ex.reWrite();

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



Как видите, в данном примере поля экземпляра класса ex инициализируются по умолчанию с помощью конструктора.

2. Использование параметров ref и out, возвращаемый объект из метода, необязательные аргументы.

Концепция и использование ref и out, создание метода и возврат объекта, необязательные аргументы

Методы могут как принимать, так и не принимать параметров, а также возвращать или не возвращать значения вызывающей стороне. Хотя определение метода в C# выглядит довольно понятно, существует несколько ключевых слов, с помощью которых можно управлять способом передачи аргументов интересующему методу:

Модификаторы параметров

Модификатор параметра	**Описание**
(отсутствует)	Если параметр не сопровождается модификатором, предполагается, что он должен передаваться по значению, т.е. вызываемый метод должен получать копию исходных данных
out	Выходные параметры должны присваиваться вызываемым методом (и, следовательно, передаваться по ссылке). Если параметрам out в вызываемом методе значения не присвоены, компилятор сообщит об ошибке
ref	Это значение первоначально присваивается вызывающим кодом и при желании может повторно присваиваться в вызываемом методе (поскольку данные также передаются по ссылке). Если параметрам ref в вызываемом методе значения не присвоены, компилятор никакой ошибки генерировать не будет
params	Этот модификатор позволяет передавать в виде одного логического параметра *переменное* количество аргументов. В каждом методе может присутствовать только один модификатор params и он должен обязательно указываться последним в списке параметров. В реальности необходимость в использовании модификатора params возникает не особо часто, однако он применяется во многих методах внутри библиотек базовых классов

Нередко требуется, чтобы метод оперировал теми аргументами, которые ему передаются. Характерным тому примером служит метод Swap(), осуществляющий перестановку значений своих аргументов. Но поскольку аргументы простых типов передаются по значению, то, используя выбираемый в C# по умолчанию механизм вызова по значению для передачи аргумента параметру, невозможно написать метод, меняющий местами значения двух его аргументов, например типа int. Это затруднение разрешает модификатор ref.

Как вам должно быть уже известно, значение возвращается из метода вызывающей части программы с помощью оператора return. Но метод может одновременно возвратить лишь одно значение. А что, если из метода требуется возвратить два или более фрагментов информации, например, целую и дробную части числового значения с плавающей точкой? Такой метод можно написать, используя модификатор out.

Давайте отдельно рассмотрим роль каждого из вышеуказанных ключевых слов.

Модификатор ref

Модификатор параметра ref принудительно организует вызов по ссылке, а не по значению. Этот модификатор указывается как при объявлении, так и при вызове метода.

Параметры, сопровождаемые таким модификатором, называются ссылочными и применяются, когда нужно позволить методу выполнять операции и обычно также изменять значения различных элементов данных, объявляемых в вызывающем коде (например, в процедуре сортировки или обмена). Обратите внимание на следующие отличия между ссылочными и выходными параметрами:

Выходные параметры

Это параметры, которые не нужно инициализировать перед передачей методу. Причина в том, что метод сам должен присваивать значения выходным параметрам перед выходом.

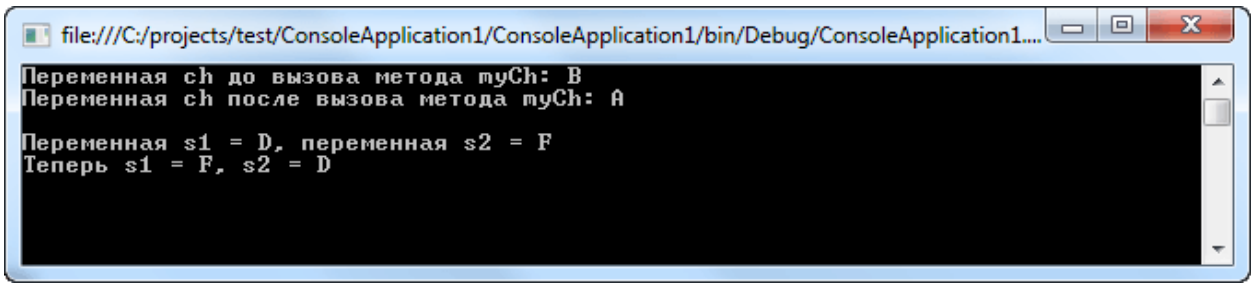
Ссылочные параметры

Эти параметры нужно обязательно инициализировать перед передачей методу. Причина в том, что они подразумевают передачу ссылки на уже существующую переменную. Если первоначальное значение ей не присвоено, это будет равнозначно выполнению операции над неинициализированной локальной переменной.

Давайте рассмотрим пример использования модификатора ref:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace ConsoleApplication1
{

   class Program
   {
      // Метод, изменяющий свой аргумент
      static void myCh(ref char c)
      {
         c = 'A';
      }

      // Метод меняющий местами аргументы
      static void Swap(ref char a, ref char b)
      {
         char c;
         c = a;
         a = b;
         b = c;
      }

      static void Main()
      {
         char ch = 'B', s1 = 'D', s2 = 'F';
         Console.WriteLine("Переменная ch до вызова метода myCh: {0}",ch);
         myCh(ref ch);
         Console.WriteLine("Переменная ch после вызова метода myCh: {0}", ch);

         Console.WriteLine("\nПеременная s1 = {0}, переменная s2 = {1}", s1, s2);
         Swap(ref s1, ref s2);
         Console.WriteLine("Теперь s1 = {0}, s2 = {1}",s1,s2);

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



В отношении модификатора ref необходимо иметь в виду следующее. Аргументу, передаваемому по ссылке с помощью этого модификатора, должно быть присвоено значение до вызова метода. Дело в том, что в методе, получающем такой аргумент в качестве параметра, предполагается, что параметр ссылается на действительное значение. Следовательно, при использовании модификатора ref в методе нельзя задать первоначальное значение аргумента.

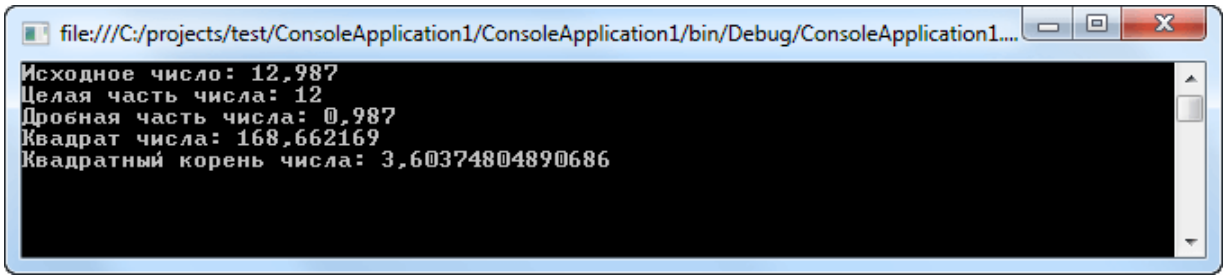
Модификатор out

Модификатор параметра out подобен модификатору ref, за одним исключением: он служит только для передачи значения за пределы метода. Поэтому переменной, используемой в качестве параметра out, не нужно (да и бесполезно) присваивать какое-то значение. Более того, в методе параметр out считается неинициализированным, т.е. предполагается, что у него отсутствует первоначальное значение. Это означает, что значение должно быть присвоено данному параметру в методе до его завершения. Следовательно, после вызова метода параметр out будет содержать некоторое значение.

Пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{

    class Program
    {
        // Метод возвращающий целую и дробную части
        // числа, квадрат и корень числа
        static int TrNumber(double d, out double dr, out double sqr, out double
sqrt)
```

```
    {
        int i = (int)d;
        dr = d - i;
        sqr = d * d;
        sqrt = Math.Sqrt(d);

        return i;
    }

    static void Main()
    {
        int i;
        double myDr, mySqr, mySqrt, myD = 12.987;
        i = TrNumber(myD, out myDr, out mySqr, out mySqrt);

        Console.WriteLine("Исходное число: {0}\nЦелая часть числа: {1}\nДробная часть числа: {2}\nКвадрат числа: {3}\nКвадратный корень числа: {4}",myD,i,myDr,mySqr,mySqrt);

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



Обратите внимание, что использование модификатора out в данном примере позволяет возвращать из метода сразу четыре значения.

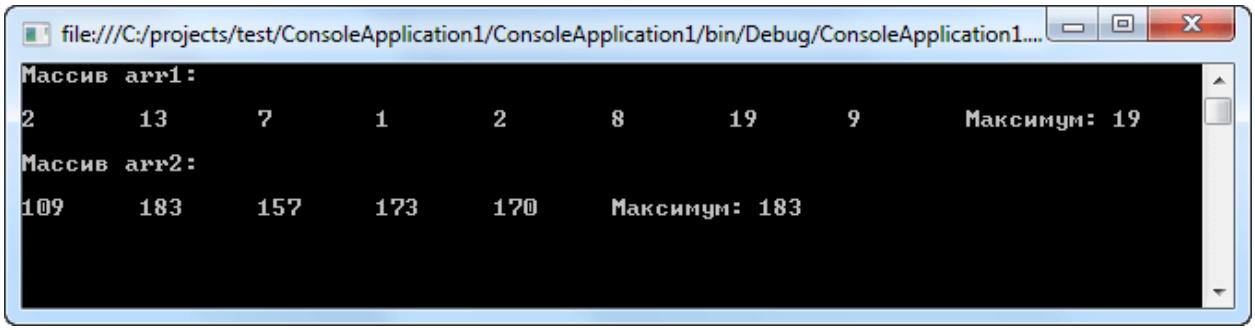
Модификатор params

В C# поддерживается использование массивов параметров за счет применения ключевого слова params. Ключевое слово params позволяет передавать методу переменное количество аргументов одного типа в виде единственного логического параметра. Аргументы, помеченные ключевым словом params, могут обрабатываться, если вызывающий код на их месте передает строго типизированный массив или разделенный запятыми список элементов.

Число элементов массива параметров будет равно числу аргументов, передаваемых методу. А для получения аргументов в программе организуется доступ к данному массиву:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{

    class Program
    {
        static void MaxArr(ref int i, params int[] arr)
        {
            // Изначально нужно обязательно выполнить проверку
            // на непустоту массива
            if (arr.Length == 0)
            {
                Console.WriteLine("Пустой массив!");
                i=0;
                return;
            }
            else
            {
                if (arr.Length == 1)
                {
                    i = arr[0];
                    return;
                }
            }

            i = arr[0];
            // Ищем максимум
            for (int j = 1; j < arr.Length; j++)
                if (arr[j] > i)
                    i = arr[j];
        }

        static void Main()
        {
            int result = 0;

            int[] arr1 = new int[8];
            int[] arr2 = new int[5];
            Random ran = new Random();
            // Инициализируем оба массива случайными числами
```

```
for (int i = 0; i < 8; i++)
    arr1[i] = ran.Next(1, 20);
for (int i = 0; i < 5; i++)
    arr2[i] = ran.Next(100, 200);

Console.WriteLine("Массив arr1: \n");
foreach (int i in arr1)
    Console.Write("{0}\t",i);
MaxArr(ref result, arr1);
Console.WriteLine("Максимум: {0}",result);

Console.WriteLine("\nМассив arr2: \n");
foreach (int i in arr2)
    Console.Write("{0}\t", i);
MaxArr(ref result, arr2);
```




```
Console.WriteLine("Максимум: {0}", result);

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

В тех случаях, когда у метода имеются обычные параметры, а также параметр переменной длины типа params, он должен быть указан последним в списке параметров данного метода. Но в любом случае параметр типа params должен быть единственным.

