Рязанский станкостроительный колледж РГРТУ

МДК.01.01 Разработка программных модулей

Тема 3. Объектно-ориентированное программирование

Рязань 2020

Оглавление

Базовые принципы ООП 4
Классы и объекты 5
Описание класса и создание объекта 5
Общая форма определения класса 5
Определение класса и создание объектов 6
Методы 8
Конструкторы 9
Деструкторы 10
Свойства 11
Автоматические свойства 13
Перегрузка методов 13
Статические члены и модификатор static 14
Статические классы 16
Практическая работа №5 17
Перегрузка операторов 20
Основы перегрузки операторов 20
Перегрузка бинарных операторов 21
Перегрузка унарных операторов 22
Перегрузка операторов true и false 22
Перегружаемые операторы 23
Практическая работа №6 24
Наследование. Иерархия классов 26
Основы наследования 26
Доступ к членам базового класса из производного класса 27
Конструкторы в производных классах 27
Практическая работа №7 29
Интерфейсы и наследование 31
Общие сведения об интерфейсах 31
Определение интерфейса 31
Применение интерфейса 32
Применение интерфейсных ссылок 34
Стандартные интерфейсы 35
Сравнение объектов (интерфейс iComparable) 35
Клонирование объектов (интерфейс iCloneable) 36
Практическая работа №8 38
Структуры 40
Описание структур 40
Применение структур 40
О назначении структур 41
Практическая работа №9 43
Делегаты 46
Описание и использование делегатов 46
Добавление методов в делегат 48
Применение делегатов 49
Коллекции 50
Общие сведения 50
Интерфейсы коллекций 50
Необобщенные коллекции 50
Обобщенные коллекции 51

Параллельные коллекции 51
Необобщенная коллекция ArrayList 52
Обобщение 53
Обобщенные классы 53
Обобщенные методы 55
Обобщенная коллекция List<T> 56
Обобщенная коллекция Queue<T> 57
Обобщенная коллекция Stack<T> 58
Обобщенная коллекция Dictonary<TKey, TValue> 60
Практическая работа №10 62
Регулярные выражения 63
Общие сведения 63
Синтаксис регулярных выражений 65
Метасимволы 65
Экранирование 66
Квантификаторы 67
Метасимволы привязки 68
Группирование 69
Обратные ссылки 70
Чередование и оператор ИЛИ 70
Примеры использования регулярных выражений 71
Практическая работа №11 73

Базовые принципы ООП

Как мы уже знаем, язык программирования C# — полностью объектно - ориентированный. Это означает, что в нем реализована парадигма ООП. В чем же особенность ООП? Важно понимать, что речь идет о способе организации программы.

На сегодня существует два основных способа организации программного кода. Это объектно - ориентированное и процедурное (структурное) программирование. Попробуем разобраться, чем отличаются эти концепции программирования.

При процедурном программировании программа реализуется как набор подпрограмм (процедуры и функции), используемых для обработки данных. То есть данные, используемые в программе, и программный код, применяемый для обработки этих данных, существуют отдельно друг от друга. В процессе выполнения программы выбираются нужные данные и обрабатываются с помощью специальных процедур и функций.

Основная задача ООП — сделать сложный код проще. Для этого программу разбивают на независимые блоки, которые мы называем объектами.

Объект — это не какая-то космическая сущность. Это всего лишь набор данных и функций — таких же, как в традиционном функциональном программировании. Можно представить, что просто взяли кусок программы и положили его в коробку и закрыли крышку. Вот эта коробка с крышками — это объект.

Программисты договорились, что данные внутри объекта будут называться свойствами, а функции — методами. Но это просто слова, по сути это те же переменные и функции.

Объект можно представить как независимый электроприбор у вас на кухне. Чайник кипятит воду, плита греет, блендер взбивает, мясорубка делает фарш. Внутри каждого устройства куча всего: моторы, контроллеры, кнопки, пружины, предохранители — но вы о них не думаете. Вы нажимаете кнопки на панели каждого прибора, и он делает то, что от него ожидается. И благодаря совместной работе этих приборов у вас получается ужин.

Таким образом, ООП — это способ организации программы через взаимодействие отдельных объектов, содержащих данные и методы для работы с этими данными. Обычно в ООП выделяют три базовых принципа:

- инкапсуляция;
- наследование;
- полиморфизм.

Любой объектно - ориентированный язык содержит средства для реализации этих принципов. Способы реализации могут отличаться, но принципы неизменны.

Инкапсуляция означает, что данные объединяются в одно целое с программным кодом, предназначенным для их обработки. Фактически организация программы через взаимодействие объектов является реализацией принципа инкапсуляции. На программном уровне инкапсуляция реализуется путем использования *классов* и *объектов* (объекты создаются на основе классов).

Наследование позволяет создавать классы на основе уже существующих классов. Соответственно существует базовый класс (предок) и производный класс (потомок). Производный класс несет в себе характеристики своего базового класса (содержит те же данные и методы), а также обладает собственными характеристиками. При этом наследуемые данные и методы описывать у потомка нет необходимости.

Полиморфизм подразумевает использование единого интерфейса для решения однотипных задач. Проявлением полиморфизма является тот факт, что нередко в программе один и тот же метод можно вызывать с разными аргументами. Это удобно.

Проявлением полиморфизма является перегрузка и переопределение методов. Также к полиморфизму относится способность объекта использовать методы производного класса, который не существует на момент создания базового.

Классы и объекты

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является ***класс***, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

Аналогии — это хорошо. Но что же такое объект, если речь идет об использовании его в программе? Скажем, мы уже знаем, что за переменной «скрывается» область в памяти. В эту область можно записывать значения и считывать значение оттуда. И для получения доступа к этой памяти достаточно знать имя переменной. Объект во многом похож на переменную, но только он более «разноплановый». Объект — это группа переменных, причем в общем случае разного типа. А еще объект — это набор методов. Метод, в свою очередь — это группа инструкций, которые можно выполнить (вызвав метод).

В итоге получается, что если обычная переменная напоминает коробку со значением внутри, то объект — это большая коробка, в которой есть коробки поменьше (переменные) и всякие пружинки и рычажки (наборы инструкций, реализованные в виде методов). С программной точки зрения объект реализуется как некоторая область памяти, содержащая переменные и методы.

Программная «начинка» объекта состоит из переменных и методов. Переменные называются полями объекта. Объекты, как мы помним, создаются на основе класса. Когда мы описываем класс, мы фактически определяем, какие поля и методы будут у объектов, создаваемых на основе класса.

Поля метода можно использовать как обычные переменные. Методы объекта можно вызывать. В этом случае выполняются инструкции, содержащиеся в теле метода. Метод, который вызывается из некоторого объекта, автоматически получает доступ к полям (и другим методам) этого объекта.

Описание класса и создание объекта

Общая форма определения класса

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

```
class имя_класса {
// Объявление переменных (полей) класса.
доступ тип переменная1;
доступ тип переменная2;
//...
```

```
доступ тип переменнаяN ;
// Объявление методов (функций).
доступ возвращаемый_тип метод1 (параметры)
{
        // тело метода
}
доступ возвращаемый_тип метод2 (параметры)
{
        // тело метода
}
//. . .
доступ возвращаемый_тип методы (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. Указывать спецификатор доступа не обязательно, но если он отсутствует, то объявляемый член считается закрытым в пределах класса..

Несмотря на отсутствие соответствующего правила в синтаксисе C#, правильно сконструированный класс должен определять одну и только одну логическую сущность. Например, класс, в котором хранятся Ф.И.О. и номера телефонов, обычно не содержит сведения о фондовом рынке, среднем уровне осадков, циклах солнечных пятен или другую информацию, не связанную с перечисляемыми фамилиями. Таким образом, в правильно сконструированном классе должна быть сгруппирована логически связанная информация.

Определение класса и создание объектов

Для того чтобы продемонстрировать классы на конкретных примерах, разработаем постепенно класс, инкапсулирующий информацию о зданиях, в том числе о домах, складских помещениях, учреждениях и т.д. В этом классе (назовем его ***Building***) будут храниться три элемента информации о зданиях: количество этажей, общая площадь и количество жильцов. Ниже приведен первый вариант класса ***Building***. В нем определены три переменные экземпляра: ***Floors, Area и Occupants***. Как видите, в классе ***Building*** вообще отсутствуют методы. Это означает, что в настоящий момент этот класс состоит только из данных.

```
class Building
{
    public int Floors; // количество этажей public
    public int Area; // общая площадь здания
    public int Occupants; // количество жильцов
}
```

Для того чтобы создать конкретный объект типа Building, придется воспользоваться следующим оператором.

Building house = new Building(); // создать объект типа Building

После выполнения этого оператора объект house станет экземпляром класса Building, т.е. обретет "физическую" реальность.

Всякий раз, когда получается экземпляр класса, создается также объект, содержащий собственную копию каждой переменной экземпляра, определенной в данном классе. Таким образом, каждый объект типа Building будет содержать свои копии переменных экземпляра Floors, Area и Occupants. Для доступа к этим переменным служит оператор доступа к члену класса, который принято называть оператором-точкой. Оператор-точка связывает имя объекта с именем члена класса. Ниже приведена общая форма оператора-точки.

объект.член

В этой форме объект указывается слева, а член — справа. Например, присваивание значения 2 переменной Floors объекта house осуществляется с помощью следующего оператора.

house.Floors = 2;

В целом, оператор-точка служит для доступа к переменным экземпляра и методам. Ниже приведен полноценный пример программы, в которой используется класс ***Building***.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Building house = new Building(); // создать объект типа Building
    int areaPP; // площадь на одного человека
                // Присвоить значения полям в объекте house,
    house.Occupants = 4;
    house.Area = 2500;
    house.Floors = 2;
    // Вычислить площадь на одного человека.
    areaPP = house.Area / house.Occupants;
    TextBox1.Text="Дом имеет:\n " +
    house.Floors + " этажа\n " +
    house.Occupants + " жильца\n " +
    house.Area +" кв. футов общей площади, из них\n " +
    areaPP + " приходится на одного человека";
}
```

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 ***Building***, то у каждого из них своя копия переменных ***Floors, Area и Occupants***, а их содержимое в обоих объектах может отличаться. Этот факт демонстрируется в следующей программе.

```
Building house = new Building();
Building office = new Building();
int areaPP; // площадь на одного человека
            // Присвоить значения полям в объекте house,
house.Occupants = 4;
house.Area = 2500;
house.Floors = 2;
// Присвоить значения полям в объекте office,
office.Occupants = 25;
office.Area = 4200;
office.Floors = 3;
// Вычислить площадь на одного человека в жилом доме.
areaPP = house.Area / house.Occupants;
TextBox1.Text = "Дом имеет:\n " + house.Floors +
  " этажа\n " + house.Occupants + " жильца\n " +
  house.Area + " кв. футов общей площади, из них\n " +
  areaPP + " приходится на одного человека";
   // Вычислить площадь на одного человека в учреждении.
   areaPP = office.Area / office.Occupants;
 TextBox1.Text = "Учреждение имеет:\n " + office.Floors +
   " этажа\n " + office.Occupants + " работников\n " +
   office.Area + " кв. футов общей площади, из них\n " +
   areaPP + " приходится на одного человека";
```

Методы

Как пояснялось выше, переменные экземпляра и методы являются двумя основными составляющими классов. До сих пор класс ***Building***, рассматриваемый здесь в качестве примера, содержал только данные, но не методы. Хотя классы, содержащие только данные, вполне допустимы, у большинства классов должны быть также методы. Методы представляют собой подпрограммы, которые манипулируют данными, определенными в классе, а во многих случаях они предоставляют доступ к этим данным. Как правило, другие части программы взаимодействуют с классом посредством его методов.

С учетом этого напомним, что в приведенных выше примерах в ***общей программе*** вычислялась площадь на одного человека путем деления общей площади здания на количество жильцов. И хотя такой способ формально считается правильным, на самом деле он оказывается далеко не самым лучшим для организации подобного вычисления. Площадь на одного человека лучше всего вычислять в самом классе ***Building***, просто потому, что так легче понять сам характер вычисления. Ведь площадь на одного человека зависит от значений в полях ***Area и Occupants***, инкапсулированных в классе ***Building***. Следовательно, данное вычисление может быть вполне произведено в самом классе ***Building***. Кроме того, вводя вычисление площади на одного человека в класс ***Building***, мы тем самым избавляем все программы, пользующиеся классом ***Building***, от необходимости выполнять это вычисление самостоятельно. И наконец, добавление в класс ***Building*** метода, вычисляющего площадь на одного человека, способствует улучшению его объектно-ориентированной структуры, поскольку величины, непосредственно связанные со зданием, инкапсулируются в классе ***Building***.

В качестве примера ниже приведен переработанный вариант класса Building, содержащий метод ***AreaPerPerson()***, который выводит площадь, рассчитанную на одного человека в конкретном здании, а также метод для формирования типового вывода информации.

```
class Building
{
    public int Floors; // количество этажей public
    public int Area; // общая площадь здания
    public int Occupants; // количество жильцов

    // Вывести площадь на одного человека,
    Ссылок: 2
    public int AreaPerPerson()
    {
        return Area / Occupants;
    }
    Ссылок: 0
    public string Show()
    {
        int areaPP = AreaPerPerson();
        string info = "Дом имеет:\n " +
            Floors + " этажа\n " +
            Occupants + " жильца\n " +
            Area + " кв. футов общей площади, из них\n " +
            areaPP + " приходится на одного человека";
        return info;
    }
}
```

```
Building house = new Building(); // создать объект типа Building
int areaPP; // площадь на одного человека
            // Присвоить значения полям в объекте house,
house.Occupants = 4;
house.Area = 2500;
house.Floors = 2;
// Вычислить площадь на одного человека.
areaPP = house.AreaPerPerson();
TextBox1.Text=house.Show();
```

Конструкторы

В приведенных выше примерах программ переменные экземпляра каждого объекта типа Building приходилось инициализировать вручную, используя, в частности, следующую последовательность операторов.

house.Occupants = 4;
house.Area = 2500;
house.Floors = 2;

Такой прием обычно не применяется в профессионально написанном коде C#. Кроме того, он чреват ошибками (вы можете просто забыть инициализировать одно из

полей). Впрочем, существует лучший способ решить подобную задачу: воспользоваться конструктором.

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***. А список параметров может быть как пустым, так и состоящим из одного или более указываемых параметров.

У всех классов имеются конструкторы, независимо от того, определите вы их или нет, поскольку в C# автоматически предоставляется конструктор, используемый по умолчанию и инициализирующий все переменные экземпляра их значениями по умолчанию. Для большинства типов данных значением по умолчанию является нулевое, для типа bool — значение false, а для ссылочных типов — пустое значение. Но как только вы определите свой собственный конструктор, то конструктор по умолчанию больше не используется.

Класс Building можно усовершенствовать, добавив в него конструктор, автоматически инициализирующий поля Floors, Area и Occupants при создании объекта.

```
// Параметризированный конструктор для класса Building,
Ссылок: 2
public Building(int initFloors, int initArea, int initOccupants)
{
    Floors = initFloors;
    Area = initArea;
    Occupants = initOccupants;
}


 // Использовать параметризированный конструктор класса Building,
 Building house = new Building(2, 2500, 4);

 int areaPP; // площадь на одного человека

 // Вычислить площадь на одного человека.
 areaPP = house.AreaPerPerson();
 TextBox1.Text=house.Show();
```

Деструкторы

В языке C# имеется возможность определить метод, который будет вызываться непосредственно перед окончательным уничтожением объекта системой "сборки мусора". Такой метод называется деструктором и может использоваться в ряде особых случаев, чтобы гарантировать четкое окончание срока действия объекта. Например, деструктор может быть использован для гарантированного освобождения системного ресурса,

задействованного освобождаемым объектом. Следует, однако, сразу же подчеркнуть, что деструкторы — весьма специфические средства, применяемые только в редких, особых случаях. И, как правило, они не нужны. Но здесь они рассматриваются вкратце ради полноты представления о возможностях языка C#. Ниже приведена общая форма деструктора:

```
~имя_класса ()
{
        // код деструктора
}
```

где имя_класса означает имя конкретного класса. Следовательно, деструктор объявляется аналогично конструктору, за исключением того, что перед его именем указывается знак "тильда" (~). Обратите внимание на то, что у деструктора отсутствуют возвращаемый тип и передаваемые ему аргументы.

Следует, однако, иметь в виду, что деструктор вызывается непосредственно перед "сборкой мусора". Он не вызывается, например, в тот момент, когда переменная, содержащая ссылку на объект, оказывается за пределами области действия этого объекта. В этом отношении деструкторы в C# отличаются от деструкторов в C++, где они вызываются в тот момент, когда объект оказывается за пределами области своего действия. Это означает, что заранее нельзя знать, когда именно следует вызывать деструктор. Кроме того, программа может завершиться до того, как произойдет "сборка мусора", а следовательно, деструктор может быть вообще не вызван.

Свойства

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

Как было показано в приведенных ранее примерах программ, поле зачастую создается, чтобы стать доступным для пользователей объекта, но при этом желательно сохранить управление над операциями, разрешенными для этого поля, например, ограничить диапазон значений, присваиваемых данному полю. Этой цели можно, конечно, добиться и с помощью закрытой переменной, а также методов доступа к ее значению, но свойство предоставляет более совершенный и рациональный путь для достижения той же самой цели.

Стандартное описание свойства имеет следующий синтаксис:

```
Тип-Доступа Тип-Свойства Имя-Свойства {
        get {//Код аксессора для чтения
         return переменная-поле; //– присвоение свойству значения }

        set {//Код аксессора для записи
         переменная-поле = value; //– получаемое от свойства значение}
}
```

где тип свойства обозначает конкретный тип свойства, например int, а имя — присваиваемое свойству имя. Как только свойство будет определено, любое обращение к свойству по имени приведет к автоматическому вызову соответствующего аксессора. Кроме того, аксессор set принимает неявный параметр value, который содержит значение,

присваиваемое свойству. Следует, однако, иметь в виду, что свойства не определяют место в памяти для хранения полей, а лишь управляют доступом к полям. Это означает, что само свойство не предоставляет поле, и поэтому поле должно быть определено независимо от свойства. Исключение из этого правила составляет автоматически реализуемое свойство, рассматриваемое далее.

Пример, использования свойств в классе ***Building***:

```
class Building
{
    int floors; // количество этажей public
    int area; // общая площадь здания
    int occupants; // количество жильцов

    Ссылок: 2
    public int Floors
    {
        get
        {
            return floors;
        }
        set
        {
            floors = value;
        }
    }
```

Здесь у нас есть закрытое поле ***floors*** и есть общедоступное свойство ***Floors***. 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 ***floors***. 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 также как и ранее.

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 количеству этажей, будем считать что этажность здания всегда положительное число:

```
class Building
{
    int floors; // количество этажей public
    int area; // общая площадь здания
    int occupants; // количество жильцов

    Ссылок: 2
    public int Floors
    {
        get
        {
            return floors;
        }
        set
        {
            if (value > 0) floors = value;
            else MessageBox.Show("Этажность здания не может быть меньше нуля");
        }
    }
```

Если бы переменная ***floors*** была бы публичной, то мы могли бы передать ей извне любое значение, в том числе отрицательное. Свойство же позволяет скрыть данные в объекте и опосредовать к ним доступ.

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.

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:

```
class Building
{
    Ссылок: 2
    public int Floors { get; set; } // количество этажей public
    Ссылок: 3
    public int Area { get; set; } // общая площадь здания
    Ссылок: 3
    public int Occupants { get; set; } // количество жильцов
```

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.

Перегрузка методов

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

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

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

Например, опишем класс калькулятор:

```
class Calculator
{
    Ссылок: 0
    public int Add(int a, int b)
    {
        return a + b;
    }
    Ссылок: 0
    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    Ссылок: 0
    public int Add(int a, int b, int c, int d)
    {
        int result = a + b + c + d;
        return result;
    }
    Ссылок: 0
    public double Add(double a, double b)
    {
        return a + b;
    }
}
```

Здесь представлены четыре разных версии метода Add, то есть определены четыре перегрузки данного метода.

Первые три версии метода отличаются по количеству параметров. Четвертая версия совпадает с первой по количеству параметров, но отличается по их типу. При этом достаточно, чтобы хотя бы один параметр отличался по типу. Поэтому это тоже допустимая перегрузка метода Add.

После определения перегруженных версий мы можем использовать их в программе:

```
Calculator calc = new Calculator();
int zn1 = calc.Add(1, 2); // 3
int zn2 = calc.Add(1, 2, 3); // 6
int zn3 = calc.Add(1, 2, 3, 4); // 10
double zn4 = calc.Add(1.4, 2.5); // 3.9
```

Статические члены и модификатор static

Кроме обычных полей, методов, свойств класс может иметь статические поля, методы, свойства. Статические поля, методы, свойства относятся ко всему классу и для обращения к подобным членам класса необязательно создавать экземпляр класса.

Например, описанный выше класс калькулятор, обычно существует в единственном числе, и нет смысла создавать экземпляр класса:

```
class Calculator
{
    ссылка: 1
    static public int Add(int a, int b)
    {
        return a + b;
    }
    ссылка: 1
    static public int Add(int a, int b, int c)
    {
        return a + b + c;
    }
```

После определения класса со статическими методами мы можем использовать их в программе, не создавая экземпляр класса:

```
int zn1 = Calculator.Add(1, 2); // 3
int zn2 = Calculator.Add(1, 2, 3); // 6
int zn3 = Calculator.Add(1, 2, 3, 4); // 10
double zn4 = Calculator.Add(1.4, 2.5); // 3.9
```

Также класс может сочетать как обычные поля, методы, свойства, так иметь и статические поля, методы, свойства. Статические поля, методы, свойства относятся ко всему классу и для обращения к подобным членам класса необязательно создавать экземпляр класса.

Например, создадим класс Account:

```
class Account
{
    public static decimal bonus = 100;
    public decimal totalSum;
    Ссылок: 2
    public Account(decimal sum)
    {
        totalSum = sum + bonus;
    }
}
```

В данном случае класс ***Account*** имеет два поля: ***bonus и totalSum***. Поле ***bonus*** является статическим, поэтому оно хранит состояние класса в целом, а не отдельного объекта. И поэтому мы можем обращаться к этому полю по имени класса:

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

При использовании класса статические элементы могут использоваться без создания экземпляра класса. Обычные элементы используются только с созданным экземпляром класса, см. пример:

```
Account.bonus += 200;
Account account1 = new Account(150);// 450
Account account2 = new Account(1000);// 1300
```

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

Статические классы

Статические классы объявляются с модификатором ***static*** и могут содержать только статические поля, свойства и методы. Например, если бы класс Calculator или Account имел бы только статические переменные, свойства и методы, то его можно было бы объявить как статический. Для статического класса нельзя создать экземпляр объекта.

В C# показательным примером статического класса является класс Math, который применяется для различных математических операций.

Практическая работа №5

Работа с классами. Перегрузка методов.

1. Разработать класс по заданию. Оформить модули комментариями.
2. Разработать программу для демонстрации использования класса и его методов.
3. Предусмотреть в программе две кнопки «**Выход**» и «**О программе**», где вывести ФИО разработчика, номер работы и формулировку задания.
4. Оформить программу комментариями.

Варианты заданий (Базовый уровень)

1. Создать базовый класс Car (машина), характеризуемый торговой маркой (строка), числом цилиндров, мощностью. Создать необходимые методы и свойства. Создать перегруженные методы SetParams, для установки параметров автомобиля.
2. Создать класс Pair (пара чисел). Создать необходимые методы и свойства. Определить методы метод сложения полей и операцию сложения пар (a, b) + (c, d) = (a + c, b + d). Создать перегруженные методы для увеличения полей на 1, сложения трех пар чисел.
3. Создать класс Man (человек), с полями: имя, возраст, пол и вес. Создать необходимые методы и свойства. Создать перегруженные методы SetParams, для установки параметров человека.
4. Создать класс Pair (пара чисел). Создать необходимые методы и свойства. Определить метод сравнения пар: пара p1 больше пары p2, если (first.pl > first.p2) или (first.pl = first.p2) и (second.pl > second.p2). Создать перегруженные методы SetParams, для установки параметров объекта.
5. Создать класс Triangle (треугольник) с полями-сторонами. Создать необходимые методы и свойства. Определить метод вычисления периметра. Создать перегруженные методы SetParams, для установки параметров объекта, в том числе увеличения размеров треугольника в 2 раза.
6. Создать класс Triad (тройка положительных чисел). Создать необходимые методы и свойства. Определить метод вычисления суммы чисел. Создать перегруженные методы SetParams, для установки параметров объекта.
7. Создать класс Triad (тройка чисел). Создать необходимые методы и свойства. Определить метод сравнения триад: триада p1 больше триады p2, если (first.pl > first.p2) или (first.pl = first.p2) и (second.pl > second.p2) и (three.pl > three.p2). Создать перегруженные методы SetParams, для установки параметров объекта, в том числе увеличения всех чисел на 10.
8. Создать класс Pair (пара четных чисел). Создать необходимые методы и свойства. Определить метод вычисления произведения чисел. Создать перегруженный метод для вычисления произведения 2 пар чисел – (a, b) * (c, d) = (a * c) , (b *d).
9. Создать класс Triad (тройка нечетных чисел). Создать необходимые методы и свойства. Определить метод сравнения двух триад на равенство. Создать перегруженный метод сравнения трех триад на равенство.
10. Создать класс Triad (тройка отрицательных чисел). Создать необходимые методы и свойства. Определить метод увеличения полей на 1. Создать перегруженные методы SetParams, для установки параметров объекта, в том числе увеличения всех чисел на 10.
11. Создать класс Pair (пара четных чисел). Создать необходимые методы и свойства. Определить метод перемножения пар (a, b) * (c, d) = (a * c, b * d). Создать перегруженный метод для удвоения пары чисел.