3. Рекурсия

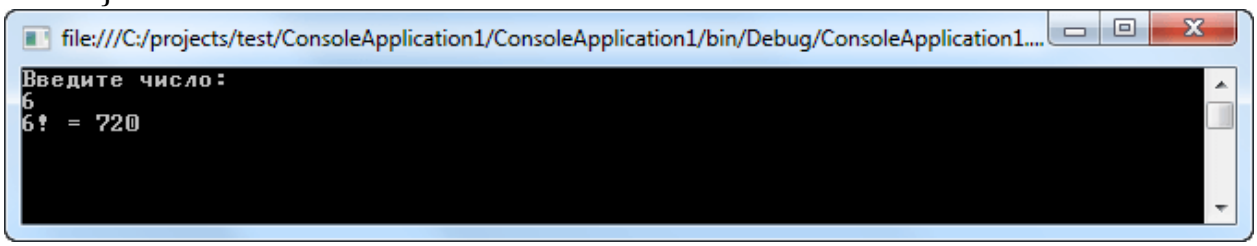
Понятие рекурсии, работа с рекурсивными методами

В C# допускается, чтобы метод вызывал самого себя. Этот процесс называется рекурсией, а метод, вызывающий самого себя, — рекурсивным. Вообще, рекурсия представляет собой процесс, в ходе которого нечто определяет само себя. В этом отношении она чем-то напоминает циклическое определение. Рекурсивный метод отличается главным образом тем, что он содержит оператор, в котором этот метод вызывает самого себя. Рекурсия является эффективным механизмом управления программой.

Классическим примером рекурсии служит вычисление факториала числа:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   class Program
   {
      // Рекурсивный метод
      static int factorial(int i)
      {
         int result;

         if (i == 1)
            return 1;
         result = factorial(i - 1) * i;
         return result;
      }

      static void Main(string[] args)
      {
         label1:
         Console.WriteLine("Введите число: ");
         try
         {
            int i = int.Parse(Console.ReadLine());
            Console.WriteLine("{0}! = {1}",i,factorial(i));
         }
         catch (FormatException)
         {
```

```
                Console.WriteLine("Некорректное число");
                goto label1;
            }

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



Обратите внимание, что рекурсивный метод factorial вызывает сам себя, при этом переменная i с каждым вызовом уменьшается на 1.

Рекурсивные варианты многих процедур могут выполняться немного медленнее, чем их итерационные эквиваленты из-за дополнительных затрат системных ресурсов на неоднократные вызовы метода. Если же таких вызовов окажется слишком много, то в конечном итоге может быть переполнен системный стек. А поскольку параметры и локальные переменные рекурсивного метода хранятся в системном стеке и при каждом новом вызове этого метода создается их новая копия, то в какой-то момент стек может оказаться исчерпанным. В этом случае возникает исключительная ситуация, и общеязыковая исполняющая среда (CLR) генерирует соответствующее исключение. Но беспокоиться об этом придется лишь в том случае, если рекурсивная процедура выполняется неправильно.

Главное преимущество рекурсии заключается в том, что она позволяет реализовать некоторые алгоритмы яснее и проще, чем итерационным способом. Например, алгоритм быстрой сортировки довольно трудно реализовать итерационным способом. А некоторые задачи, например искусственного интеллекта, очевидно, требуют именно рекурсивного решения.

При написании рекурсивных методов следует непременно указать в соответствующем месте условный оператор, например if, чтобы организовать возврат из метода без рекурсии. В противном случае возврата из вызванного однажды рекурсивного метода может вообще не произойти. Подобного рода ошибка весьма характерна для реализации рекурсии в практике программирования. В этом случае рекомендуется пользоваться операторами, содержащими вызовы метода WriteLine(), чтобы следить за происходящим в рекурсивном методе и прервать его выполнение, если в нем обнаружится ошибка.

Тренировочные задачи на рекурсию

В этом разделе собраны тренировочные задачи. Их решение необязательно, но рекомендуется тем, кто плохо освоил задачи на рекурсию. Также в этом разделе есть несколько довольно сложных задач, которые могут быть интересны всем.

В задачах этого раздела нельзя использовать циклы и массивы. Также могут быть и другие ограничения в каких-то конкретных задачах (например, запрет использования строк в арифметических задачах).

Задачи проверяются только автоматически и сразу же получают OK при прохождении всех тестов. Однако, если при последующей проверке кода будет выявлено нарушение вышеизложенных требований или будут другие замечания к решению, статус OK может быть изменен.

A: От 1 до n

Дано натуральное число *n*. Выведите все числа от 1 до *n*.

Ввод	Вывод
5	1 2 3 4 5

```
public class Solution {
   public static String recursion(int n) {
      // Базовый случай
      if (n == 1) {
         return "1";
      }
      // Шаг рекурсии / рекурсивное условие
      return recursion(n - 1) + " " + n;
   }
   public static void main(String[] args) {
      System.out.println(recursion(10)); // вызов рекурсивной функции
   }
}
```

B: От A до B

Даны два целых числа A и B (каждое в отдельной строке). Выведите все числа от A до B включительно, в порядке возрастания, если A < B, или в порядке убывания в противном случае.

Ввод	Вывод
5 1	5 4 3 2 1

```
public class Solution {
   public static String recursion(int a, int b) {
         // основное условие задачи
      if (a > b) {
         // Базовый случай
         if (a == b) {
            return Integer.toString(a);
         }
         // Шаг рекурсии / рекурсивное условие
         return a + " " + recursion(a - 1, b);
      } else {
         // Базовый случай
         if (a == b) {
            return Integer.toString(a);
         }
         // Шаг рекурсии / рекурсивное условие
         return a + " " + recursion(a + 1, b);
      }
   }
   public static void main(String[] args) {
      System.out.println(recursion(20, 15)); // вызов рекурсивной функции
      System.out.println(recursion(10, 15)); // вызов рекурсивной функции
   }
}
```

C: Функция Аккермана

В теории вычислимости важную роль играет функция Аккермана $A(m,n)$, определенная следующим образом:

$$A(m,\ n) = \begin{cases} n+1, & m = 0; \\ A(m-1,\ 1), & m > 0,\ n = 0; \\ A(m-1,\ A(m,\ n-1)), & m > 0,\ n > 0. \end{cases}$$

Даны два целых неотрицательных числа m и n, каждое в отдельной строке. Выведите $A(m,n)$.

Ввод	Вывод
2 2	7

```
public class Solution {
   public static int recursion(int m, int n) {
      // Базовый случай
      if (m == 0) {
         return n + 1;
      } // Шаг рекурсии / рекурсивное условие
```

```
        else if (n == 0 && m > 0) {
            return recursion(m - 1, 1);
        } // Шаг рекурсии / рекурсивное условие
        else {
            return recursion(m - 1, recursion(m, n - 1));
        }
    }
    public static void main(String[] args) {
        System.out.println(recursion(3, 2)); // вызов рекурсивной функции
    }
}
```

D: Точная степень двойки

Дано натуральное число N. Выведите слово YES, если число N является точной степенью двойки, или слово NO в противном случае.
Операцией возведения в степень пользоваться нельзя!

Ввод	**Вывод**
8	YES
3	NO

```
public class Solution {
    public static int recursion(double n) {
        // Базовый случай
        if (n == 1) {
            return 1;
        } // Базовый случай
        else if (n > 1 && n < 2) {
            return 0;
        } // Шаг рекурсии / рекурсивное условие
        else {
            return recursion(n / 2);
        }
    }
    public static void main(String[] args) {
        double n = 64;
        // вызов рекурсивной функции
        if (recursion(n) == 1) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }
    }
}
```

E: Сумма цифр числа

Дано натуральное число N. Вычислите сумму его цифр.

При решении этой задачи нельзя использовать строки, списки, массивы (ну и циклы, разумеется).

Ввод	Вывод
179	17

```
public class Solution {
   public static int recursion(int n) {
      // Базовый случай
      if (n < 10) {
         return n;
      }// Шаг рекурсии / рекурсивное условие
      else {
         return n % 10 + recursion(n / 10);
      }
   }
   public static void main(String[] args) {
      System.out.println(recursion(123)); // вызов рекурсивной функции
   }
}
```

F: Цифры числа справа налево

Дано натуральное число N. Выведите все его цифры по одной, в обратном порядке, разделяя их пробелами или новыми строками.

При решении этой задачи нельзя использовать строки, списки, массивы (ну и циклы, разумеется). Разрешена только рекурсия и целочисленная арифметика.

Ввод	Вывод
179	9 7 1

```
public class Solution {
   public static int recursion(int n) {
      // Базовый случай
      if (n < 10) {
         return n;
      }// Шаг рекурсии / рекурсивное условие
      else {
         System.out.print(n % 10 + " ");
         return recursion(n / 10);
      }
   }
   public static void main(String[] args) {
```

```
        System.out.println(recursion(123)); // вызов рекурсивной функции
    }
}
```

G: Цифры числа слева направо

Дано натуральное число N. Выведите все его цифры по одной, в обычном порядке, разделяя их пробелами или новыми строками.

При решении этой задачи нельзя использовать строки, списки, массивы (ну и циклы, разумеется). Разрешена только рекурсия и целочисленная арифметика.

Ввод	**Вывод**
179	1 7 9

```
public class Solution {
    public static String recursion(int n) {
        // Базовый случай
        if (n < 10) {
            return Integer.toString(n);
        } // Шаг рекурсии / рекурсивное условие
        else {
            return recursion(n / 10) + " " + n % 10;
        }
    }
    public static void main(String[] args) {
        System.out.println(recursion(153)); // вызов рекурсивной функции
    }
}
```

H: Проверка числа на простоту

Дано натуральное число $n>1$. Проверьте, является ли оно простым. Программа должна вывести слово YES, если число простое и NO, если число составное. Алгоритм должен иметь сложность $O(n\sqrt{})$.

Ввод	**Вывод**
2	YES
4	NO

Указание. Понятно, что задача сама по себе нерекурсивна, т.к. проверка числа n на простоту никак не сводится к проверке на простоту меньших чисел. Поэтому нужно сделать еще один параметр рекурсии: делитель числа, и именно по этому параметру и делать рекурсию.

```
public class Solution {
    public static boolean recursion(int n, int i) {
        // i- дополнительный параметр. При вызове должен быть равен 2
```

```
        // Базовый случай
        if (n < 2) {
            return false;
        } // Базовый случай
        else if (n == 2) {
            return true;
        } // Базовый случай
        else if (n % i == 0) {
            return false;
        } // Шаг рекурсии / рекурсивное условие
        else if (i < n / 2) {
            return recursion(n, i + 1);
        } else {
            return true;
        }
    }
    public static void main(String[] args) {
        System.out.println(recursion(18, 2)); // вызов рекурсивной функции
    }
}
```

I: Разложение на множители

Дано натуральное число $n>1$. Выведите все простые делители этого числа в порядке неубывания с учетом кратности. Алгоритм должен иметь сложность $O(n\sqrt{})$.