12. Создать класс Triad (тройка чисел). Создать необходимые методы и свойства. Определить методы с операциями сложения с числом, умножения на число, проверки на равенство. Создать перегруженный метод для сложения элементов одой триады с другой триадой.
13. Создать базовый класс Pair (пара целых чисел). Создать необходимые методы и свойства. Определить методы с операциями проверки на равенство и перемножения полей. Реализовать операцию вычитания пар по формуле (a, b) - (c, d) = (a - c, b - d). Создать перегруженный метод для вычитания трех пар чисел.
14. Создать класс Liquid (жидкость), имеющий поля названия, плотности и объема. Создать необходимые методы и свойства. Создать перегруженные методы SetParams, для установки параметров жидкости.
15. Создать класс Triad (тройка положительных чисел). Создать необходимые методы и свойства. Определить метод увеличения полей на заданное число. Создать перегруженный метод для удвоения всех полей.

Пример №1. Создать класс Two (пара чисел). Создать необходимые методы и свойства. Определить методы арифметических операций для чисел и операцию проверки на равенство чисел. Создать перегруженный метод для проверки на равенство двух пар чисел (a, b) и (c, d) = (a = c и b = d)..

```
class Two
{
    Ссылок: 4
    public int One { get; set; }//Первое число
    Ссылок: 4
    public int Second { get; set; }//Второе число

    // Возвращает сумму чисел
    Ссылок: 0
    int Sum()
    {
        return One + Second;
    }

    // Сравниваем числа
    // Возвращает true - числа равны, false в противном случае
    Ссылок: 0
    bool Compare()
    {
        if (One == Second) return true;
        else return false;
    }
    // Сравниваем числа
    // value - сравниваемая пара чисел
    // Возвращает true - числа равны, false в противном случае
    Ссылок: 0
    bool Compare(Two value)
    {
        if (One == value.One && Second == value.Second) return true;
        else return false;
    }
}
```

Перегрузка операторов

Основы перегрузки операторов

Наряду с методами мы можем также перегружать операторы. Например, пусть у нас есть следующий класс ***Counter,*** данный класс представляет некоторый счетчик, значение которого хранится в свойстве ***Value***:

```
class Counter
{
    Ссылок: 0
    public int Value { get; set; }
}
```

Допустим, у нас есть два объекта класса ***Counter*** - два счетчика, которые мы хотим сравнивать или складывать на основании их свойства ***Value***, используя стандартные приемы работы с классом:

```
Counter c1 = new Counter { Value = 23 };
Counter c2 = new Counter { Value = 45 };
Counter c3 = new Counter();
bool result;

//Используем счетчики для сравнения и сложения
if (c1.Value > c2.Value) result = true;
else result = false;
c3.Value = c1.Value + c2.Value;
```

Но удобнее было бы сравнивать или складывать оба счетчика на основании их свойства ***Value***, используя стандартные операции сравнения и сложения:

```
Counter c1 = new Counter { Value = 23 };
Counter c2 = new Counter { Value = 45 };
Counter c3;
bool result;

//А так использовать было бы удобнее
result = c1 > c2;
c3 = c1 + c2;
```

Но на данный момент ни операция сравнения, ни операция сложения для объектов ***Counter*** не доступны. Эти операции могут использоваться для ряда примитивных типов. Например, по умолчанию мы можем складывать числовые значения, но как складывать объекты комплексных типов - классов и структур компилятор не знает. И для этого нам надо выполнить перегрузку нужных нам операторов.

Перегрузка операторов заключается в определении в классе, для объектов которого мы хотим определить оператор, специального метода:

public static возвращаемый_тип operator оператор(параметры)
{ }

Этот метод должен иметь модификаторы ***public static***, так как перегружаемый оператор будет использоваться для всех объектов данного класса. Далее идет название возвращаемого типа. Возвращаемый тип представляет тот тип, объекты которого мы хотим получить. К примеру, в результате сложения двух объектов ***Counter*** мы ожидаем

получить новый объект ***Counter***. А в результате сравнения двух мы хотим получить объект типа ***bool***, который указывает истинно ли условное выражение или ложно. Но в зависимости от задачи возвращаемые типы могут быть любыми.

Затем вместо названия метода идет ключевое слово ***operator*** и собственно сам оператор. И далее в скобках перечисляются параметры. Бинарные операторы принимают два параметра, унарные - один параметр. И в любом случае один из параметров должен представлять тот тип - класс или структуру, в котором определяется оператор.

И еще одно замечание: в параметрах оператора нельзя использовать модификатор ref или out.

Перегрузка бинарных операторов

Бинарные операторы принимают два параметра.

Так как в случае с операцией сложения мы хотим сложить два объекта класса ***Counter***, то оператор принимает два объекта этого класса. И так как мы хотим в результате сложения получить новый объект ***Counter***, то данный класс также используется в качестве возвращаемого типа. Все действия этого оператора сводятся к созданию, нового объекта, свойство ***Value*** которого складывает значения свойства ***Value*** обоих параметров:

```
public static Counter operator +(Counter c1, Counter c2)
{
    Counter result = new Counter();
    result.Value = c1.Value + c2.Value;
    return result;
    //return new Counter { Value = c1.Value + c2.Value };
}
```

Также перегрузим две операции сравнения. Сами операторы сравнения сравнивают значения свойств ***Value*** и в зависимости от результата сравнения возвращают либо ***true***, либо ***false***:

```
public static bool operator >(Counter c1, Counter c2)
{
    bool result;
    if (c1.Value > c2.Value) result = true;
    else result = false;
    return result;
}
Ссылок: 0
public static bool operator <(Counter c1, Counter c2)
{
    return c1.Value < c2.Value;
}
```

Теперь используем перегруженные операторы в программе:

```
Counter c1 = new Counter { Value = 23 };
Counter c2 = new Counter { Value = 45 };
Counter c3;

bool result = c1 > c2;
c3 = c1 + c2;
```

Стоит отметить, что так как по сути определение оператора представляет собой метод, то этот метод мы также можем перегрузить, то есть создать для него еще одну версию. Например, добавим в класс ***Counter*** еще один оператор:

```
public static Counter operator +(Counter c1, int val)
{
    Counter result = new Counter();
    result.Value = c1.Value + val;
    return result;
    //return new Counter { Value = c1.Value + val };
}
```

Данный метод складывает значение свойства ***Value*** и некоторое число, возвращая объект ***Counter*** с их суммой. И также мы можем применить этот оператор:

```
Counter c1 = new Counter { Value = 23 };
Counter c2 = new Counter { Value = 45 };
Counter c3;

bool result = c1 > c2;
c3 = c1 + c2;
c1 = c1 + 5;
```

Перегрузка унарных операторов

Унарные операторы принимают один параметр.

Рассмотрим в качестве примера унарный оператор инкремента:

```
public static Counter operator ++(Counter c1)
{
    Counter result = new Counter();
    result.Value = c1.Value + 1;
    return result;
    //return new Counter { Value = c1.Value + 1 };
}
```

В качестве результата возвращается новый объект, который содержит в свойстве ***Value*** инкрементированное значение.

При этом нам не надо определять отдельно операторы для префиксного и для постфиксного инкремента, так как одна реализация будет работать в обоих случаях.

Например, используем операцию инкремента:

```
Counter counter = new Counter() { Value = 1 };
int x1 = counter.Value; // 1
counter++; // 2
int x2 = (++counter).Value;  // 3
int x3 = (counter++).Value;  // 3
int x4 = counter.Value; // 4
```

Перегрузка операторов true и false

Ключевые слова ***true*** и ***false*** можно также использовать в качестве унарных операторов для целей перегрузки. Перегружаемые варианты этих операторов позволяют определить назначение ключевых слов ***true и false*** специально для создаваемых классов.

После перегрузки этих ключевых слов в качестве унарных операторов для конкретного класса появляется возможность использовать объекты этого класса для управления операторами ***if, while, for и do-while.***

Операторы ***true и false*** должны перегружаться попарно, а не раздельно. Рассмотрим в качестве примера перегрузку операторов ***true и false,*** для нулевого счетчика – ***false***, для счетчика <> 0 - ***true***:

```
public static bool operator true(Counter c1)
{
    if (c1.Value != 0) return true;
    else return false;
    //return c1.Value != 0;
}
Ссылок: 0
public static bool operator false(Counter c1)
{
    return c1.Value == 0;
}
```

Эти операторы перегружаются, когда мы хотим использовать объект типа в качестве условия. Например:

```
Counter counter = new Counter() { Value = 0 };
if (counter) Console.WriteLine(true);
else Console.WriteLine(false);
```

Перегружаемые операторы

При перегрузке операторов надо учитывать, что не все операторы можно перегрузить. В частности, мы можем перегрузить следующие операторы:

- унарные операторы +, -, !, ~, ++, --
- бинарные операторы +, -, *, /, %
- операции сравнения ==, !=, <, >, <=, >=
- логические операторы &&, ||

И есть ряд операторов, которые нельзя перегрузить, например, операцию равенства =, а также ряд других.

Операторы сравнения должны перегружаться парами. То есть при перегрузке оператора из пары другой оператор тоже должен перегружаться. Ниже приведены эти пары:

- Операторы == и !=
- Операторы < и >
- Операторы <= и >=

Полный список перегружаемых операторов можно найти в документации msdn.

Практическая работа №6

Определение операций в классе

1. Доработать класс разработанный в практической работе №5 по заданию. Оформить модули комментариями.
2. Разработать программу для демонстрации использования операций класса.
3. Предусмотреть в программе две кнопки «**Выход**» и «**О программе**», где вывести ФИО разработчика, номер работы и формулировку задания.
4. Оформить программу комментариями.

Варианты заданий (Базовый уровень)

1. Использовать базовый класс Car (машина), характеризуемый торговой маркой (строка), числом цилиндров, мощностью. Разработать операции для определения крутости машин. Машина считается круче, если у одной машины количество цилиндров и мощность больше чем у другой машины или при равенстве одного из параметров второй параметр больше. Разработать операцию увеличение мощности на 1.
2. Использовать класс Pair (пара чисел). Разработать операцию сложения пар (a, b) + (c, d) = (a + c, b + d). Разработать операцию для уменьшения полей на 1.
3. Использовать класс Man (человек), с полями: имя, возраст, пол и вес. Разработать операцию для увеличения возраста на 1 год. Разработать операции для определения кто тяжелее или легче.
4. Использовать класс Pair (пара чисел). Разработать операции сравнения пар: пара p1 больше пары p2, если (first.pl > first.p2) или (first.pl = first.p2) и (second.pl > second.p2).
5. Использовать класс Triangle (треугольник) с полями-сторонами. Разработать операцию для определения возможности существование треугольника с заданными сторонами true/false. Разработать операции для увеличения/уменьшения сторон на 1.
6. Использовать класс Triad (тройка положительных чисел). Разработать операции определения равенства/неравенства чисел true/false. Разработать операции проверки полного равенства/неравенства чисел в триадах (a1,b1,c1) == (a2,b2,c2).
7. Использовать класс Triad (тройка чисел). Разработать операции сравнения триад: триада p1 больше триады p2, если (first.pl > first.p2) или (first.pl = first.p2) и (second.pl > second.p2) и (three.pl > three.p2). Разработать операцию увеличения всех чисел на 10.
8. Использовать класс Pair (пара четных чисел). Разработать операцию инкремента - (a, b) = (a+b, b). Разработать операцию для вычисления произведения 2 пар чисел – (a, b) * (c, d) = (a * c, b * d).
9. Использовать класс Triad (тройка нечетных чисел). Разработать операции проверки полного равенства/неравенства чисел в триадах (a1,b1,c1) == (a2,b2,c2). Разработать операции определения, что вся тройка чисел нечетна true/false.
10. Использовать класс Triad (тройка чисел). Разработать операцию инкремента полей на 1. Разработать операцию для получения полной суммы триады с числом.
11. Использовать класс Pair (пара четных чисел). Разработать операцию перемножения пар (a, b) * (c, d) = (a * c, b * d). Разработать операцию инкремента для удвоения пары чисел.
12. Использовать класс Triad (тройка чисел). Разработать операцию для сложения триады с числом. Разработать операцию для сложения элементов одой триады с другой триадой.

13. Использовать базовый класс Pair (пара целых чисел). Разработать операции определения равенства/неравенства чисел true/false. Разработать операции вычитания пар по формуле (a, b) - (c, d) = (a - c, b - d).
14. Использовать класс Liquid (жидкость), имеющий поля названия, плотности и объема. Разработать операции для проверки, что сосуды имеют одинаковые жидкости равного объема. Разработать операции увеличения/уменьшения объема жидкости на 1.
15. Использовать класс Triad (тройка положительных чисел). Разработать операцию инкремента увеличения полей на заданное число. Разработать операцию для умножения триады с числом.

Наследование. Иерархия классов.

Основы наследования

Наследование является одним из трех основополагающих принципов объектно-ориентированного программирования, поскольку оно допускает создание иерархических классификаций. Благодаря наследованию можно создать общий класс, в котором определяются характерные особенности, присущие множеству связанных элементов. От этого класса могут затем наследовать другие, более конкретные классы, добавляя в него свои индивидуальные особенности.

В языке C# класс, который наследуется, называется базовым, а класс, который наследует, — производным. Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе, добавляя к ним свои собственные элементы.

Ниже приведена общая форма объявления класса, наследующего от базового класса:

```
class имя_производного_класса : имя_базового_класса
{
        // тело класса
}
```

Рассмотрим класс ***Person***, который описывает отдельного человека:

```
class Person
{
    private string name;
    Ссылок: 2
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    ссылка: 1
    public string Gender {get; set;}
    ссылка: 1
    public int Age { get; set; }
}
```

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) от класса **Person**, который, в свою очередь, называется базовым классом или родителем:

```
class Employee:Person
{
    Ссылок: 0
    public string Position { get; set; }
}
```

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.

Рассмотрим применение объектов этих классов:

```
Person p1 = new Person();
p1.Name = "Петр";
p1.Gender = "M";
p1.Age = 15;
Employee p2 = new Employee();
p2.Name = "Василий";
p2.Gender = "M";
p2.Age = 25;
p2.Position = "Менеджер";
```

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().***

Доступ к членам базового класса из производного класса

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

```
class Employee : Person
{
    ссылка: 1
    public string Position { get; set; }
    Ссылок: 0
    public string GetName()
    {
        return name;
    }
}
```

CS0122 'Person.name" недоступен из-за его уровня защиты.

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.

Если мы хотим сохранить защиту элементов класса и одновременно сделать так, чтобы к ним был доступ в производных классах можно использовать модификатор ***protected***.

Конструкторы в производных классах

Конструкторы не передаются производному классу при наследовании.

В связи с этим возникает вопрос, какой конструктор отвечает за построение производного класса? На этот вопрос можно ответить так: конструктор базового класса инициализирует и создает базовую часть объекта, конструктор производного класса – производную часть этого объекта.

Если оба класса не содержат конструктор, то оба класса инициализируются и создаются конструкторами по умолчанию. Базовая часть – базовым конструктором по умолчанию. Производная часть - производным конструктором по умолчанию.

Если конструктор задан только в производном классе, то им инициализируется и создается часть объекта производного класса, а базовая часть автоматически создается конструктором по умолчанию базовой части объекта.

Если заданы оба конструктора, то процесс построения объекта усложняется, так как должны выполняться конструкторы обоих классов. Доступ к элементам базового класса, в том числе и конструктору, осуществляется с помощью ключевого слова ***base***. Форма конструктора следующая:

```
Конструктор_производного_класса(параметры):base(аргументы)
{
        //Тело конструктора
}
```

Рассмотрим примеры конструкторов с параметрами для базового класса ***Person*** и производного ***Employee***:

```
class Person
{
    private string name;
    Ссылок: 3
    public string Gender { get; set; }
    Ссылок: 3
    public int Age { get; set; }

    ссылка: 1
    public Person(string name, string gender, int age)
    {
        Name = name; Gender = gender; Age = age;
    }


class Employee : Person
{
    Ссылок: 2
    public string Position { get; set; }
    ссылка: 1
    public Employee(string name, string gender, int age, string position):
        base(name, gender, age)
    {
        Position = position;
    }
```

В данном варианте конструктор ***Employee()*** вызывает метод ***base*** с параметрами ***name, gender и age***. Это, в свою очередь, приводит к вызову конструктора ***Person()***, инициализирующего свойства ***Name, Gender и Age*** значениями параметров ***name, gender и age***. Они больше не инициализируются средствами самого класса ***Employee***, где теперь остается инициализировать только его собственный член ***Position***,

Практическая работа №7

Создание наследованных классов.

1. Доработать класс разработанный в практической работе №6 по заданию. Оформить модули комментариями.
2. Разработать программу для демонстрации использования производного класса и его методов.
3. Предусмотреть в программе две кнопки «**Выход**» и «**О программе**», где вывести ФИО разработчика, номер работы и формулировку задания.
4. Оформить программу комментариями.

Варианты заданий (Базовый уровень)

1. Использовать класс Car (машина), характеризуемый торговой маркой (строка), числом цилиндров, мощностью. Создать производный класс Lorry (грузовик), характеризуемый также грузоподъемностью кузова. Определить функции переназначения марки и изменения грузоподъемности.
2. Использовать класс Pair (пара чисел). Определить класс-наследник Money с характеристиками: рубли и копейки. Переопределить операцию сложения и определить методы вычитания и деления денежных сумм.
3. Использовать класс Man (человек), с полями: имя, возраст, пол и вес. Создать производный класс Student, имеющий характеристики: факультет, курс, группа. Определить методы изменения возраста, веса, перехода на следующий курс, перевода в другую группу.
4. Использовать класс Pair (пара чисел). Определить класс-наследник Fraction с характристиками: целая часть числа и дробная часть числа. Определить операцию сложения двух денежных едениц.
5. Использовать класс Triangle (треугольник) с полями-сторонами. Создать производный класс Equilateral (равносторонний), имеющий поле площади. Определить метод вычисления площади.
6. Использовать класс Triad (тройка положительных чисел). Определить производный класс Triangle с полями-сторонами. Определить методы вычисления углов и площади треугольника.
7. Использовать класс Triad (тройка чисел). Определить производный класс Date с полями: год, месяц и день. Определить полный набор методов сравнения дат.
8. Использовать класс Pair (пара четных чисел). Определить производный класс Rectangle (прямоугольник) с характеристиками стороны прямоугольника. Определить методы вычисления периметра и площади прямоугольника.
9. Использовать класс Triad (тройка нечетных чисел). Определить производный класс Time с полями: час, минута и секунда. Определить полный набор методов сравнения моментов времени.
10. Использовать класс Triad (тройка чисел). Определить производный класс Date с полями: год, месяц и день. Переопределить методы увеличения полей на 1 и определить метод увеличения даты на n дней.
11. Использовать класс Pair (пара четных чисел). Определить производный класс треугольник RightAngled с полями-катетами. Определить методы вычисления гипотенузы и площади треугольника.
12. Использовать класс Triad (тройка чисел). Создать производный класс vector3D, задаваемый тройкой координат. Должны быть реализованы: операция сложения векторов, скалярное произведение векторов.

13. Использовать базовый класс Pair (пара целых чисел). Создать производный класс Rational; определить новые операции сложения (a, b) + (c, d) = (ad + be, bd) и деления (a, b)/ (c, d) = (ad, be); переопределить операцию вычитания (a, b) - (c, d) = = (ad - be, bd).
14. Использовать класс Liquid (жидкость), имеющий поля названия, плотности и объема. Создать необходимые методы и свойства. Создать перегруженные методы SetParams, для установки параметров жидкости. Создать производный класс Alcohol (спирт), имеющий крепость. Из класса Alcohol создать производный класс Beer (пиво), имеющий процент содержания хмеля. Определить методы переназначения и изменения крепости и процента хмеля.
15. Использовать класс Triad (тройка положительных чисел). Определить класс-наследник Time с полями: час, минута, секунда. Переопределить методы увеличения полей на 1 и определить методы увеличения на n секунд m минут.

Примечание:

Расчет угла треугольника

$$\gamma = arccos(\frac{a^2 + b^2 - c^2}{2ab})$$

Расчет площади по трем сторонам. Формула Герона.

$$S = \sqrt{p(p-a)(p-b)(p-c)}, \; p = \frac{a+b+c}{2}$$

Скалярное произведение векторов

$a * b = a_x b_x + a_y b_y + a_z b_z;$

Интерфейсы и наследование

Общие сведения об интерфейсах

Иногда в объектно-ориентированном программировании полезно определить, что именно должен делать класс, но не как он должен это делать. А в производном классе должна быть обеспечена своя собственная реализация каждого метода, определенного в его базовом классе.

В C# предусмотрено разделение интерфейса класса и его реализации с помощью ключевого слова ***interface***. В интерфейсе ни у одного из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации. В нем указывается только, что именно следует делать, но не как это делать. Как только интерфейс будет определен, он может быть реализован в любом количестве классов. Кроме того, в одном классе может быть реализовано любое количество интерфейсов.

Для реализации интерфейса в классе должны быть предоставлены тела (т.е. конкретные реализации) методов, описанных в этом интерфейсе. Каждому классу предоставляется полная свобода для определения деталей своей собственной реализации интерфейса. Следовательно, один и тот же интерфейс может быть реализован в двух классах по-разному. Тем не менее в каждом из них должен поддерживаться один и тот же набор методов данного интерфейса. А в том коде, где известен такой интерфейс, могут использоваться объекты любого из этих двух классов, поскольку интерфейс для всех этих объектов остается одинаковым.

Благодаря поддержке интерфейсов в C# может быть в полной мере реализован главный принцип полиморфизма: один интерфейс — множество методов.

Определение интерфейса

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

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

- Методы
- Свойства
- Индексаторы
- События

Интерфейсы не могут содержать члены данных. В них нельзя также определить конструкторы, деструкторы или операторные методы.

Ниже приведена упрощенная форма объявления интерфейса.

```
interface имя
{
        тип имя_свойства1{ get; set; }
        …
        тип имя_свойстваN{ get; set; }
        возвращаемый_тип имя_метода1 (список_параметров);
        возвращаемый_тип имя_метода2 (список_параметров);
        …
        возвращаемый_тип имя_методаN (список_параметров);

}
```

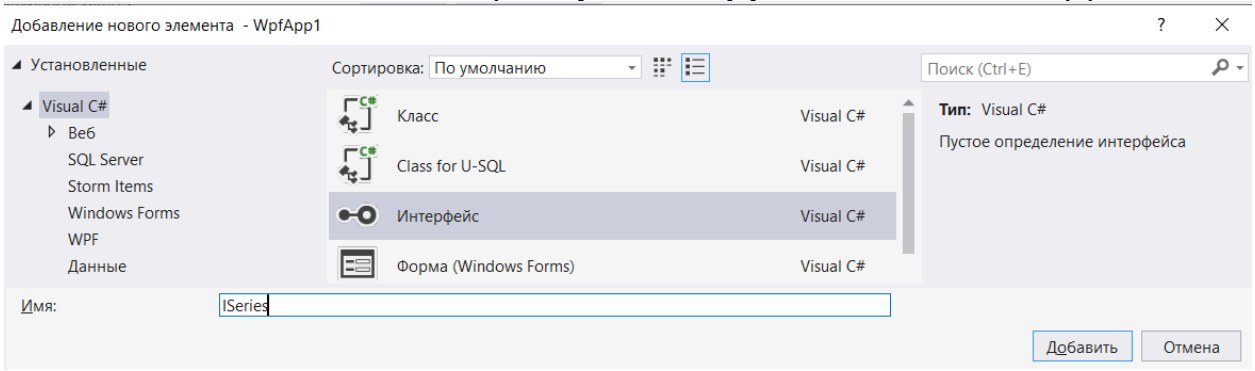
Рассмотрим пример объявления интерфейса для класса, генерирующего последовательный ряд чисел.

```
interface ISeries
{
    Ссылок: 0
    int Next { get; } // Возвратить следующее по порядку число
    Ссылок: 0
    int GetNext(); // Возвратить следующее по порядку число
    Ссылок: 0
    void Reset(); // Перезапустить
    Ссылок: 0
    void SetStart(int x); // Задать начальное значение
}
```

Этому интерфейсу присваивается имя ***ISeries***. Префикс I в имени интерфейса указывать необязательно, но это принято делать в практике программирования, чтобы как-то отличать интерфейсы от классов.