Ввод	**Вывод**
18	2 3 3

```
public class Solution {
    public static void recursion(int n, int k) {
        // k- дополнительный параметр. При вызове должен быть равен 2
        // Базовый случай
        if (k > n / 2) {
            System.out.println(n);
            return;
        }
        // Шаг рекурсии / рекурсивное условие
        if (n % k == 0) {
            System.out.println(k);
            recursion(n / k, k);
        } // Шаг рекурсии / рекурсивное условие
        else {
            recursion(n, ++k);
```

```
        }
    }
    public static void main(String[] args) {
        recursion(150, 2); // вызов рекурсивной функции
    }
}
```

J: Палиндром

Дано слово, состоящее только из строчных латинских букв. Проверьте, является ли это слово палиндромом. Выведите YES или NO.

При решении этой задачи нельзя пользоваться циклами, в решениях на питоне нельзя использовать срезы с шагом, отличным от 1.

Ввод	**Вывод**
radar	YES
yes	NO

```
public class Solution {
    public static String recursion(String s) {
        // Базовый случай
        if (s.length() == 1) {
            return "YES";
        } else {
            if (s.substring(0, 1).equals(s.substring(s.length() - 1, s.length()))) {
                // Базовый случай
                if (s.length() == 2) {
                    return "YES";
                }
                // Шаг рекурсии / рекурсивное условие
                return recursion(s.substring(1, s.length() - 1));
            } else {
                return "NO";
            }
        }
    }
    public static void main(String[] args) {
        System.out.println(recursion("radar")); // вызов рекурсивной функции
    }
}
```

Другое решение

```
public class Solution {
    public static boolean recursion(String s) {
        char firstChar;
```

```
        char lastChar;
        int size = s.length();
        String subString;
        // Базовый случай
        if (size <= 1) {
            return true;
        } else {
            firstChar = s.toCharArray()[0];
            lastChar = s.toCharArray()[size - 1];
            subString = s.substring(1, size - 1);
            // Шаг рекурсии / рекурсивное условие
            return firstChar == lastChar && recursion(subString);
        }
    }
    public static void main(String[] args) {
        // вызов рекурсивной функции
        if (recursion("radar")) {
            System.out.println("YES");
        } else {
            System.out.println("NO");
        }
    }
}
```

K: Вывести нечетные числа последовательности

Дана последовательность натуральных чисел (одно число в строке), завершающаяся числом 0. Выведите все нечетные числа из этой последовательности, сохраняя их порядок.

В этой задаче нельзя использовать глобальные переменные и передавать какие-либо параметры в рекурсивную функцию. Функция получает данные, считывая их с клавиатуры. Функция не возвращает значение, а сразу же выводит результат на экран. Основная программа должна состоять только из вызова этой функции.

Ввод	**Вывод**
3 1 2 0	3 1

```
public class Solution {
    public static void recursion() {
        java.util.Scanner in = new java.util.Scanner(System.in);
```

```
        int n = in.nextInt();
        // Базовый случай
        if (n > 0) {
            // Шаг рекурсии / рекурсивное условие
            if (n % 2 == 1) {
                System.out.println(n);
                recursion();
            } else {
                recursion();
            }
        }
    }
    public static void main(String[] args) {
        recursion(); // вызов рекурсивной функции
    }
}
```

L: Вывести члены последовательности с нечетными номерами

Дана последовательность натуральных чисел (одно число в строке), завершающаяся числом 0. Выведите первое, третье, пятое и т.д. из введенных чисел. Завершающий ноль выводить не надо.

В этой задаче нельзя использовать глобальные переменные и передавать какие-либо параметры в рекурсивную функцию. Функция получает данные, считывая их с клавиатуры. Функция не возвращает значение, а сразу же выводит результат на экран. Основная программа должна состоять только из вызова этой функции.

Ввод	**Вывод**
7 2 9 5 0	7 9

```
public class Solution {
    public static void recursion() {
        java.util.Scanner in = new java.util.Scanner(System.in);
        int n = in.nextInt();
        // Базовый случай
        if (n > 0) {
            int m = in.nextInt();
            System.out.println(n);
            // Базовый случай
```

```
            if (m > 0) {
                // Шаг рекурсии / рекурсивное условие
                recursion();
            }
        }
    }
    public static void main(String[] args) {
        recursion(); // вызов рекурсивной функции
    }
}
```

M: Максимум последовательности

Дана последовательность натуральных чисел (одно число в строке), завершающаяся числом 0. Определите наибольшее значение числа в этой последовательности.

В этой задаче нельзя использовать глобальные переменные и передавать какие-либо параметры в рекурсивную функцию. Функция получает данные, считывая их с клавиатуры. Функция возвращает единственное значение: максимум считанной последовательности. Гарантируется, что последовательность содержит хотя бы одно число (кроме нуля).

Ввод	**Вывод**
1 7 2 4 0	7

```
public class Solution {
    public static int recursion() {
        java.util.Scanner in = new java.util.Scanner(System.in);
        int n = in.nextInt();
        // Базовый случай
        if (n == 0) {
            return 0;
        } // Шаг рекурсии / рекурсивное условие
        else {
            return Math.max(n, recursion());
        }
    }
    public static void main(String[] args) {
        System.out.println(recursion()); // вызов рекурсивной функции
    }
}
```

N: Среднее значение последовательности

Дана последовательность натуральных чисел (одно число в строке), завершающаяся числом 0. Определите среднее значение элементов этой последовательности (без учета последнего нуля).

В этой задаче нельзя использовать глобальные переменные. Функция получает данные, считывая их с клавиатуры, а не получая их в виде параметра. В программе на языке Python функция возвращает кортеж из пары чисел: число элементов в последовательности и их сумма. В программе на языке C++ результат записывается в две переменные, которые передаются в функцию по ссылке.

Гарантируется, что последовательность содержит хотя бы одно число (кроме нуля).

Ввод	Вывод
1 7 9 0	5.666666666666667

```
public class Solution {
   public static void recursion(int sum, int count) {
      java.util.Scanner in = new java.util.Scanner(System.in);
      int n = in.nextInt();
      // Базовый случай
      if (n > 0) {
         // Шаг рекурсии / рекурсивное условие
         recursion(sum + n, ++count);
      } else if (sum > 0 && count > 0) {
         System.out.println((float) sum / count);
      }
   }
   public static void main(String[] args) {
      recursion(0, 0); // вызов рекурсивной функции
   }
}
```

O: Второй максимум

Дана последовательность натуральных чисел (одно число в строке), завершающаяся числом 0. Определите значение второго по величине элемента в этой последовательности, то есть элемента, который будет наибольшим, если из последовательности удалить наибольший элемент.

В этой задаче нельзя использовать глобальные переменные. Функция получает данные, считывая их с клавиатуры, а не получая их в виде параметра. В программе на языке Python функция возвращает результат в

виде кортежа из нескольких чисел и функция вообще не получает никаких параметров. В программе на языке C++ результат записывается в переменные, которые передаются в функцию по ссылке. Других параметров, кроме как используемых для возврата значения, функция не получает.

Гарантируется, что последовательность содержит хотя бы два числа (кроме нуля).

Ввод	Вывод
1 7 9 0	7
1 2 2 1 0	2

```
public class Solution {
    public static void recursion(int max1, int max2) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        // Базовый случай
        if (n > 0) {
            // Шаг рекурсии / рекурсивное условие
            if (max1 < n) {
                recursion(n, max1);
            } // Шаг рекурсии / рекурсивное условие
            else if (max2 < n) {
                recursion(max1, n);
            } // Шаг рекурсии / рекурсивное условие
            else {
                recursion(max1, max2);
            }
        } else {
            System.out.println(max2);
        }
    }
    public static void main(String[] args) {
        recursion(0, 0); // вызов рекурсивной функции
    }
}
```

P: Количество элементов, равных максимуму

Дана последовательность натуральных чисел (одно число в строке), завершающаяся числом 0. Определите, какое количество элементов этой последовательности, равны ее наибольшему элементу.

В этой задаче нельзя использовать глобальные переменные. Функция получает данные, считывая их с клавиатуры, а не получая их в виде параметра. В программе на языке Python функция возвращает результат в виде кортежа из нескольких чисел и функция вообще не получает никаких параметров. В программе на языке C++ результат записывается в переменные, которые передаются в функцию по ссылке. Других параметров, кроме как используемых для возврата значения, функция не получает.

Гарантируется, что последовательность содержит хотя бы одно число (кроме нуля).

Ввод	Вывод
1 7 9 0	1
1 3 3 1 0	2

```
public class Solution {
    public static void recursion(int max, int count) {
        java.util.Scanner in = new java.util.Scanner(System.in);
        int n = in.nextInt();
        // Базовый случай
        if (n > 0) {
            // Шаг рекурсии / рекурсивное условие
            if (n > max) {
                recursion(n, 1);
            } // Шаг рекурсии / рекурсивное условие
            else if (n == max) {
                recursion(max, ++count);
            } // Шаг рекурсии / рекурсивное условие
            else {
                recursion(max, count);
            }
        } else {
```

```
            System.out.println(count);
        }
    }
    public static void main(String[] args) {
        recursion(0, 0); // вызов рекурсивной функции
    }
}
```

Q: Количество единиц

Дана последовательность натуральных чисел (одно число в строке), завершающаяся двумя числами 0 подряд. Определите, сколько раз в этой последовательности встречается число 1. Числа, идущие после двух нулей, необходимо игнорировать.

В этой задаче нельзя использовать глобальные переменные и параметры, передаваемые в функцию. Функция получает данные, считывая их с клавиатуры, а не получая их в виде параметров.

Ввод	**Вывод**
1 0 1 0 0 1	2

```
public class Solution {
    public static int recursion() {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        // Базовый случай
        if (n == 1) {
            int m = in.nextInt();
            // Базовый случай
            if (m == 1) {
                // Шаг рекурсии / рекурсивное условие
                return recursion() + n + m;
            } else {
                int k = in.nextInt();
                // Базовый случай
                if (k == 1) {
                    // Шаг рекурсии / рекурсивное условие
                    return recursion() + n + m + k;
                } else {
```

```
                return n + m + k;
            }
        }
    } else {
        int m = in.nextInt();
        // Базовый случай
        if (m == 1) {
            // Шаг рекурсии / рекурсивное условие
            return recursion() + n + m;
        } else {
            return n + m;
        }
    }
}
public static void main(String[] args) {
    System.out.println(recursion()); // вызов рекурсивной функции
}
}
```

R: Треугольная последовательность

Дана монотонная последовательность, в которой каждое натуральное число k встречается ровно k раз: 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, ...

По данному натуральному n выведите первые n членов этой последовательности. Попробуйте обойтись только одним циклом for.

Ввод	Вывод
2	1 2
5	1 2 2 3 3

```
public class Solution {
    public static String recursion(int n) {
        int sum = 0;
        int j = 0;
        // Базовый случай
        if (n == 1) {
            System.out.print("1");
        } else {
            for (int i = 1; sum < n; i++) {
                sum += i;
                j = i;
            }
            // Шаг рекурсии / рекурсивное условие
            System.out.print(recursion(--n) + " " + j);
```

```
        }
        return "";
    }
    public static void main(String[] args) {
        recursion(5); // вызов рекурсивной функции
    }
}
```

S: Заданная сумма цифр

Даны натуральные числа *k* и *s*. Определите, сколько существует *k*-значных натуральных чисел, сумма цифр которых равна *d*. Запись натурального числа не может начинаться с цифры 0.

В этой задаче можно использовать цикл для перебора всех цифр, стоящих на какой-либо позиции.

Ввод	**Вывод**
3 15	69

```
public class Solution {
    public static int recursion(int len, int sum, int k, int s) {
        // Базовый случай
        if (len == k) {
            if (sum == s) {
                return 1;
            } else {
                return 0;
            }
        }
        int c = (len == 0 ? 1 : 0);
        int res = 0;
        // Шаг рекурсии / рекурсивное условие
        for (int i = c; i < 10; i++) {
            res += recursion(len + 1, sum + i, k, s);
        }
        return res;
    }
    public static void main(String[] args) {
        System.out.println(recursion(0, 0, 3, 15)); // вызов рекурсивной функции
    }
}
```

T: Без двух нулей

Даны числа *a* и *b*. Определите, сколько существует последовательностей из *a* нулей и *b* единиц, в которых никакие два нуля не стоят рядом.

Ввод	Вывод
2 2	3

```
public class Solution {
   public static int recursion(int a, int b) {
      // Базовый случай
      if (a > b + 1) {
         return 0;
      }
      // Базовый случай
      if (a == 0 || b == 0) {
         return 1;
      }
      // Шаг рекурсии / рекурсивное условие
      return recursion(a, b - 1) + recursion(a - 1, b - 1);
   }
   public static void main(String[] args) {
      System.out.println(recursion(5, 8)); // вызов рекурсивной функции
   }
}
```

U: Разворот числа

Дано число *n*, десятичная запись которого не содержит нулей. Получите число, записанное теми же цифрами, но в противоположном порядке.

При решении этой задачи нельзя использовать циклы, строки, списки, массивы, разрешается только рекурсия и целочисленная арифметика.

Фунция должна возвращать целое число, являющееся результатом работы программы, выводить число по одной цифре нельзя.

Ввод	Вывод
179	971

```
public class Solution {
   public static int recursion(int n, int i) {
      return (n==0) ? i : recursion( n/10, i*10 + n%10 );
   }
   public static void main(String[] args) {
```

```
        System.out.println(recursion(158, 0));
    }
}
// Fibonacci.cpp : Defines the entry point for the console application.
//
#include <conio.h>
 #include "stdafx.h"
// #include <iostream.h>
 #include <stdlib.h>
 int fibo(int n)  //рекурсивная функция
{
  if (n==0||n==1) return n; //применена логическая операция или
 else return fibo(n-1)+fibo(n-2); //в этой строке вызываем функцию рекурсивно
}
 void main()
{
  //system("cls");
   int n;
   //cout<<"N=";cin>>n;
   scanf("%i",&n);;
   //printf("Fibonachi = %i",fibo(n)); //Точка вызова рекурсивной функции
  for (int i=1;i<=n;i++)  //Вывести все числа можно циклом, либо же
непосредственно внутри рекурсивной функции. Я циклом.
      // std::cout<<"Fibonachi = "<<fibo(i)<<"\n"; //Точка вызова рекурсивной
функции
 printf("Fibonachi = %i\n",fibo(i)); //Точка вызова рекурсивной функции
   //getch();
  return;
}
```

4. Использование ключевое слово static, статические классы. Статическая концепция, использование ключевого слова Static, работа со статическими классами

Иногда требуется определить такой член класса, который будет использоваться независимо от всех остальных объектов этого класса. Как правило, доступ к члену класса организуется посредством объекта этого класса, но в то же время можно создать член класса для самостоятельного применения без ссылки на конкретный экземпляр объекта. Для того чтобы создать такой член класса, достаточно указать в самом начале его объявления ключевое слово static.

Если член класса объявляется как static, то он становится доступным до создания любых объектов своего класса и без ссылки на какой-нибудь объект. С помощью ключевого слова static можно объявлять как переменные, так и методы. Наиболее характерным примером члена типа static служит метод Main(), который объявляется таковым потому, что он должен вызываться операционной системой в самом начале выполняемой программы.

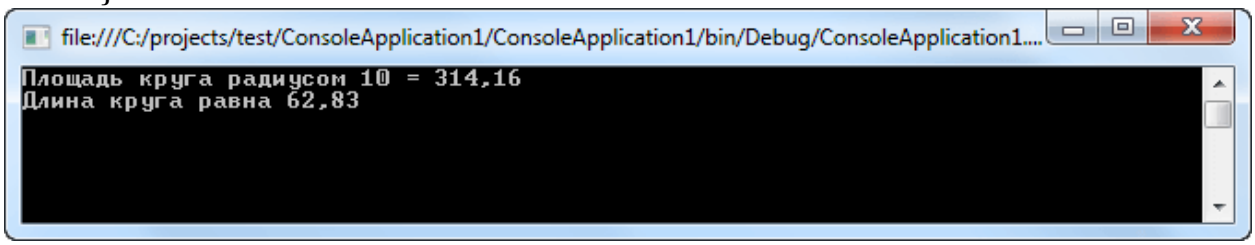
Для того чтобы воспользоваться членом типа static за пределами класса, достаточно указать имя этого класса с оператором-точкой. Но создавать объект для этого не нужно. В действительности член типа static оказывается доступным не по ссылке на объект, а по имени своего класса.

Переменные, объявляемые как static, по существу, являются глобальными. Когда же объекты объявляются в своем классе, то копия переменной типа static не создается. Вместо этого все экземпляры класса совместно пользуются одной и той же переменной типа static. Такая переменная инициализируется перед ее применением в классе.

Пример использования ключевого слова static:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   class myCircle
   {
      // 2 метода, возвращающие площадь и длину круга
      public static double SqrCircle(int radius)
      {
         return Math.PI * radius * radius;
      }
```

```
        public static double LongCircle(int radius)
        {
            return 2 * Math.PI * radius;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            int r = 10;
            // Вызов методов из другого класса
            // без создания экземпляра объекта этого класса
            Console.WriteLine("Площадь круга радиусом {0} = {1:#.##}",r,myCircle.SqrCircle(r));
            Console.WriteLine("Длина круга равна {0:#.##}",myCircle.LongCircle(r));

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



На применение методов типа static накладывается ряд следующих ограничений:

В методе типа static должна отсутствовать ссылка this, поскольку такой метод не выполняется относительно какого-либо объекта

В методе типа static допускается непосредственный вызов только других методов типа static, но не метода экземпляра из того самого же класса. Дело в том, что методы экземпляра оперируют конкретными объектами, а метод типа static не вызывается для объекта. Следовательно, у такого метода отсутствуют объекты, которыми он мог бы оперировать

Аналогичные ограничения накладываются на данные типа static. Для метода типа static непосредственно доступными оказываются только другие данные типа static, определенные в его классе. Он, в частности, не может оперировать переменной экземпляра своего класса, поскольку у него отсутствуют объекты, которыми он мог бы оперировать

Статические конструкторы

Конструктор можно также объявить как static. Статический конструктор, как правило, используется для инициализации компонентов,

применяемых ко всему классу, а не к отдельному экземпляру объекта этого класса. Поэтому члены класса инициализируются статическим конструктором до создания каких-либо объектов этого класса:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class MyClass
    {
        public static int a;
        public int b;

        // Статический конструктор
        static MyClass()
        {
            a = 10;
        }

        // Обычный конструктор
        public MyClass()
        {
            b = 12;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Доступ к экземпляру класса a: " + MyClass.a);

            MyClass obj = new MyClass();
            Console.WriteLine("Доступ к экземпляру класса b: " + obj.b);

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

Обратите внимание на то, что конструктор типа static вызывается автоматически, когда класс загружается впервые, причем до конструктора

экземпляра. Из этого можно сделать более общий вывод: статический конструктор должен выполняться до любого конструктора экземпляра. Более того, у статических конструкторов отсутствуют модификаторы доступа — они пользуются доступом по умолчанию, а следовательно, их нельзя вызывать из программы.

Статические классы

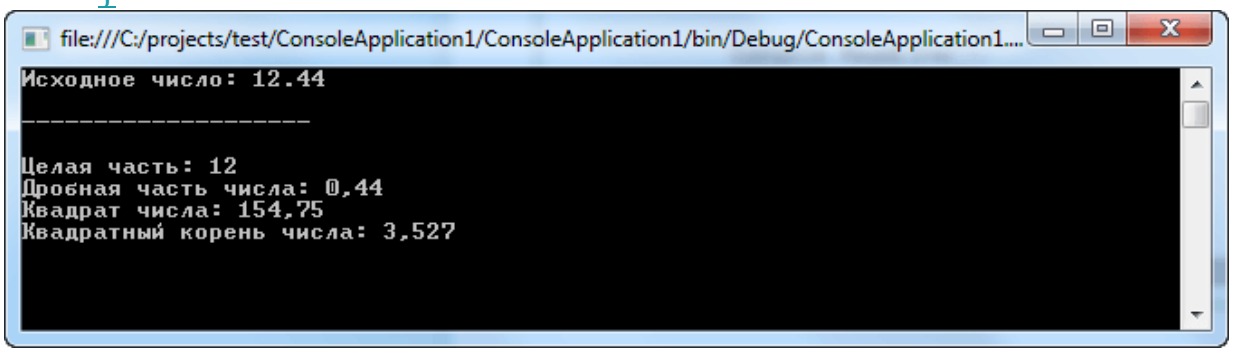
Класс можно объявлять как static. Статический класс обладает двумя основными свойствами. Во-первых, объекты статического класса создавать нельзя. И во-вторых, статический класс должен содержать только статические члены. Статический класс создается по приведенной ниже форме объявления класса, видоизмененной с помощью ключевого слова static.

static class имя класса { // ...

Статические классы применяются главным образом в двух случаях. Во-первых, статический класс требуется при создании метода расширения. Методы расширения связаны в основном с языком LINQ. И во-вторых, статический класс служит для хранения совокупности связанных друг с другом статических методов:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    // В данном классе инкапсулируются статические методы
    // выполняющие простейшие операции
    static class MyMath
    {
        // Целая часть числа
        static public int round(double d)
        {
            return (int)d;
        }

        // Дробная часть числа
        static public double doub(double d)
        {
            return d - (int)d;
        }

        // Квадрат числа
        static public double sqr(double d)
        {
```

```
        return d * d;
    }

    // Квадратный корень числа
    static public double sqrt(double d)
    {
        return Math.Sqrt(d);
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
        Console.WriteLine("Исходное число: 12.44\n\n----------
\n");
        Console.WriteLine("Целая часть: {0}",MyMath.round(d: 12.44));
        Console.WriteLine("Дробная часть числа: {0}",MyMath.doub(d:
12.44));
        Console.WriteLine("Квадрат числа: {0:#.##}",MyMath.sqr(d:
12.44));
        Console.WriteLine("Квадратный корень числа:
{0:#.###}",MyMath.sqrt(d: 12.44));

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



Стоит отметить, что для статического класса не допускается наличие конструктора экземпляра, но у него может быть статический конструктор.

5.Свойства и инкапсуляция

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

Стандартное описание свойства имеет следующий синтаксис:

```
[модификатор_доступа] возвращаемый_тип произвольное_название
{
    // код свойства
}
```

Например:

```
class Person
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }
}
```

Здесь у нас есть закрытое поле name и есть общедоступное свойство Name. Хотя они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия у них могут быть произвольные и не обязательно должны совпадать.