Еще один момент в объявлении интерфейса: если 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.

Стоит отметить, что в ***Visual Studio*** есть специальный компонент для добавления нового интерфейса в отдельном файле. Для добавления интерфейса в проект можно использовать команду ***Проект – Добавить новый элемент*** и в диалоговом окне добавления нового компонента выбрать пункт ***Интерфейс*** и задать имя интерфейса:



Применение интерфейса

Как только интерфейс будет определен, он может быть реализован в одном или нескольких классах. Для реализации интерфейса достаточно указать его имя после имени класса, аналогично базовому классу. Ниже приведена общая форма реализации интерфейса в классе.

```
class имя_класса : имя_интерфейса
{
        // тело класса
}
```

В классе допускается реализовывать несколько интерфейсов. В этом случае все реализуемые в классе интерфейсы указываются списком через запятую. В классе можно наследовать базовый класс и в тоже время реализовать один или более интерфейс. В таком случае имя базового класса должно быть указано перед списком интерфейсов, разделяемых запятой.

Методы, реализующие интерфейс, должны быть объявлены как ***public***. Дело в том, что в самом интерфейсе эти методы неявно подразумеваются как открытые, поэтому их реализация также должна быть открытой. Кроме того, возвращаемый тип и сигнатура

реализуемого метода должны точно соответствовать возвращаемому типу и сигнатуре, указанным в определении интерфейса.

Рассмотрим пример программы, в которой реализуется представленный ранее интерфейс ***ISeries***. В этой программе создадим класс ***TwoSeries***, генерирующий арифметическую прогрессию чисел, в котором каждое последующее число на два больше предыдущего числа.

```
class TwoSeries : ISeries
{
    int start; // Начальное значение
    int val;   // Текущее значение
    Ссылок: 0
    public TwoSeries()
    {
        start = 0;
        val = 0;
    }
    ссылка: 1
    public int Next
    {
        get { return GetNext(); }
    }
    Ссылок: 2
    public int GetNext()
    {
        val += 2;
        return val;
    }
    ссылка: 1
    public void Reset()
    {
        val = start;
    }
    ссылка: 1
    public void SetStart(int x)
    {
        start = x;
        val = start;
    }
}
```

Можно используя интерфейс ***ISeries*** создать и другие классы для формирования последовательностей чисел, например:

класс арифметическая прогрессия (ArithmeticProgression);
класс геометрическая прогрессия (GeometricProgression);
класс последовательность простых чисел (Primes);
класс последовательность четных чисел (Evens);

Рассмотрим применение класса ***TwoSeries***, реализующего интерфейс ***ISeries***.

```
TwoSeries series1 = new TwoSeries();
for (int i = 0; i < 5; i++)
    listBox1.Items.Add(series1.GetNext());

series1.Reset();
for (int i = 0; i < 5; i++)
    listBox2.Items.Add(series1.GetNext());

series1.SetStart(100);
for (int i = 0; i < 5; i++)
    listBox3.Items.Add(series1.GetNext());
```

Определим, что выводится в списки ListBox при выполнении кода программы.

Применение интерфейсных ссылок

Как это ни покажется странным, но в C# допускается объявлять переменные ссылочного интерфейсного типа, т.е. переменные ссылки на интерфейс. Такая переменная может ссылаться на любой объект, реализующий ее интерфейс. При вызове метода для объекта посредством интерфейсной ссылки выполняется его вариант, реализованный в классе данного объекта.

В приведенном ниже примере программы демонстрируется применение интерфейсной ссылки. В этой программе переменная ссылки на интерфейс используется с целью вызвать методы для объектов обоих классов — ***TwoSeries и Primes***. Для сокращения примера описание интерфейса и классов опущено, т.к. оно уже было приведено выше.

```
interface ISeries
{ }
Ссылок: 4
class TwoSeries : ISeries
{ }
Ссылок: 0
class Primes : ISeries
{ }

TwoSeries twoOb = new TwoSeries();
Primes primeOb = new Primes();
ISeries ob;
for (int i = 0; i < 5; i++)
{
    ob = twoOb;
    listBox2.Items.Add(ob.GetNext());
    ob = primeOb;
    listBox2.Items.Add(ob.GetNext());
}// 2 3 4 5 6 7 8 11 10 13
```

Определим, что означают цифры в комментарии программы.

Переменная ***ob*** объявляется для ссылки на интерфейс ***ISeries***. Это означает, что в ней могут храниться ссылки на объект любого класса, реализующего интерфейс ***ISeries***. В данном случае она служит для ссылки на объекты ***twoOb и primeOb*** классов ***TwoSeries и Primes*** соответственно, в которых реализован интерфейс ***ISeries***.

Замечание: переменной ссылки на интерфейс доступны только методы, объявленные в ее интерфейсе. Поэтому интерфейсную ссылку нельзя использовать для доступа к любым другим переменным и методам, которые не поддерживаются объектом класса, реализующего данный интерфейс.

Стандартные интерфейсы

В библиотеке ***.NET*** определено множество стандартных интерфейсов, задающих желаемое поведение объектов.

С реализацией ряда интерфейсов мы уже сталкивались. Например, когда работали с элементом ***ListBox***, подумаем какой это интерфейс из таблицы ниже.

Таблица 1. Примеры стандартных интерфейсов

Интерфейс	Назначение
ICloneable	Позволяет клонировать объекты
ICompable	Позволяет сравнивать два объекта
IDictionary	Позволяет представлять содержимое объекта в виде пар "имя-значение"
Ilist	Поддерживает методы добавления, удаления и индексирования элементов в списке объектов

Сравнение объектов (интерфейс iComparable)

Интерфейс ***IComparable*** содержит всего один метод ***CompareTo()***, возвращающий результат сравнения двух объектов – текущего и переданного ему в качестве параметра:

```
interface IComparable
{
        int CompareTo(object obj)
}
```

Метод должен возвращать:

- 0, если текущий объект и параметр равны;
- отрицательное число, если текущий объект меньше параметра;
- положительное число, если текущий объект больше параметра.

Реализуем интерфейс ***IComparable*** в классе ***Monster***. В качестве критерия сравнения объектов выберем поле ***health***.

```
class Monster : IComparable
{
    string name;
    int health, ammo;
    Ссылок: 0
    public int CompareTo(object obj)// реализация интерфейса
    {
        Monster temp = (Monster)obj;
        if (this.health > temp.health) return 1;
        if (this.health < temp.health) return -1;
        return 0;
    }
}
```

Рассмотрим пример использования функции ***CompareTo***:

```
Monster m1 = new Monster();
m1.health = 1000;
Monster m2 = new Monster();
m2.health = 1100;
if (m1.CompareTo(m2) == 0) MessageBox.Show("Монстры равны");
```

Клонирование объектов (интерфейс iCloneable)

Клонирование - создание копии объекта. Копия объекта называется ***клоном***. При присваивании одного экземпляра другому копируется ссылка, а не сам объект, т.е. два элемента реально ссылаются на один объект.

Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом ***MemberwiseClone()***, который любой объект наследует от класса ***object***. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются. Это называется ***поверхностным клонированием***.

Для создания полностью независимых объектов необходимо ***глубокое копирование***, когда в памяти создается дубликат всего дерева объектов, то есть объектов, на которые ссылаются поля объекта, поля полей, и т.д. Алгоритм глубокого копирования сложен, требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.

Объект, имеющий собственные алгоритмы клонирования, должен объявляться как производный интерфейса ***ICloneable*** и переопределять его единственный метод ***Clone()***.

```
interface ICloneable
{
        object Clone().
}
```

Модернизируем предыдущий пример и создадим поверхностную копию экземпляра класса Monster с помощью метода ***MemberwiseClone()*** и реализации интерфейса ***ICloneable***. Метод ***MemberwiseClone()*** можно вызывать только из методов класса, так как он объявлен в классе ***object*** как ***protected***.

```
class Monster : IComparable, ICloneable
{
    public string name;
    public int health, ammo;
    //Поверхностное клонирование
    Ссылок: 0
    public Monster ShallowClone()
    {
        return (Monster)this.MemberwiseClone();
    }
    //Полное клонирование
    Ссылок: 0
    public object Clone()
    {
        Monster m = new Monster();
        m.name = this.name;
        m.health = this.health;
        m.ammo = this.ammo;
        return m;
    }
```

Используем полученные функции для клонирования объекта в программе:

```
Monster m = new Monster();
m.health = 1000;
m.name = "Вася";
m.ammo = 10;
//Ссылка
Monster x1 = m;
//Поверхностное клонирование
Monster x2 = m.ShallowClone();
//Полное клонирование
Monster x3 = (Monster)m.Clone();
```

Практическая работа №8

Работа с объектами через интерфейсы

1. Разработать класс по заданию. Оформить модули комментариями.
2. Используя интерфейсы ***IComparable и ICloneable*** разработать методы сравнения и клонирования разработанных классов.
3. Разработать программу для демонстрации использования класса и его методов.
4. Предусмотреть в программе две кнопки «**Выход**» и «**О программе**», где вывести ФИО разработчика, номер работы и формулировку задания.
5. Оформить программу комментариями.

Варианты заданий (Базовый уровень)

1. Создать интерфейс - человек. Создать классы – студент и студент-отец семейства. Классы должны включать конструкторы, функцию для формирования строки информации о студенте. Сравнение производить по фамилии.
2. Создать интерфейсы - корабль, грузовой транспорт. Создать класс грузовой корабль. Класс должен включать конструктор, функцию для формирования строки информации о корабле. Сравнение производить по грузоподъемности.
3. Создать интерфейсы - автомобиль, пассажирский транспорт. Создать класс автобус. Класс должен включать конструктор, функцию для формирования строки информации об автобусе. Сравнение производить по вместимости пассажиров.
4. Создать интерфейсы - работник и отец-семейства. Создать класс работника-отца семейства. Класс должен включать конструктор, функцию для формирования строки информации о работнике. Сравнение производить по фамилии.
5. Создать интерфейс – серия чисел (см. лекцию). Создать класс – геометрическая прогрессия. Класс должен включать конструктор. Сравнение производить по шагу прогрессии.
6. Создать интерфейсы – человек и печать (для формирования информации об объекте). Создать класс – студент. Класс должен включать конструкторы функцию для формирования строки информации о студенте. Сравнение производить по фамилии.
7. Создать интерфейс – фигура (площадь, периметр). Создать классы - прямоугольник, круг. Классы должны включать конструкторы, функцию для формирования строки информации о фигуре. Сравнение производить по площади.
8. Создать интерфейс – работник. Создать классы - служащий с почасовой оплатой, служащий с окладом. Классы должны включать конструкторы, функцию для формирования строки информации о работнике. Определить функцию начисления зарплаты. Сравнение производить по фамилии.
9. Создать интерфейс - человек. Создать классы – девушка и парень. Классы должны включать конструкторы, функцию для формирования строки информации о человеке. Определить функцию реакции человека на вновь увиденного другого человека. Сравнение производить по фамилии.
10. Создать интерфейс – фигура (объем). Создать классы - параллелепипед, шар. Классы должны включать конструкторы, функцию для формирования строки информации о фигуре. Объем шара $V=4\ \pi r^3/3$. Сравнение производить по объему.
11. Создать интерфейс - человек, у которого есть имя, функция печати. Создать класс отец, у которого функция печати выводит имя. Создать класс ребенок, у которого есть отец, отчество, функция печати выводит имя и отчество. Классы должны включать конструкторы. Сравнение производить по фамилии.

12. Создать интерфейс – арифметические операции (+, -, *, /). Создать класс пара чисел. Класс должен включать конструктор, функцию для формирования строки с арифметической операцией. Сравнение производить по парам чисел.
13. Создать интерфейс - человек. Создать классы - работник и работник-отец семейства. Классы должны включать конструкторы, функцию для формирования строки информации о работнике. Сравнение производить по фамилии.
14. Создать интерфейс – серия чисел (см. лекцию). Создать класс – простые числа. Класс должен включать конструктор. Сравнение производить по текущему значению.
15. Создать интерфейсы – счет (сумма на счету, положить и получить деньги) и клиент. Создать класс – Клиент. Класс должен включать конструкторы функцию для формирования строки информации о клиенте. Сравнение производить по сумме на счету.