Через это свойство мы можем управлять доступом к переменной name. Стандартное определение свойства содержит блоки **get** и **set**. В блоке get мы возвращаем значение поля, а в блоке set устанавливаем. Параметр value представляет передаваемое значение.

Мы можем использовать данное свойство следующим образом:

```
Person p = new Person();

// Устанавливаем свойство - срабатывает блок Set
// значение "Tom" и есть передаваемое в свойство value
p.Name = "Tom";

// Получаем значение свойства и присваиваем
его переменной - срабатывает блок Get
string personName = p.Name;
```

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, например, при присвоении переменной класса какого-либо значения. Например, нам надо установить проверку по возрасту:

```
class Person
{
    private int age;

    public int Age
    {
        set
        {
            if (value < 18)
            {
                Console.WriteLine("Возраст должен быть больше 17");
            }
            else
            {
                age = value;
            }
        }
        get { return age; }
    }
}
```

Блоки set и get не обязательно одновременно должны присутствовать в свойстве. Если свойство определяют 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:

```
1    class Person
2    {
3        private string name;
4        // свойство только для чтения
5        public string Name
6        {
7            get
8            {
9                return name;
10           }
11       }
12
13       private int age;
14       // свойство только для записи
15       public int Age
16       {
17           set
18           {
19               age = value;
20           }
21       }
22   }
```

Модификаторы доступа

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

```
1    class Person
2    {
3        private string name;
4
5        public string Name
6        {
7            get
8            {
9                return name;
10           }
11
12           private set
13           {
14               name = value;
15           }
16       }
17       public Person(string name, int age)
```

```
18      {
19          Name = name;
20          Age = age;
21      }
22  }
```

Теперь закрытый блок set мы сможем использовать только в данном классе - в его методах, свойствах, конструкторе, но никак не в другом классе:

```
1   Person p = new Person("Tom", 24);
2
3   // Ошибка - set объявлен с модификатором private
4   //p.Name = "John";
5
6   Console.WriteLine(p.Name);
```

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

Инкапсуляция

Выше мы посмотрели, что через свойства устанавливается доступ к приватным переменным класса. Подобное сокрытие состояния класса от вмешательства извне представляет механизм **инкапсуляции**, который представляет одну из ключевых концепций объектно-ориентированного программирования. (Стоит отметить, что само понятие инкапсуляции имеет довольно много различных трактовок, которые не всегда пересекаются друг с другом) Применение модификаторов доступа типа private защищает переменную от внешнего доступа. Для управления доступом во многих языках программирования используются специальные методы, геттеры и сеттеры. В C# их роль, как правило, выполняют свойства.

Например, есть некоторый класс Account, в котором определено поле sum, представляющее сумму:

```
1   class Account
```

```
2   {
3       public int sum;
4   }
```

Поскольку переменная sum является публичной, то в любом месте программы мы можем получить к ней доступ и изменить ее, в том числе установить какое-либо недопустимое значение, например, отрицательное. Вряд ли подобное поведение является желательным. Поэтому применяется инкапсуляция для ограничения доступа к переменной sum и сокрытию ее внутри класса:

```
1     class Account
2     {
3         private int sum;
4         public int Sum
5         {
6             get {return sum;}
7             set
8             {
9                 if (value > 0)
10                {
11                    sum=value;
12                }
13            }
14        }
15    }
```

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 фреймворк .NET были добавлены автоматические свойства. Они имеют сокращенное объявление:

```
1     class Person
2     {
3         public string Name { get; set; }
4         public int Age { get; set; }
5
6         public Person(string name, int age)
7         {
8             Name = name;
9             Age = age;
10        }
```

```
11    }
```

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):

```
1     class Person
2     {
3         public string Name { get; set; } = "Tom";
4         public int Age { get; set; } = 23;
5     }
6
7     class Program
8     {
9         static void Main(string[] args)
10        {
11            Person person = new Person();
12            Console.WriteLine(person.Name); // Tom
13            Console.WriteLine(person.Age);  // 23
14
15            Console.Read();
16        }
17    }
```

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:

```
1    class Person
2    {
3        public string Name { private set; get;}
4        public Person(string n)
5        {
6            Name = n;
7        }
8    }
```

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:

```
1    class Person
2    {
3        public string Name { get;} = "Tom"
4    }
```

Сокращенная запись свойств

Как и методы, мы можем сокращать свойства. Например:

```
1    class Person
2    {
3        private string name;
4
5        // эквивалентно public string Name { get { return name; } }
6        public string Name => name;
7    }
```

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 индекатора:

```
1возвращаемый_тип this [Тип параметр1, ...]
2{
3    get { ... }
4    set { ... }
5}
```

В отличие от свойств индексатор не имеет названия. Вместо его 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 People, который представляет группу людей. Используем индексаторы для определения класса People:

```
class Person
{
    public string Name { get; set; }
}
class People
{
    Person[] data;
    public People()
    {
        data = new Person[5];
    }
    // индексатор
    public Person this[int index]
    {
        get
        {
            return data[index];
        }
        set
        {
            data[index] = value;
        }
    }
}
```

Конструкция public Person this[int index] и представляет индексатор. Здесь определяем, во-первых, тип возвращаемого или присваиваемого объекта, то есть тип Person. Во-вторых, определяем через параметр int index способ доступа к элементам.

По сути все объекты Person хранятся в классе в массиве data. Для получения их по индексу в индекаторе определен блок get:

```
get
{
   return data[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 data.

В блоке set получаем через параметр **value** переданный объект Person и сохраняем его в массив по индексу.

```
1set
2{
3   data[index] = value;
4}
```

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

```
1  class Program
2      {
3      static void Main(string[] args)
4      {
5          People people = new People();
6          people[0] = new Person { Name = "Tom" };
7          people[1] = new Person { Name = "Bob" };
8
9          Person tom = people[0];
10         Console.WriteLine(tom?.Name);
11
12         Console.ReadKey();
13     }
14}
```

Индексатор, как полагается получает набор индексов в виде параметров. Однако индексы необязаткельно должны представлять тип int. Например, мы можем рассматривать объект как хранилище свойств и передавать имя атрибута объекта в виде строки:

```
1  class User
2  {
3      string name;
4      string email;
5      string phone;
6
7      public string this[string propname]
8      {
9          get
10         {
11             switch (propname)
```

```
12          {
13              case "name": return "Mr/Ms. " + name;
14              case "email": return email;
15              case "phone": return phone;
16              default: return null;
17          }
18      }
19      set
20      {
21          switch (propname)
22          {
23              case "name":
24                  name = value;
25                  break;
26              case "email":
27                  email = value;
28                  break;
29              case "phone":
30                  phone = value;
31                  break;
32          }
33      }
34  }
35}
36class Program
37{
38  static void Main(string[] args)
39  {
40      User tom = new User();
41      tom["name"] = "Tom";
42      tom["email"] = "tomekvilmovskiy@gmail.ru";
43
44      Console.WriteLine(tom["name"]); // Mr/Ms. Tom
45
46      Console.ReadKey();
47  }
48}
```

Применение нескольких параметров

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

```
1  class Matrix
```

```
2  {
3     private int[,] numbers = new int[,] { { 1, 2, 4}, { 2, 3, 6 }, { 3, 4, 8 } };
4     public int this[int i, int j]
5     {
6        get
7        {
8           return numbers[i,j];
9        }
10       set
11       {
12          numbers[i, j] = value;
13       }
14    }
15}
```

Теперь для определения индексатора используются два индекса - i и j. И в программе мы уже должны обращаться к объекту, используя два индекса:

```
1Matrix matrix = new Matrix();
2Console.WriteLine(matrix[0, 0]);
3matrix[0, 0] = 111;
4Console.WriteLine(matrix[0, 0]);
```

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

```
1  class Matrix
2  {
3     private int[,] numbers = new int[,] { { 1, 2, 4}, { 2, 3, 6 }, { 3, 4, 8 } };
4     public int this[int i, int j]
5     {
6        get
7        {
8           return numbers[i,j];
9        }
10    }
11}
```

Также мы можем ограничивать доступ к блокам get и set, используя модификаторы доступа. Например, сделаем блок set приватным:

```
1  class Matrix
2  {
3      private int[,] numbers = new int[,] { { 1, 2, 4}, { 2, 3, 6 }, { 3, 4, 8 } };
4      public int this[int i, int j]
5      {
6          get
7          {
8              return numbers[i,j];
9          }
10         private set
11         {
12             numbers[i, j] = value;
13         }
14     }
15}
```

Перегрузка индексаторов

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

```
class Person

{

    public string Name { get; set; }

    public int Age { get; set; }

}

class People

{

    Person[] data;

    public People()

    {
```

```
    data = new Person[5];
}
public Person this[int index]
{
    get
    {
        return data[index];
    }
    set
    {
        data[index] = value;
    }
}
public Person this[string name]
{
    get
    {
        Person person = null;
        foreach(var p in data)
        {
            if(p?.Name == name)
            {
                person = p;
```

```
                break;
            }
        }
        return person;
    }
  }
}
class Program
{
  static void Main(string[] args)
  {
    People people = new People();
    people[0] = new Person { Name = "Tom" };
    people[1] = new Person { Name = "Bob" };

    Console.WriteLine(people[0].Name);      // Tom
    Console.WriteLine(people["Bob"].Name);  // Bob

    Console.ReadKey();
  }
}
```

В данном случае класс People содержит две версии индексатора. Первая версия получает и устанавливает объект Person по индексу, а вторая - только получае объект Person по его имени.

6. Наследование

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

Пусть у нас есть следующий класс Person, который описывает отдельного человека:

```
1    class Person
2    {
3        private string _name;
4
5        public string Name
6        {
7            get { return _name; }
8            set { _name = value; }
9        }
10       public void Display()
11       {
12           Console.WriteLine(Name);
13       }
14   }
```

Но вдруг нам потребовался класс, описывающий сотрудника предприятия - класс Employee. Поскольку этот класс будет реализовывать тот же функционал, что и класс Person, так как сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (или наследником, или подклассом) от класса Person, который, в свою очередь, называется базовым классом или родителем (или суперклассом):

```
1    class Employee : Person
2    {
3
4    }
```

После двоеточия мы указываем базовый класс для данного класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же свойства, методы, поля, которые есть в классе Person. Единственное, что не передается при наследовании, это конструкторы базового класса.

Таким образом, наследование реализует отношение **is-a** (является), объект класса Employee также является объектом класса Person:

```
1    static void Main(string[] args)
2    {
3        Person p = new Person { Name = "Tom"};
4        p.Display();
5        p = new Employee { Name = "Sam" };
6        p.Display();
7        Console.Read();
8    }
```

И поскольку объект Employee является также и объектом Person, то мы можем так определить переменную: Person p = new Employee().

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**.
- Если класс объявлен с модификатором **sealed**, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

```
1    sealed class Admin
2    {
3    }
```

- Нельзя унаследовать класс от статического класса.

Доступ к членам базового класса из класса-наследника

Вернемся к нашим классам Person и Employee. Хотя Employee наследует весь функционал от класса Person, посмотрим, что будет в следующем случае:

```
1    class Employee : Person
2    {
```

```
3       public void Display()
4       {
5           Console.WriteLine(_name);
6       }
7   }
```

Этот код не сработает и выдаст ошибку, так как переменная _name объявлена с модификатором private и поэтому к ней доступ имеет только класс Person. Но зато в классе Person определено общедоступное свойство Name, которое мы можем использовать, поэтому следующий код у нас будет работать нормально:

```
1   class Employee : Person
2   {
3       public void Display()
4       {
5           Console.WriteLine(Name);
6       }
7   }
```

Таким образом, 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**, **protected** и **protected internal**.

Ключевое слово base

Теперь добавим в наши классы конструкторы:

```
1    class Person
2    {
3        public string Name { get;  set; }
4
5        public Person(string name)
6        {
7            Name = name;
8        }
9
10       public void Display()
11       {
12           Console.WriteLine(Name);
13       }
14   }
15
16   class Employee : Person
```

```
17  {
18      public string Company { get; set; }
19
20      public Employee(string name, string company)
21          : base(name)
22      {
23          Company = company;
24      }
25  }
```

Класс Person имеет конструктор, который устанавливает свойство Name. Поскольку класс Employee наследует и устанавливает то же свойство Name, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса Person. К тому же свойств, которые надо установить в конструкторе базового класса, и параметров может быть гораздо больше.

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).

```
1   static void Main(string[] args)
2   {
3       Person p = new Person("Bill");
4       p.Display();
5       Employee emp = new Employee ("Tom", "Microsoft");
6       emp.Display();
7       Console.Read();
8   }
```

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. Например, из класса Employee уберем определение конструктора:

```
1   class Employee : Person
2   {
3       public string Company { get; set; }
4   }
```

В данном случае мы получим ошибку, так как класс Employee не соответствует классу Person, а именно не вызывает конструктор базового класса. Даже если бы мы добавили какой-нибудь конструктор, который бы устанавливал все те же свойства, то мы все равно бы получили ошибку:

```
public Employee(string name, string company)
{
    Name = name;
    Company = company;
}
```

То есть в классе Employee через ключевое слово **base** надо явным образом вызвать конструктор класса Person:

```
public Employee(string name, string company)
        : base(name)
{
    Company = company;
}
```

Либо в качестве альтернативы мы могли бы определить в базовом классе конструктор без параметров:

```
class Person
{
    // остальной код класса
    // конструктор по умолчанию
    public Person()
    {
        FirstName = "Tom";
        Console.WriteLine("Вызов конструктора без параметров");
    }
}
```

Тогда в любом конструкторе производного класса, где нет обращения конструктору базового класса, все равно неявно вызывался бы этот конструктор по умолчанию. Например, следующий конструктор

```
public Employee(string company)
{
```

```
3       Company = company;
4   }
```

Фактически был бы эквивалентен следующему конструктору:

```
1   public Employee(string company)
2       :base()
3   {
4       Company = company;
5   }
```

Порядок вызова конструкторов

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных. Например, возьмем следующие классы:

```
1    class Person
2    {
3        string name;
4        int age;
5
6        public Person(string name)
7        {
8            this.name = name;
9            Console.WriteLine("Person(string name)");
10       }
11       public Person(string name, int age) : this(name)
12       {
13           this.age = age;
14           Console.WriteLine("Person(string name, int age)");
15       }
16   }
17   class Employee : Person
18   {
19       string company;
20
21       public Employee(string name, int age, string company) : base(name, age)
22       {
23           this.company = company;
24           Console.WriteLine("Employee(string name, int age, string company)");
25       }
26   }
```

При создании объекта Employee:

```
Employee tom = new Employee("Tom", 22, "Microsoft");
```

Мы получим следующий консольный вывод:

```
Person(string name)
Person(string name, int age)
Employee(string name, int age, string company)
```

В итоге мы получаем следующую цепь выполнений.

1. Вначале вызывается конструктор Employee(string name, int age, string company). Он делегирует выполнение конструктору Person(string name, int age)
2. Вызывается конструктор Person(string name, int age), который сам пока не выполняется и передает выполнение конструктору Person(string name)
3. Вызывается конструктор Person(string name), который передает выполнение конструктору класса System.Object, так как это базовый по умолчанию класс для Person.
4. Выполняется конструктор System.Object.Object(), затем выполнение возвращается конструктору Person(string name)
5. Выполняется тело конструктора Person(string name), затем выполнение возвращается конструктору Person(string name, int age)
6. Выполняется тело конструктора Person(string name, int age), затем выполнение возвращается конструктору Employee(string name, int age, string company)
7. Выполняется тело конструктора Employee(string name, int age, string company). В итоге создается объект Employee

7. Интерфейсы

Введение в интерфейсы

Интерфейс представляет ссылочный тип, который определяет набор методов и свойств, но не реализует их. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

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. Например, интерфейс IMovable:

```
1    interface IMovable
2    {
3        void Move();
4    }
```

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

Еще один момент в объявлении интерфейса: все его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

В целом интерфейсы могут определять следующие сущности:

- Методы
- Свойства
- Индексаторы
- События

Однако интерфейсы не могут определять статические члены, переменные, константы.

Затем какой-нибудь класс или структура могут применить данный интерфейс:

```
1    // применение интерфейса в классе
```

```
class Person : IMovable
{
    public void Move()
    {
        Console.WriteLine("Человек идет");
    }
}
// применение интерфейса в структуре
struct Car : IMovable
{
    public void Move()
    {
        Console.WriteLine("Машина едет");
    }
}
```

При применени интерфейса, как и при наследовании после имени класса или структуры указывается двоеточие и затем идут названия применяемых интерфейсов. При этом класс должен реализовать все методы и свойства применяемых интерфейсов. При этом поскольку все методы и свойства интерфейса являются публичными, при реализации этих методов и свойств в классе к ним можно применять только модификатор public. Поэтому если класс должен иметь метод с каким-то другим модификатором, например, protected, то интерфейс не подходит для определения подобного метода.

Если класс или структура не реализуют какие-либо свойства или методы интерфейса, то мы столкнемся с ошибкой на этапе компиляции.

Применение интерфейса в программе:

```
using System;

namespace HelloApp
{
    interface IMovable
    {
        void Move();
    }
    class Person : IMovable
    {
        public void Move()
        {
            Console.WriteLine("Человек идет");
        }
```

```
15      }
16      struct Car : IMovable
17      {
18          public void Move()
19          {
20              Console.WriteLine("Машина едет");
21          }
22      }
23      class Program
24      {
25          static void Action(IMovable movable)
26          {
27              movable.Move();
28          }
29          static void Main(string[] args)
30          {
31              Person person = new Person();
32              Car car = new Car();
33              Action(person);
34              Action(car);
35              Console.Read();
36          }
37      }
38  }
```

В данной программе определен метод Action(), который в качестве параметра принимает объект интерфейса IMovable. На момент написания кода мы можем не знать, что это будет за объект - какой-то класс или структура. Единственное, в чем мы можем быть уверены, что этот объект обязательно реализует метод Move и мы можем вызвать этот метод.

Иными словами, интерфейс - это контракт, что какой-то определенный тип обязательно реализует некоторый функционал.

Консольный вывод данной программы:

```
Человек идет
Машина едет
```

Если класс применяет интерфейс, то этот класс должен реализовать все методы и свойства интерфейса. Однако также можно и не реализовать методы, сделав их абстрактными, переложив право их реализации на производные классы:

```
1   interface IMovable
2   {
3       void Move();
4   }
5   abstract class Person : IMovable
6   {
7       public abstract void Move();
8   }
9   class Driver : Person
10  {
11      public override void Move()
12      {
13          Console.WriteLine("Шофер ведет машину");
14      }
15  }
```

Стоит отметить, что в Visual Studio есть специальный компонент для добавления нового интерфейса в отдельном файле. Для добавления интерфейса в проект можно нажать правой кнопкой мыши на проект и в появившемся контекстном меню выбрать **Add-> New Item...** и в диалоговом окне добавления нового компонента выбрать пункт **Interface**:

При реализации интерфейса учитываются также методы и свойства, унаследованные от базового класса. Например:

```
1   interface IAction
2   {
3       void Move();
4   }
5   class BaseAction
6   {
7       public void Move()
8       {
9           Console.WriteLine("Move in BaseAction");
10      }
11  }
12  class HeroAction : BaseAction, IAction
13  {
14  }
```

Здесь класс HeroAction реализует интерфейс IAction, однако для реализации метода Move из интерфейса применяется метод Move, унаследованный от базового класса BaseAction. Таким образом, класс HeroAction может не реализовать метод Move, так как этот метод уже определен в базовом классе BaseAction.

Следует отметить, что если класс одновременно наследует другой класс и реализует интерфейс, как в примере выше класс HeroAction, то название базового класса должно быть указано до реализуемых интерфейсов: class HeroAction : **BaseAction**, IAction

Множественная реализация интерфейсов

Интерфейсы имеют еще одну важную функцию: в C# не поддерживается множественное наследование, то есть мы можем унаследовать класс только от одного класса, в отличие, скажем, от языка C++, где множественное наследование можно использовать. Интерфейсы позволяют частично обойти это ограничение, поскольку в C# класс может реализовать сразу несколько интерфейсов. Все реализуемые интерфейсы указываются через запятую:

```
myClass: myInterface1, myInterface2, myInterface3, ...
{

}
```

Рассмотрим на примере:

```
using System;

namespace HelloApp
{
    interface IAccount
    {
        int CurrentSum { get; }  // Текущая сумма на счету
        void Put(int sum);      // Положить деньги на счет
        void Withdraw(int sum); // Взять со счета
    }
    interface IClient
    {
        string Name { get; set; }
    }
    class Client : IAccount, IClient
    {
        int _sum; // Переменная для хранения суммы
```

```
18          public string Name { get; set; }
19          public Client(string name, int sum)
20          {
21              Name = name;
22              _sum = sum;
23          }
24
25          public int CurrentSum { get { return _sum; } }
26
27          public void Put(int sum) {  _sum += sum; }
28
29          public void Withdraw(int sum)
30          {
31              if (_sum >= sum)
32              {
33                  _sum -= sum;
34              }
35          }
36      }
37      class Program
38      {
39          static void Main(string[] args)
40          {
41              Client client = new Client("Tom", 200);
42              client.Put(30);
43              Console.WriteLine(client.CurrentSum); //230
44              client.Withdraw(100);
45              Console.WriteLine(client.CurrentSum); //130
46              Console.Read();
47          }
48      }
49  }
```

В данном случае определены два интерфейса. Интерфейс IAccount определяет свойство CurrentSum для текущей суммы денег на счете и два метода Put и Withdraw для добавления денег на счет и изъятия денег. Интерфейс IClient определяет свойство для хранения имени клиента.

Обатите внимание, что свойства CurrentSum и Name в интерфейсах похожи на автосвойства, но это не автосвойства. При реализации мы можем развернуть их в полноценные свойства, либо же сделать автосвойствами.

Класс Client реализует оба интерфейса и затем применяется в программе.

Интерфейсы в преобразованиях типов

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Поскольку класс Client реализует интерфейс IAccount, то переменная типа IAccount может хранить ссылку на объект типа Client:

```
// Все объекты Client являются объектами IAccount
IAccount account = new Client("Том", 200);
account.Put(200);
Console.WriteLine(account.CurrentSum); // 400
// Не все объекты IAccount являются объектами Client,
необходимо явное приведение
Client client = (Client)account;
// Интерфейс IAccount не имеет свойства Name, необходимо
 явное приведение
string clientName = ((Client)account).Name;
```

Преобразование от класса к его интерфейсу, как и преобразование от производного типа к базовому, выполняется автоматически. Так как любой объект Client реализует интерфейс IAccount.

Обратное преобразование - от интерфейса к реализующему его классу будет аналогично преобразованию от базового класса к производному. Так как не каждый объект IAccount является объектом Client (ведь интерфейс IAccount могут реализовать и другие классы), то для подобного преобразования необходима операция приведения типов. И если мы хотим обратиться к методам класса Client, которые не определены в интерфейсе IAccount, но являются частью класса Client, то нам надо явным образом выполнить преобразование типов: string clientName = ((Client)account).Name;

Список List<T>

Класс List<T> представляет простейший список однотипных объектов.

Среди его методов можно выделить следующие:

- **void Add(T item)**: добавление нового элемента в список
- **void AddRange(ICollection collection)**: добавление в список коллекции или массива
- **int BinarySearch(T item)**: бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.

- **int IndexOf(T item)**: возвращает индекс первого вхождения элемента в списке
- **void Insert(int index, T item)**: вставляет элемент item в списке на позицию index
- **bool Remove(T item)**: удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true
- **void RemoveAt(int index)**: удаление элемента по указанному индексу index
- **void Sort()**: сортировка списка

Посмотрим реализацию списка на примере:

```
1   using System;
2   using System.Collections.Generic;
3
4   namespace Collections
5   {
6       class Program
7       {
8           static void Main(string[] args)
9           {
10              List<int> numbers = new List<int>() { 1, 2, 3, 45 };
11              numbers.Add(6); // добавление элемента
12
13              numbers.AddRange(new int[] { 7, 8, 9 });
14
15              numbers.Insert(0, 666); // вставляем на первое место в списке число 666
16
17              numbers.RemoveAt(1); //  удаляем второй элемент
18
19              foreach (int i in numbers)
20              {
21                  Console.WriteLine(i);
22              }
23
24              List<Person> people = new List<Person>(3);
25              people.Add(new Person() { Name = "Том" });
26              people.Add(new Person() { Name = "Билл" });
27
28              foreach (Person p in people)
29              {
30                  Console.WriteLine(p.Name);
31              }
32
33              Console.ReadLine();
```

```
34          }
35      }
36
37      class Person
38      {
39          public string Name { get; set; }
40      }
41  }
```

Здесь у нас создаются два списка: один для объектов типа int, а другой - для объектов Person. В первом случае мы выполняем начальную инициализацию списка: List<int> numbers = new List<int>() { 1, 2, 3, 45 };

Во втором случае мы используем другой конструктор, в который передаем начальную емкость списка: List<Person> people = new List<Person>(3);. Указание начальной емкости списка (capacity) позволяет в будущем увеличить производительность и уменьшить издержки на выделение памяти при добавлении элементов. Также начальную емкость можно установить с помощью свойства Capacity, которое имеется у класса List.

Двухсвязный список LinkedList<T>

Класс LinkedList<T> представляет двухсвязный список, в котором каждый элемент хранит ссылку одновременно на следующий и на предыдущий элемент.

Если в простом списке List<T> каждый элемент представляет объект типа T, то в LinkedList<T> каждый узел представляет объект класса LinkedListNode<T>. Этот класс имеет следующие свойства:

- **Value**: само значение узла, представленное типом T
- **Next**: ссылка на следующий элемент типа LinkedListNode<T> в списке. Если следующий элемент отсутствует, то имеет значение null
- **Previous**: ссылка на предыдущий элемент типа LinkedListNode<T> в списке. Если предыдущий элемент отсутствует, то имеет значение null

Используя методы класса LinkedList<T>, можно обращаться к различным элементам, как в конце, так и в начале списка:

- **AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode)**: вставляет узел newNode в список после узла node.
- **AddAfter(LinkedListNode<T> node, T value)**: вставляет в список новый узел со значением value после узла node.

- **AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode)**: вставляет в список узел newNode перед узлом node.
- **AddBefore(LinkedListNode<T> node, T value)**: вставляет в список новый узел со значением value перед узлом node.
- **AddFirst(LinkedListNode<T> node)**: вставляет новый узел в начало списка
- **AddFirst(T value)**: вставляет новый узел со значением value в начало списка
- **AddLast(LinkedListNode<T> node)**: вставляет новый узел в конец списка
- **AddLast(T value)**: вставляет новый узел со значением value в конец списка
- **RemoveFirst()**: удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным
- **RemoveLast()**: удаляет последний узел из списка

Посмотрим на двухсвязный список в действии:

```
using System;
using System.Collections.Generic;

namespace Collections
{
    class Program
    {
        static void Main(string[] args)
        {
            LinkedList<int> numbers = new LinkedList<int>();

            numbers.AddLast(1); // вставляем узел со значением 1 на последнее
место
            // так как в списке нет узлов, то последнее будет также и первым
            numbers.AddFirst(2); // вставляем узел со значением 2 на первое
место
            numbers.AddAfter(numbers.Last, 3); // вставляем после последнего
узла новый узел со значением 3
            // теперь у нас список имеет следующую последовательность: 2,
1, 3
            foreach (int i in numbers)
            {
                Console.WriteLine(i);
            }

            LinkedList<Person> persons = new LinkedList<Person>();

```

```
28          // добавляем persona в список и получим объект
29  LinkedListNode<Person>, в котором хранится имя Tom
30          LinkedListNode<Person> tom = persons.AddLast(new Person() {
31  Name = "Tom" });
32          persons.AddLast(new Person() { Name = "John" });
33          persons.AddFirst(new Person() { Name = "Bill" });
34
35          Console.WriteLine(tom.Previous.Value.Name); // получаем узел
36  перед томом и его значение
37          Console.WriteLine(tom.Next.Value.Name); // получаем узел после
38  тома и его значение
39
40          Console.ReadLine();
        }
    }

    class Person
    {
        public string Name { get; set; }
    }
}
```

Здесь создаются и используются два списка: для чисел и для объектов класса Person. С числами, наверное, все более менее понятно. Разберем работу с классом Person.

Методы вставки (AddLast, AddFirst) при добавлении в список возвращают ссылку на добавленный элемент LinkedListNode<T> (в нашем случае LinkedListNode<Person>). Затем управляя свойствами Previous и Next, мы можем получить ссылки на предыдущий и следующий узлы в списке.

8.Коллекции

Введение в коллекции

Хотя в языке C# есть массивы, которые хранят в себе наборы однотипных объектов, но работать с ними не всегда удобно. Например, массив хранит фиксированное количество объектов, однако что если мы заранее не знаем, сколько нам потребуется объектов. И в этом случае намного удобнее применять коллекции. Еще один плюс коллекций состоит в том, что некоторые из них реализует стандартные структуры данных, например, стек, очередь, словарь, которые могут пригодиться для решения различных специальных задач.

Большая часть классов коллекций содержится в пространствах имен System.Collections (простые необобщенные классы коллекций),System.Collections.Generic (обобщенные или типизированные классы коллекций) и System.Collections.Specialized (специальные классы коллекций). Также для обеспечения параллельного выполнения задач и многопоточного доступа применяются классы коллекций из пространства имен System.Collections.Concurrent

Основой для создания всех коллекций является реализация интерфейсов **IEnumerator** и **IEnumerable** (и их обобщенных двойников**IEnumerator<T>** и **IEnumerable<T>**). Интерфейс IEnumerator представляет перечислитель, с помощью которого становится возможен последовательный перебор коллекции, например, в цикле **foreach**. А интерфейс IEnumerable через свой метод GetEnumerator предоставляет перечислитель всем классам, реализующим данный интерфейс. Поэтому интерфейс IEnumerable (IEnumerable<T>) является базовым для всех коллекций.

Рассмотрим создание и применение двух коллекций:

```
1    using System;
2    using System.Collections;
3    using System.Collections.Generic;
4
5    namespace Collections
6    {
7        class Program
8        {
9            static void Main(string[] args)
10           {
11               // необобщенная коллекция ArrayList
```

```
12          ArrayList objectList = new ArrayList() { 1, 2, "string", 'c', 2.0f };
13
14          object obj = 45.8;
15
16          objectList.Add(obj);
17          objectList.Add("string2");
18          objectList.RemoveAt(0); // удаление первого элемента
19          foreach (object o in objectList)
20          {
21              Console.WriteLine(o);
22          }
23          Console.WriteLine("Общее число элементов коллекции: {0}",
24  objectList.Count);
25          // обобщенная коллекция List
26          List<string> countries = new List<string>() { "Россия", "США",
27  "Великобритания", "Китай" };
28          countries.Add("Франция");
29          countries.RemoveAt(1); // удаление второго элемента
30          foreach (string s in countries)
31          {
32              Console.WriteLine(s);
33          }
34
35          Console.ReadLine();
36      }
    }
}
```

Здесь используются две коллекции: необобщенная - ArrayList и обобщенная - List. Большинство коллекций поддерживают добавление элементов. Например, в данном случае добавление производится методом Add, но для других коллекций название метода может отличаться. Также большинство коллекций реализуют удаление (в данном примере производится с помощью метода RemoveAt).

С помощью свойства Count у коллекций можно посмотреть количество элементов.

И так как коллекции реализуют интерфейс IEnumerable/IEnumerable<T>, то все они поддерживают перебор в цикле foreach.

Конкретные методы и способы использования могут различаться от одного класса коллекции к другому, но общие принципы будут одни и те же для всех классов коллекций.

Интерфейсы IEnumerable и IEnumerator

Как мы увидели, основной для большинства коллекций является реализация интерфейсов IEnumerable и IEnumerator. Благодаря такой реализации мы можем перебирать объекты в цикле foreach:

```
1  foreach(var item in перечислимый_объект)
2  {
3
4  }
```

Перебираемая коллекция должна реализовать интерфейс IEnumerable.

Интерфейс IEnumerable имеет метод, возвращающий ссылку на другой интерфейс - перечислитель:

```
1  public interface IEnumerable
2  {
3      IEnumerator GetEnumerator();
4  }
```

А интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере:

```
1  public interface IEnumerator
2  {
3      bool MoveNext(); // перемещение на одну позицию вперед в
   контейнере элементов
4      object Current {get;}  // текущий элемент в контейнере
5      void Reset(); // перемещение в начало контейнера
6  }
```

Метод **MoveNext()** перемещает указатель на текущий элемент на следующую позицию в последовательности. Если последовательность еще не закончилась, то возвращает true. Если же последовательность закончилась, то возвращается false.

Свойство **Current** возвращает объект в последовательности, на который указывает указатель.

Метод **Reset()** сбрасывает указатель позиции в начальное положение.

Каким именно образом будет осуществляться перемещение указателя и получение элементов зависит от реализации интерфейса. В различных реализациях логика может быть построена различным образом.

Например, без использования цикла foreach перебирем коллекцию с помощью интерфейса IEnumerator:

```
1   using System;
2   using System.Collections;
3
4   namespace HelloApp
5   {
6       class Program
7       {
8           static void Main(string[] args)
9           {
10              int[] numbers = { 0, 2, 4, 6, 8, 10 };
11
12              IEnumerator ie = numbers.GetEnumerator(); // получаем IEnumerator
13              while (ie.MoveNext())   // пока не будет возвращено false
14              {
15                  int item = (int)ie.Current;     // берем элемент на текущей позиции
16                  Console.WriteLine(item);
17              }
18              ie.Reset(); // сбрасываем указатель в начало массива
19              Console.Read();
20          }
21      }
22  }
```

Реализация IEnumerable и IEnumerator

Рассмотрим простешую реализацию **IEnumerable** на примере:

```
1   using System;
2   using System.Collections;
3
4   namespace HelloApp
5   {
6       class Week : IEnumerable
7       {
8           string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",
9                             "Friday", "Saturday", "Sunday" };
10
11          public IEnumerator GetEnumerator()
```

```
12          {
13              return days.GetEnumerator();
14          }
15      }
16      class Program
17      {
18          static void Main(string[] args)
19          {
20              Week week = new Week();
21              foreach(var day in week)
22              {
23                  Console.WriteLine(day);
24              }
25              Console.Read();
26          }
27      }
28  }
```

В данном случае класс Week, который представляет неделю и хранит все дни недели, реализует интерфейс IEnumerable. Однако в данном случае мы поступили очень просто - вместо реализации IEnumerator мы просто возвращаем в методе GetEnumerator объект IEnumerator для массива.