Структуры

Описание структур

Как вам должно быть уже известно, классы относятся к ссылочным типам данных. Это означает, что объекты конкретного класса доступны по ссылке, в отличие от значений простых типов, доступных непосредственно. Но иногда прямой доступ к объектам как к значениям простых типов оказывается полезно иметь, например, ради повышения эффективности программы. Ведь каждый доступ к объектам (даже самым мелким) по ссылке связан с дополнительными издержками на расход вычислительных ресурсов и оперативной памяти. Для разрешения подобных затруднений в C# предусмотрена структура, которая подобна классу, но относится к типу значения, а не к ссылочному типу данных.

Структуры объявляются с помощью ключевого слова ***struct*** и с точки зрения синтаксиса подобны классам. Ниже приведена общая форма объявления структуры:

```
struct имя : интерфейсы
{
        // объявления членов
}
```

где имя обозначает конкретное имя структуры.

Применение структур

Одни структуры не могут наследовать другие структуры и классы или служить в качестве базовых для других структур и классов.

Структуры, как и все остальные типы данных в C#, наследуют класс ***object***.

В структуре можно реализовать один или несколько интерфейсов, которые указываются после имени структуры списком через запятую.

Как и у классов, у каждой структуры имеются свои члены: методы, поля, индексаторы, свойства, операторные методы и события.

В структурах допускается также определять конструкторы, но не деструкторы. В то же время для структуры нельзя определить конструктор, используемый по умолчанию (т.е. конструктор без параметров). Дело в том, что конструктор, вызываемый по умолчанию, определяется для всех структур автоматически и не подлежит изменению. Такой конструктор инициализирует поля структуры значениями, задаваемыми по умолчанию.

Поскольку структуры не поддерживают наследование, то их члены нельзя указывать как ***abstract, virtual или protected***.

Объект структуры может быть создан с помощью оператора ***new*** таким же образом, как и объект класса, но в этом нет особой необходимости. Ведь когда используется оператор ***new***, то вызывается конструктор, используемый по умолчанию. А когда этот оператор не используется, объект по-прежнему создается, хотя и не инициализируется. В этом случае инициализацию любых членов структуры придется выполнить вручную.

В приведенном ниже примере описываем структуру для хранения информации о книге.

```
struct Book {
    public string Author;
    public string Title;
    public int Copyright;
    ссылка: 1
    public Book(string author, string title, int copyright)
    {
        Author = author;
        Title = title;
        Copyright = copyright;
    }
}
```

В приведенном ниже примере программы демонстрируется применение структуры для хранения информации о книге.

```
// вызов явно заданного конструктора
Book book1 = new Book("Герберт Шилдт", "Полный справочник по C# 4.0", 2010);
Book book2 = new Book(); // вызов конструктора по умолчанию
Book book3;  // конструктор не вызывается

MessageBox.Show(book1.Author + ", " + book1.Title + ", (c) " + book1.Copyright);

if (book2.Title == null) MessageBox.Show("Член book2.Title пуст.");
// А теперь ввести информацию в структуру book2.
book2.Title = "О дивный новый мир";
book2.Author = "Олдос Хаксли";
book2.Copyright = 1932;
MessageBox.Show("Структура book2 теперь содержит:\n");
MessageBox.Show(book2.Author + ", " + book2.Title + ", (c) " + book2.Copyright);

MessageBox.Show(book3.Title); // неверно, этот член структуры
// нужно сначала инициализировать
book3.Title = "Красный шторм";
MessageBox.Show(book3.Title); // теперь верно
```

Как демонстрирует приведенный выше пример программы, структура может быть инициализирована с помощью оператора ***new*** для вызова конструктора или же путем простого объявления объекта. Так, если используется оператор ***new***, то поля структуры инициализируются конструктором, вызываемым по умолчанию (в этом случае во всех полях устанавливается задаваемое по умолчанию значение), или же конструктором, определяемым пользователем. Если оператор ***new*** не используется, как это имеет место для структуры ***book3***, то объект структуры не инициализируется, а его поля должны быть установлены вручную перед тем, как пользоваться данным объектом.

Когда одна структура присваивается другой, создается копия ее объекта. В этом заключается одно из главных отличий структуры от класса. Ранее мы выдели, что при работе с классами один класс присваивается ссылке на другой класс, в итоге ссылка в левой части оператора присваивания указывает на тот же самый объект, что и ссылка в правой его части.

О назначении структур

В связи с изложенным выше возникает резонный вопрос: зачем в C# включена структура, если она обладает более скромными возможностями, чем класс? Ответ на этот

вопрос заключается в повышении эффективности и производительности программ. Структуры относятся к типам значений, и поэтому ими можно оперировать непосредственно, а не по ссылке. Следовательно, для работы со структурой вообще не требуется переменная ссылочного типа, а это означает в ряде случаев существенную экономию оперативной памяти. Более того, работа со структурой не приводит к ухудшению производительности, столь характерному для обращения к объекту класса. Ведь доступ к структуре осуществляется непосредственно, а к объектам — по ссылке, поскольку классы относятся к данным ссылочного типа. Косвенный характер доступа к объектам подразумевает дополнительные издержки вычислительных ресурсов на каждый такой доступ, тогда как обращение к структурам не влечет за собой подобные издержки. И вообще, если нужно просто сохранить группу связанных вместе данных, не требующих наследования и обращения по ссылке, то с точки зрения производительности для них лучше выбрать структуру

Практическая работа №9

Работа с типом данных структура

1. Разработать структуру по заданию. Оформить модули комментариями.
2. Разработать программу для демонстрации использования структуры.
3. Предусмотреть в программе две кнопки «**Выход**» и «**О программе**», где вывести ФИО разработчика, номер работы и формулировку задания.
4. Оформить программу комментариями.

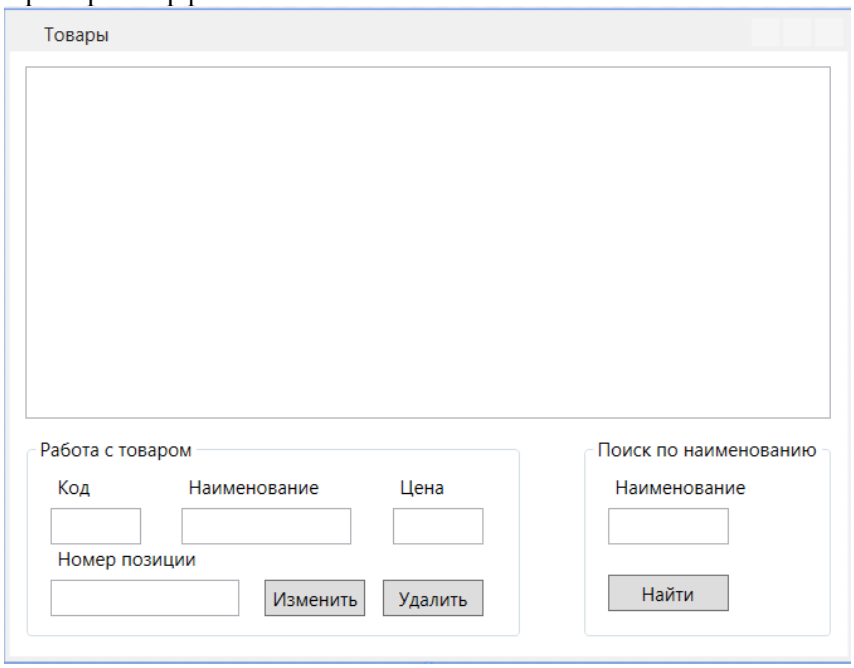
Варианты заданий (Базовый уровень)

1. Заполнить таблицу анкетных данных на 5 человек с полями: ФИО, пол, год рождения, место рождения, национальность. Вывести результат на экран. Вывести средний возраст.
2. Заполнить таблицу с аппаратными средствами на 5 компьютеров с полями: тип процессор, память, HDD, видео. Вывести результат на экран. Вывести средний объем памяти.
3. Заполнить таблицу участников забега на 100 метров на 8 человек с полями: ФИО, номер, результат. Вывести результат на экран. Вывести средний результат.
4. Заполнить таблицу фильмотеки на 7 кассет с полями: фильм, жанр, год выпуска, продолжительность просмотра. Вывести результат на экран. Вывести список фильмов заданного жанра.
5. Заполнить таблицу со списком сотрудников на 7 человек с полями: ФИО, пол, должность, стаж работы, оклад. Вывести результат на экран. Вывести средний оклад.
6. Заполнить таблицу на 5 предметов с полями: предмет, ФИО лектора, номер аудитории, кол-во часов в семестре. Вывести результат на экран. Вывести список лекторов работающих в заданной аудитории.
7. Заполнить таблицу со списком телефонных абонентов на 7 человек с полями: ФИО, номер телефона, адрес. Вывести результат на экран. Вывести список абонентов живущих на заданной улице.
8. Заполнить таблицу со списком 5 стран мира с полями: страна, народонаселение, столица, денежная единица. Вывести результат на экран. Вывести общее население всех стран.
9. Заполнить таблицу с учебной литературой с полями: название, автор, издательство, кол-во страниц. Вывести результат на экран. Вывести список книг заданного издательства.
10. Заполнить таблицу успеваемости студента по 5 дисциплинам с полями: дисциплина, успеваемость за месяц. Вывести результат на экран. Вывести среднюю успеваемость по всем дисциплинам.
11. Багаж пассажира характеризуется количеством вещей и общим весом вещей. Сведения о багаже каждого пассажира представляют собой структуру с двумя полями: одно поле целого типа (количество вещей) и одно - действительное (вес в килограммах). Вывести результат на экран. Найти багаж, средний вес одной вещи в котором отличается не более, чем на 0.3 кг от общего среднего веса одной вещи.
12. После поступления в ВУЗ о студентах собрана информация: фамилия, нуждается ли в общежитии, стаж, работал ли, что окончил, какой язык изучал. Вывести результат на экран. Вывести информацию, сколько человек нуждаются в общежитии.

13. Описать, используя структуру, данные на учеников (фамилия, улица, дом, квартира). Вывести результат на экран. Вывести информацию, сколько учеников живет на заданной улице.
14. Описать, используя структуру, почтовую сортировку (город, улица, дом, квартира, кому, ценность). Вывести результат на экран. Вывести информацию, сколько посылок отправлено в заданный город.
15. Описать, используя структуру, завод (наименование станка, время простоя в месяц, время работы в месяц). Вывести результат на экран. Вывести информацию, показывающую общее время простоя на заводе.
16. На олимпиаде по информатике на школьников заполнялись анкеты: фамилия, номер школы, класс, занятое место. Вывести результат на экран. Напечатать списки школ, занявших призовые места.

Пример: Требуется описать структуру с именем Tovar, содержащую информацию о хранящемся на складе товаре: (код товара, наименование товара, цену). Организовать поиск нужного товара по его коду.

Пример интерфейса.



Пример кода программы.

```
struct Goods
{
    public int Kod;
    public string Product;
    public Double Price;
}

Goods[] goods;

ссылка: 1
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    goods = new Goods[10];
    for (int i=1; i <= 10;i++) ListBox.Items.Add("");
}
// Кнопка изменить
ссылка: 1
private void btn_Edit_Click(object sender, RoutedEventArgs e)
{
    int pos = Convert.ToInt32(txtKod.Text);
    goods[pos].Kod = Convert.ToInt32(txtKod.Text);
    goods[pos].Product = txtName.Text;
    goods[pos].Price = Convert.ToDouble(txtPrice.Text);
    ListBox.Items[pos] =
        $" {goods[pos].Kod,5}  {goods[pos].Product,15}  {goods[pos].Price,7} ";
}
```

Делегаты

Описание и использование делегатов

Делегаты представляют такие объекты, которые указывают на методы. То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

Для объявления делегата используется ключевое слово ***delegate***, после которого идет возвращаемый тип, название и параметры. Например:

delegate void Message()

Делегат ***Message*** в качестве возвращаемого типа имеет тип ***void*** (то есть ничего не возвращает) и не принимает никаких параметров. Это значит, что этот делегат может указывать на любой метод, который не принимает никаких параметров и ничего не возвращает.