```
1   public IEnumerator GetEnumerator()
2   {
3       return days.GetEnumerator();
4   }
```

Благодаря этому мы можем перебрать все дни недели в цикле foreach.

В то же время стоит отметить, что для перебора коллекции через foreach в принципе необязательно реализовать интерфейс IEnumerable. Достаточно в классе определить публичный метод **GetEnumerator**, который бы возвращал объект IEnumerator. Например:

```
1   class Week
2   {
3       string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",
4                         "Friday", "Saturday", "Sunday" };
5
6       public IEnumerator GetEnumerator()
7       {
8           return days.GetEnumerator();
9       }
10  }
```

Однако это было довольно просто - мы просто используем уже готовый перчислитель массива. Однако, возможно, потребуется задать свою собственную логику перебора объектов. Для этого реализуем интерфейс **IEnumerator**:

```
using System;
using System.Collections;

namespace HelloApp
{
    class WeekEnumerator : IEnumerator
    {
        string[] days;
        int position = -1;
        public WeekEnumerator(string[] days)
        {
            this.days = days;
        }
        public object Current
        {
            get
            {
                if (position == -1 || position >= days.Length)
                    throw new InvalidOperationException();
                return days[position];
            }
        }

        public bool MoveNext()
        {
            if(position < days.Length - 1)
            {
                position++;
                return true;
            }
            else
                return false;
        }

        public void Reset()
        {
            position = -1;
        }
    }
```

```
40      class Week
41      {
42          string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",
43                            "Friday", "Saturday", "Sunday" };
44
45          public IEnumerator GetEnumerator()
46          {
47              return new WeekEnumerator(days);
48          }
49      }
50      class Program
51      {
52          static void Main(string[] args)
53          {
54              Week week = new Week();
55              foreach(var day in week)
56              {
57                  Console.WriteLine(day);
58              }
59              Console.Read();
60          }
61      }
62  }
```

Здесь теперь класс Week использует не встроенный перечислитель, а WeekEnumerator, который реализует IEnumerator.

Ключевой момент при реализации перечислителя - перемещения указателя на элемент. В классе WeekEnumerator для хранения текущей позиции определена переменная position. Следует учитывать, что в самом начале (в исходном состоянии) указатель должен указывать на позицию условно перед первым элементом. Когда будет производиться цикл foreach, то данный цикл вначале вызывает метод MoveNext и фактически перемещает указатель на одну позицию в перед и только затем обращается к свойству Current для получения элемента в текущей позиции.

В примерах выше использовались необобщенные версии интерфейсов, однако мы также можем использовать их обобщенные двойники:

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace HelloApp
{
```

```
class Week
{
   string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",
                     "Friday", "Saturday", "Sunday" };

   public IEnumerator<string> GetEnumerator()
   {
      return new WeekEnumerator(days);
   }
}
class WeekEnumerator : IEnumerator<string>
{
   string[] days;
   int position = -1;
   public WeekEnumerator(string[] days)
   {
      this.days = days;
   }

   public string Current
   {
      get
      {
         if (position == -1 || position >= days.Length)
            throw new InvalidOperationException();
         return days[position];
      }
   }

   object IEnumerator.Current => throw new NotImplementedException();

   public bool MoveNext()
   {
      if(position < days.Length - 1)
      {
         position++;
         return true;
      }
      else
         return false;
   }

   public void Reset()
   {
```

```
            position = -1;
        }
        public void Dispose() { }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Week week = new Week();
            foreach(var day in week)
            {
                Console.WriteLine(day);
            }
            Console.Read();
        }
    }
}
```

Итераторы и оператор yield

Итератор по сути представляет блок кода, который использует оператор **yield** для перебора набора значений. Данный блок кода может представлять тело метода, оператора или блок get в свойства.

Итератор использует две специальных инструкции:

- **yield return**: определяет возвращаемый элемент
- **yield break**: указывает, что последовательность больше не имеет элементов

Рассмотрим небольшой пример:

```
1   using System;
2   using System.Collections;
3
4   namespace HelloApp
5   {
6       class Program
7       {
8           static void Main(string[] args)
9           {
10              Numbers numbers = new Numbers();
11              foreach (int n in numbers)
```

```
            {
                Console.WriteLine(n);
            }
            Console.ReadKey();
        }
    }

    class Numbers
    {
        public IEnumerator GetEnumerator()
        {
            for(int i = 0; i < 6; i++)
            {
                yield return i * i;
            }
        }
    }
```

В классе Numbers метод **GetEnumerator()** фактически представляет итератор. С помощью оператора **yield return** возвращается некоторое значение (в данном случае квадрат числа).

В программе с помощью цикла foreach мы можем перебрать объект Numbers как обычную коллекцию. При получении каждого элемента в цикле foreach будет срабатывать оператор yield return, который будет возвращать один элемент и запоминать текущую позицию.

Другой пример: пусть у нас есть коллекция Library, которая представляет хранилище книг - объектов Book. Используем оператор yield для перебора этой коллекции:

```
class Book
{
    public Book(string name)
    {
        this.Name = name;
    }
    public string Name { get; set; }
}

class Library
{
    private Book[] books;

    public Library()
    {
        books = new Book[] { new Book("Отцы и дети"), new Book("Война и
```

```
17    мир"),
18                  new Book("Евгений Онегин") };
19        }
20
21        public int Length
22        {
23            get { return books.Length; }
24        }
25
26        public IEnumerator GetEnumerator()
27        {
28            for (int i = 0; i < books.Length; i++)
29            {
30                yield return books[i];
31            }
32        }
      }
```

Метод GetEnumerator() представляет итератор. И когда мы будем осуществлять перебор в объекте Library в цикле foreach, то будет идти к обращение к вызову yield return books[i];. При обращении к оператору yield return будет сохраняться текущее местоположение. И когда метод foreach перейдет к следующей итерации для получения нового объекта, итератор начнет выполнения с этого местоположения.

Ну и в основной программе в цикле foreach выполняется собственно перебор, благодаря реализации итератора:

```
1    foreach (Book b in library)
2    {
3        Console.WriteLine(b.Name);
4    }
```

Хотя при реализации итератора в методе GetEnumerator() применялся перебор массива в цикле for, но это необязательно делать. Мы можем просто определить несколько вызовов оператора yield return:

```
1    IEnumerator IEnumerable.GetEnumerator()
2    {
3        yield return books[0];
4        yield return books[1];
5        yield return books[2];
6    }
```

В этом случае при каждом вызове оператора yield return итератор также будет запоминать текущее местоположение и при последующих вызовах начинать с него.

Именованный итератор

Выше для создания итератора мы использовали метод GetEnumerator. Но оператор yield можно использовать внутри любого метода, только такой метод должен возвращать объект интерфейса IEnumerable. Подобные методы еще называют **именованными итераторами**.

Создадим такой именованный итератор в классе Library и используем его:

```
1   class Book
2   {
3       public Book(string name)
4       {
5           this.Name=name;
6       }
7       public string Name { get; set; }
8   }
9
10  class Library
11  {
12      private Book[] books;
13
14      public Library()
15      {
16          books = new Book[] { new Book("Отцы и дети"), new Book("Война и
17  мир"),
18              new Book("Евгений Онегин") };
19      }
20
21      public int Length
22      {
23          get { return books.Length; }
24      }
25
26      public IEnumerable GetBooks(int max)
27      {
28          for (int i = 0; i < max; i++)
29          {
```

```
            if (i == books.Length)
            {
                yield break;
            }
            else
            {
                yield return books[i];
            }
        }
    }
}
```

Определенный здесь итератор - метод IEnumerable GetBooks(int max) в качестве параметра принимает количество выводимых объектов. В процессе работы программы может сложиться, что его значение будет больше, чем длина массива books. И чтобы не произошло ошибки, используется оператор **yield break**. Этот оператор прерывает выполнение итератора.

Применение итератора:

```
Library library = new Library();

foreach (Book b in library.GetBooks(5))
{
    Console.WriteLine(b.Name);
}
```

Вызов library.GetBooks(5) будет возвращать набор из не более чем 5 объектов Book. Но так как у нас всего три таких объекта, то в методе GetBooks после трех операций сработает оператор yield break.

Заключение

Предмет программной инженерии учит синтаксису программной инженерии и работе на необходимых платформах для создания практических программ на нем, для создания программного обеспечения произвольной сложности, для создания современного распределенного программного обеспечения. Далее в рамках этого курса студенты изучают технологии программирования для работы с Entity Framework, MVC 5 и проектами в них и получают возможность эффективно использовать их в практической работе, исследованиях, а также в системе образования.

Использованная литература

1. Roger Pressman, Bruce Maxim, Software Engineering: A Practitioner's Approach, John Wiley & Sons, USA 2014.
2. Ian Sommerville. Software Engineering Hardcover. Pearson 2010 USA
3. Robert W. Sebesta, Concepts of Programming Languages, John Wiley & Sons, USA 2015.
4. Fundamentals of Computer Programming With C# (The Bulgarian C# Programming Book). Svetlin Nakov & Co., 2013.
5. Шилдт, Герберт. C# 4.0: полное руководство. : Пер. с англ. — М. : ООО "И.Д. Вильяме", 2011.

O‘ZBEKISTON RESPUBLIKASI
OLIY VA O‘RTA MAXSUS
TA’LIM VAZIRLIGI

GUVOHNOMA

O‘QUV ADABIYOTINING NASHR RUXSATNOMASI

O‘zbekiston Respublikasi Oliy va o‘rta maxsus ta’lim vazirligining 2022 yil “ 9 ” sentabr dagi “ 302 ” -sonli buyrug‘iga asosan

D.T.Muxamedieva, A.T.Varlamova S.A.Baxromov
(muallifning familiyasi, ismi-sharifi)

5330100-Axborot tizimlarining matematik va dasturiy ta’minoti

(ta’lim yo’nalishi (mutaxassisligi))

ning

talabalari (o‘quvchilari) uchun tavsiya etilgan

Программная инженерия C#-основы программирования
(o’quv adabiyotining nomi va turi: darslik, o’quv qo’llanma)

nomli o‘quv qo‘llanmasi

ga

O‘zbekiston Respublikasi Vazirlar Mahkamasi tomonidan litsenziya berilgan nashriyotlarda nashr etishga ruxsat berildi.

M.O‘

Vazir _____ A.Toshkulov
(imzo)

Ro‘yxatga olish raqami

302-0382