Рассмотрим применение этого делегата:

```
delegate void Message(); // 1. Объявляем делегат
ссылка: 1
private void GoodMorning()
{
    MessageBox.Show("Good Morning");
}
ссылка: 1
private void GoodEvening()
{
    MessageBox.Show("Good Evening");
}
private void button1_Click(object sender, EventArgs e)
{
    Message mes; // 2. Создаем переменную делегата
    if (DateTime.Now.Hour < 12)
    {
        mes = GoodMorning; // 3. Присваиваем этой переменной адрес метода
    }
    else
    {
        mes = GoodEvening;
    }
    mes(); // 4. Вызываем метод
}
```

Здесь сначала мы определяем делегат:

```
delegate void Message(); // 1. Объявляем делегат
```

В данном случае делегат можно определить внутри класса, но также можно определить делегат вне класса внутри пространства имен.

Для использования делегата объявляется переменная этого делегата:

```
Message mes; // 2. Создаем переменную делегата
```

С помощью свойства ***DateTime.Now.Hour*** получаем текущий час. И в зависимости от времени в делегат передается адрес определенного метода. Обратите внимание, что методы эти имеют то же возвращаемое значение и тот же набор параметров (в данном случае отсутствие параметров), что и делегат.

```
mes = GoodMorning; // 3. Присваиваем этой переменной адрес метода
```

Затем через делегат вызываем метод, на который ссылается данный делегат:

```
mes(); // 4. Вызываем метод
```

Вызов делегата производится подобно вызову метода.

Рассмотрим пример использование делегата с функциями, имеющими параметры и возвращаемый результат:

```
delegate int Operation(int x, int y);
ссылка: 1
private static int Add(int x, int y)
{
    return x + y;
}
ссылка: 1
private static int Multiply(int x, int y)
{
    return x * y;
}
ссылка: 1
private void button2_Click(object sender, EventArgs e)
{
    // присваивание адреса метода через контруктор
    Operation op = Add; // делегат указывает на метод Add
    int result = op(4, 5); // фактически Add(4, 5)

    op = Multiply; // теперь делегат указывает на метод Multiply
    result = op(4, 5); // фактически Multiply(4, 5)
}
```

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

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

Делегаты необязательно могут указывать только на методы, которые определены в том же классе, где определена переменная делегата. Это могут быть также методы из других классов и структур.

```
class Math
{
    ссылка: 1
    public int Sum(int x, int y) { return x + y; }
}
delegate int Operation(int x, int y);
ссылка: 1
private void button3_Click(object sender, EventArgs e)
{
    Math math = new Math();
    Operation op = math.Sum;
    int result = op(4, 5);
}
```

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

```
delegate int Operation(int x, int y);
ссылка: 1
private void button3_Click(object sender, EventArgs e)
{
    Math math = new Math();
    Operation op1 = math.Sum;
    Operation op2 = new Operation(math.Sum);
    int result = op1(4, 5);
}
```

Оба способа равноценны.

Добавление методов в делегат

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

```
delegate void Message();
ссылка: 1
private static void Hello()
{
    Console.WriteLine("Hello");
}
ссылка: 1
private static void HowAreYou()
{
    Console.WriteLine("How are you?");
}
ссылка: 1
private void button4_Click(object sender, EventArgs e)
{
    Message mes1 = Hello;
    mes1 += HowAreYou;  // теперь mes1 указывает на два метода
    mes1(); // вызываются оба метода - Hello и HowAreYou
}
```

В данном случае в список вызова делегата ***mes1*** добавляются два метода - ***Hello и HowAreYou***. И при вызове ***mes1*** вызываются сразу оба этих метода.

Для добавления делегатов применяется операция +=. Однако стоит отметить, что в реальности будет происходить создание нового объекта делегата, который получит методы старой копии делегата и новый метод, и новый созданный объект делегата будет присвоен переменной mes1.

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

```
Message mes1 = Hello;
mes1 += HowAreYou;  // теперь mes1 указывает на два метода
mes1 += Hello;
mes1 += Hello;
mes1();
```

Подобным образом мы можем удалять методы из делегата с помощью операции -=:

```
Message mes1 = Hello;
mes1 += HowAreYou;  // теперь mes1 указывает на два метода
mes1 += Hello;
mes1 += Hello;
mes1 -= HowAreYou;
mes1();
```

Применение делегатов

Выше были рассмотрены общие моменты по работе с делегатами. Приведенные примеры, возможно, не показывают истинной силы делегатов, так как нужные нам методы в данном случае мы можем вызвать и напрямую без всяких делегатов. Однако наиболее сильная сторона делегатов состоит в том, что они позволяют делегировать выполнение некоторому коду извне, так как на момент написания программы мы можем не знать, что за код будет выполняться. Мы просто вызываем делегат. А какой метод будет непосредственно выполняться при вызове делегата, будет решаться потом.

Например, наши классы будут распространяться в виде отдельной библиотеки классов, которая будет подключаться в проект другого разработчика. И этот разработчик захочет определить какую-то свою логику обработки, но изменить исходный код нашей библиотеки классов он не может. И делегаты как раз предоставляют возможность вызвать некое действие, которое задается извне и которое на момент написания кода может быть неизвестно.

Подробно в текущем курсе рассматриваться применение делегатов не будет. Тема делегаты изучается для общего развития. Подробно работу с делегатами можете изучить самостоятельно.

Коллекции

Общие сведения

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, которое трудно изменить. Очень часто бываем, 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

Интерфейсы коллекций

Для работы с коллекциями определен целый ряд интерфейсов коллекций. Которые определяют функциональные возможности, которые являются общими для всех классов коллекций. Такой подход позволяет со всеми коллекциями работать однотипными методами, что упрощает использование разных коллекций.

Таблица 2. Интерфейсы коллекций

Класс	***Описание***
ICollection	Определяет элементы, которые должны иметь все коллекции
IComparer	Определяет метод ***Compare()*** для сравнения объектов, хранящихся в коллекции
IDictionary	Определяет коллекцию, состоящую из пар “ключ-значение”
IEnumerable	Определяет метод ***GetEnumerator()***, предоставляющий перечислитель для любого класса коллекции
IEnumerator	Предоставляет методы, позволяющие получать содержимое коллекции по очереди
IList	Определяет коллекцию, доступ к которой можно получить с помощью индексатора

Необобщенные коллекции

Необобщенные коллекции представляют собой структуры данных общего назначения, оперирующие ссылками на объекты. Таким образом, они позволяют манипулировать объектом любого типа, хотя и не типизированным способом. В этом состоит их преимущество и в то же время недостаток. Благодаря тому, что необобщенные коллекции оперируют ссылками на объекты, в них можно хранить разнотипные данные. Это удобно в тех случаях, когда требуется манипулировать совокупностью разнотипных объектов или же когда типы хранящихся в коллекции объектов заранее неизвестны. Но если коллекция предназначается для хранения объекта конкретного типа, то необобщенные коллекции не обеспечивают типовую безопасность, которую можно обнаружить в обобщенных коллекциях.

Таблица 3. Классы необобщенных коллекций

Класс	***Описание***
ArrayList	Определяет динамический массив, т.е. такой массив, который может при необходимости увеличивать свой размер
Hashtable	Определяет хеш-таблицу для пар “ключ-значение”
Queue	Определяет очередь, или список, действующий по принципу “первым пришел — первым обслужен”
SortedList	Определяет отсортированный список пар “ключ-значение”
Stack	Определяет стек, или список, действующий по принципу “первым пришел — последним обслужен”

Обобщенные коллекции

Классы обобщенных коллекций являются не более чем обобщенными эквивалентами рассматривавшихся ранее классов необобщенных коллекций, хотя это соответствие не является взаимно однозначным.

Обобщенные коллекции типизированы. Это означает, что в обобщенной коллекции можно хранить только те элементы, которые совместимы по типу с ее аргументом.

Таблица 4. Классы обобщенных коллекций

Класс	***Описание***
Dictionary <TKey, TValue>	Сохраняет пары “ключ-значение". Обеспечивает такие же функциональные возможности, как и необобщенный класс ***Hashtable***
HashSet <T>	Сохраняет ряд уникальных значений, используя хеш-таблицу
LinkedList <T>	Сохраняет элементы в двунаправленном списке
List <T>	Создает динамический массив. Обеспечивает такие же функциональные возможности, как и необобщенный класс ***ArrayList***
Queue <T>	Создает очередь. Обеспечивает такие же функциональные возможности, как и необобщенный класс ***Queue***
SortedDictionary <TKey, TValue>	Создает отсортированный список из пар “ключ-значение"
SortedList <TKey, TValue>	Создает отсортированный список из пар “ключ-значение”. Обеспечивает такие же функциональные возможности, как и необобщенный класс SortedList
SortedSet <T>	Создает отсортированное множество
Stack <T>	Создает стек. Обеспечивает такие же функциональные возможности, как и необобщенный класс Stack

Параллельные коллекции

В версию 4.0 среды .NET Framework добавлено новое пространство имен System.Collections.Concurrent. Оно содержит коллекции, которые являются потокобезопасными и специально предназначены для параллельного программирования. Это означает, что они могут безопасно использоваться в многопоточной программе, где возможен одновременный доступ к коллекции со стороны двух или больше параллельно исполняемых потоков.

Таблица 5. Классы параллельных коллекций

Класс	***Описание***
ConcurrentDictionary <TKey, TValue>	Сохраняет пары “ключ-значение", а значит, реализует параллельный словарь

ConcurrentQueue <T>	Реализует параллельную очередь и соответствующий вариант интерфейса ***IProducerConsumerCollection***
ConcurrentStack <T>	Реализует параллельный стек и соответствующий вариант интерфейса ***IproducerConsumerCollection***

Необобщенная коллекция ArrayList

В классе ***ArrayList*** поддерживаются динамические массивы, расширяющиеся и сокращающиеся по мере необходимости. В языке C# стандартные массивы имеют фиксированную длину, которая не может изменяться во время выполнения программы. Это означает, что количество элементов в массиве нужно знать заранее. Но иногда требуемая конкретная длина массива остается неизвестной до самого момента выполнения программы. Именно для таких ситуаций и предназначен класс ***ArrayList***. В классе ***ArrayList*** определяется массив переменной длины, который состоит из ссылок на объекты и может динамически увеличивать и уменьшать свой размер. Массив типа ***ArrayList*** создается с первоначальным размером. Если этот размер превышается, то массив автоматически расширяется. А при удалении объектов из такого массива он автоматически сокращается.

В классе ArrayList реализуются интерфейсы ***ICollection, IList, IEnumerable и ICloneable***. Описан класс в пространстве имен ***System.Collections***.

Таблица 6. Основные методы класса ***ArrayList***

Методы	***Описание***
int Add(object value)	Добавляет в список объект ***value***
void AddRange(ICollection col)	Добавляет в список объекты коллекции ***col***, которая представляет интерфейс ***ICollection*** - интерфейс, реализуемый коллекциями
void Clear()	Удаляет из списка все элементы
bool Contains(object value)	Проверяет, содержится ли в списке объект ***value***. Если содержится, возвращает ***true***, иначе возвращает ***false***
void CopyTo(Array array)	Копирует текущий список в массив ***array***
ArrayList GetRange(int index, int count)	Возвращает новый список ***ArrayList***, который содержит ***count*** элементов текущего списка, начиная с индекса ***index***
int IndexOf(object value)	Возвращает индекс элемента ***value***
void Insert(int index, object value)	Вставляет в список по индексу ***index*** объект ***value***
void InsertRange(int index, ICollection col)	Вставляет в список начиная с индекса ***index*** коллекцию ***ICollection***
int LastIndexOf(object value)	Возвращает индекс последнего вхождения в списке объекта ***value***
void Remove(object value)	Удаляет из списка объект ***value***
void RemoveAt(int index)	Удаляет из списка элемент по индексу ***index***
void RemoveRange(int index, int count)	Удаляет из списка ***count*** элементов, начиная с индекса ***index***
void Reverse()	Переворачивает список

void SetRange(int index, ICollection col)	Копирует в список элементы коллекции ***col***, начиная с индекса ***index***
void Sort()	Сортирует коллекцию
object[] ToArray()	Копирует элементы списка ***ArrayList*** в новый массив ***Object***
Array ToArray (Type type)	Копирует элементы списка ***ArrayList*** в новый массив с элементами указанного типа. `int [] m = (int[]) mas.ToArray(typeof ( int ))`

Рассмотрим пример использования коллекции ***ArrayList***:

```
ArrayList mas = new ArrayList();
ссылка: 1
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Добавим элементы в массив
    mas.Add(1);
    mas.Add(2);
    mas.Add(3);
    mas.Add("4");
    mas.Add("Конец");
    MessageBox.Show("Количество элементов - " + mas.Count);

    //Отобразим элементы
    for (int i = 0; i < mas.Count; i++) MessageBox.Show(mas[i].ToString());

    //Удалить элементы
    mas.Remove(1);
    mas.RemoveAt(1);
    //Отобразим массив через визуальный элемент,
    //который работает с коллекциями через интрфейсы
    listBox1.Items.Clear();
    listBox1.ItemsSource = mas;

    //Скопируем элементы коллекции в обычный массив
    //Почемум тут коментарий
    //int [] m = (int[]) mas.ToArray(typeof ( int ));
```

Обобщение

Обобщенные классы

Кроме обычных типов фреймворк .NET также поддерживает ***обобщенные типы*** (generics), а также создание ***обобщенных методов***. Чтобы разобраться в особенности данного явления, сначала посмотрим на проблему, которая могла возникнуть до появления обобщенных типов. Посмотрим на примере. Допустим, мы определяем класс для представления банковского счета. К примеру, он мог бы выглядеть следующим образом:

```
class Account
{
    Ссылок: 0
    public int Id { get; set; }
    Ссылок: 0
    public int Sum { get; set; }
}
```

Класс ***Account*** определяет два свойства: ***Id*** - уникальный идентификатор и ***Sum*** - сумму на счете.

Здесь идентификатор задан как числовое значение, то есть банковские счета будут иметь значения 1, 2, 3, 4 и так далее. Однако также нередко для идентификатора используются и строковые значения. И у числовых, и у строковых значений есть свои плюсы и минусы. И на момент написания класса мы можем точно не знать, что лучше выбрать для хранения идентификатора - строки или числа. Либо, возможно, этот класс будет использоваться другими разработчиками, которые могут иметь свое мнение по данной проблеме.

И на первый взгляд, чтобы выйти из подобной ситуации, мы можем определить свойство ***Id*** как свойство типа ***object***. Так как тип ***object*** является универсальным типом, от которого наследуется все типы, соответственно в свойствах подобного типа мы можем сохранить и строки, и числа:

```
class Account
{
    Ссылок: 4
    public object Id { get; set; }
    Ссылок: 2
    public int Sum { get; set; }
}
```

Затем этот класс можно было использовать для создания банковских счетов в программе:

```
Account account1 = new Account { Sum = 5000 };
Account account2 = new Account { Sum = 4000 };
account1.Id = 2;
account2.Id = "4356";
int id1 = (int)account1.Id;
string id2 = (string)account2.Id;
```

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

Так, при присвоении свойству Id значения типа int, происходит упаковка этого значения в тип Object:

```
account1.Id = 2; // упаковка в значения int в тип Object
```

Чтобы обратно получить данные в переменную типов int, необходимо выполнить распаковку:

```
int id1 = (int)account1.Id; // Распаковка в тип int
```

Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.

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

```
Account account2 = new Account { Sum = 4000 };
account2.Id = "4356";
int id2 = (int)account2.Id; // Исключение InvalidCastException
```

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

Эти проблемы были призваны устранить ***обобщенные типы*** (также часто называют универсальными типами). Обобщенные типы позволяют указать конкретный тип, который будет использоваться.

Поэтому определим класс ***Account*** как ***обобщенный***:

```
class Account<T>
{
    Ссылок: 0
    public T Id { get; set; }
    Ссылок: 0
    public int Sum { get; set; }
}
```

Угловые скобки в описании ***class Account<T>*** указывают, что класс является обобщенным, а тип ***T***, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву ***T***, это может быть и любая другая буква или набор символов, например, ***<TypeId>*** . Причем сейчас нам неизвестно, что это будет за тип, это может быть любой тип. Поэтому параметр ***T*** в угловых скобках еще называется ***универсальным параметром***, так как вместо него можно подставить любой тип.

Например, вместо параметра ***T*** можно использовать объект ***int***, то есть число, представляющее номер счета. Это также может быть объект ***string***, либо или любой другой класс или структура:

```
Account<int> account1 = new Account<int> { Sum = 5000 };
Account<string> account2 = new Account<string> { Sum = 4000 };
account1.Id = 2;// упаковка не нужна
account2.Id = "4356";
int id1 = account1.Id;// распаковка не нужна
string id2 = account2.Id;
```

Поскольку класс ***Account*** является обобщенным, то при определении переменной после названия типа в угловых скобках необходимо указать тот тип, который будет использоваться вместо универсального параметра ***T***. В данном случае объекты **Account** типизируется типами ***int и string***.

Поэтому у первого объекта ***account1*** свойство ***Id*** будет иметь тип ***int,*** а у объекта ***account2*** - тип ***string***.

При попытке присвоить значение свойства ***Id*** переменной другого типа мы получим ошибку компиляции:

```
Account<string> account2 = new Account<string> { Sum = 4000 };
account2.Id = "4356";
int id1 = account2.Id;// Ошибка компиляции
```

Тем самым мы избежим проблем с типобезопасностью. Таким образом, используя обобщенный вариант класса, мы снижаем время на выполнение и количество потенциальных ошибок.

Обобщенные методы

Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. Например, определим обобщенный метод ***Swap***, который принимает параметры по ссылке и меняет их значения. При этом в данном случае не важно, какой тип представляют эти параметры.

```
// Обмен значений
//Вх.Вых. параметры x, y
Ссылок: 2
public static void Swap<T>(ref T x, ref T y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

```
private void button1_Click(object sender, EventArgs e)
{
    int x = 7;
    int y = 25;
    Swap<int>(ref x, ref y); // или так Swap(ref x, ref y);

    string s1 = "hello";
    string s2 = "bye";
    Swap<string>(ref s1, ref s2); // или так Swap(ref s1, ref s2);
```

Обобщенная коллекция List<T>

В классе ***List<T>*** реализуется обобщенный динамический массив. Он ничем принципиально не отличается от класса необобщенной коллекции ***ArrayList***.

Таблица 7. Основные методы класса ***List<T>***

Методы	***Описание***
void Add(T item)	Добавление нового элемента в список
void AddRange(IEnumerable<T> collection)	Добавляет элементы указанной коллекции в конец списка ***List<T>***
void Clear()	Удаляет из списка все элементы
bool Contains(T item)	Проверяет, содержится ли в списке объект ***item***. Если содержится, возвращает ***true***, иначе возвращает ***false***
void CopyTo(T[] array)	Копирует текущий список в массив ***array***
List<T> GetRange(int index, int count)	Возвращает новый список ***List<T>***, который содержит ***count*** элементов текущего списка, начиная с индекса ***index***
int IndexOf(T item)	Возвращает индекс элемента ***item***
int IndexOf(T item, int index)	Осуществляет поиск указанного объекта и возвращает отсчитываемый от нуля индекс первого вхождения в диапазоне элементов списка ***List<T>***, начиная с заданного индекса и до последнего элемента
void Insert(int index, T item)	Вставляет в список по индексу ***index*** объект ***item***
void InsertRange(int index, IEnumerable<T> collection)	Вставляет элементы коллекции в список ***List<T>*** в позиции с указанным индексом
int LastIndexOf(T item)	Возвращает индекс последнего вхождения в списке объекта ***item***
void Remove(T item)	Удаляет из списка объект ***item***
void RemoveAt(int index)	Удаляет из списка элемент по индексу ***index***
void RemoveRange(int index, int count)	Удаляет из списка ***count*** элементов, начиная с индекса ***index***
void Reverse()	Переворачивает список
void Sort()	Сортирует коллекцию
T[] ToArray()	Копирует элементы списка ***List<T>*** в новый массив

Рассмотрим пример использования коллекции ***List<T>***:

```
List<int> mas = new List<int>();

ссылка: 1
private void button1_Click(object sender, EventArgs e)
{
    // Добавим элементы в массив
    mas.Add(1);
    mas.Add(2);
    mas.Add(3);
    mas.Add(4);
    mas.Add(5);
    MessageBox.Show("Количество элементов - " + mas.Count);
    //Отобразим элементы
    for (int i = 0; i < mas.Count; i++) MessageBox.Show(mas[i].ToString());

    //Удалить элементы
    mas.Remove(1);
    mas.RemoveAt(1);

    //Отобразим массив через визуальный элемент,
    //который работает с коллекциями через интрфейсы
    listBox1.Items.Clear();
    listBox1.ItemsSource = mas;

    //Скопируем элементы коллекции в обычный массив
    int[] m = mas.ToArray();
```

Обобщенная коллекция Queue<T>

Класс ***Queue<T>*** представляет обычную очередь, работающую по алгоритму ***FIFO*** ("первый вошел - первый вышел").

У класса ***Queue<T>*** можно отметить следующие методы:

- ***Dequeue***: извлекает и возвращает первый элемент очереди;
- ***Enqueue***: добавляет элемент в конец очереди;
- ***Peek***: просто возвращает первый элемент из начала очереди без его удаления.

Таблица 8. Основные методы класса ***Queue <T>***

Методы	***Описание***
void Clear()	Удаляет из списка все элементы
bool Contains(T item)	Проверяет, содержится ли в списке объект ***item***. Если содержится, возвращает ***true***, иначе возвращает ***false***
void CopyTo(T[] array, int arrayIndex)	Копирует текущий список в массив ***array*** начиная с указанного значения индекса массива
T Dequeue ()	Удаляет объект из начала очереди ***Queue<T>*** и возвращает его
void Enqueue (T item)	Добавляет объект в конец коллекции ***Queue<T>***
T Peek ()	Возвращает объект, находящийся в начале очереди ***Queue<T>,*** но не удаляет его
bool TryDequeue (out T result)	Удаляет объект в начале ***Queue<T>*** и копирует его в параметр ***result***. Значение ***true***, если объект удален успешно;

	значение ***false***, если ***Queue<T>*** пуст
bool TryPeek (out T result)	Возвращает значение, указывающее, имеется ли в начале ***Queue<T>*** объект, и если он присутствует, копирует его в параметр ***result***. Объект не удаляется из ***Queue<T>***
T[] ToArray()	Копирует элементы списка ***Queue <T>*** в новый массив

Рассмотрим пример использования коллекции очередь ***Queue <T>***:

```
class Person
{
    Ссылок: 3
    public string Name { get; set; }
    Ссылок: 3
    public int Age { get; set; }
}
Queue<Person> mas = new Queue<Person>();
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    // Добавим элементы в конец очереди
    Person p = new Person();
    p.Name = textBox1.Text;
    p.Age = Convert.ToInt32(textBox2.Text);
    mas.Enqueue(p);
    MessageBox.Show("Количество элементов - " + mas.Count);
}
private void Button_Click_2(object sender, RoutedEventArgs e)
{
    //Отобразим элементы
    Person p;
    p = mas.Peek();
    MessageBox.Show(p.Name + " " + p.Age);

    //Удалить и извлечь 1 элемент
    mas.Dequeue();

    //Отобразим массив через визуальный элемент,
    //который работает с коллекциями через интрфейсы
    listBox1.Items.Clear();
    Person[] m = mas.ToArray();
    for (int i = 0; i < m.Length; i++)
        listBox1.Items.Add(m[i].Name + " " + m[i].Age);
}
```

Обобщенная коллекция Stack<T>

Класс ***Stack<T>*** представляет коллекцию, которая использует алгоритм ***LIFO*** ("последний вошел - первый вышел"). При такой организации каждый следующий добавленный элемент помещается поверх предыдущего. Извлечение из коллекции

происходит в обратном порядке - извлекается тот элемент, который находится выше всех в стеке.

В классе ***Stack*** можно выделить три основных метода, которые позволяют управлять элементами:

- ***Push***: добавляет элемент в стек на первое место
- ***Pop***: извлекает и возвращает первый элемент из стека
- ***Peek***: просто возвращает первый элемент из стека без его удаления

Таблица 9. Основные методы класса ***Stack <T>***

Методы	***Описание***
void Clear()	Удаляет из списка все элементы
bool Contains(T item)	Проверяет, содержится ли в списке объект ***item***. Если содержится, возвращает ***true***, иначе возвращает ***false***
void CopyTo(T[] array, int arrayIndex)	Копирует текущий список в массив ***array*** начиная с указанного значения индекса массива
T Peek ()	Возвращает объект, находящийся в начале стека ***Stack <T>,*** но не удаляет его
T Pop ()	Удаляет и возвращает объект, находящийся в начале ***Stack<T>***
void Push (T item)	Вставляет объект как верхний элемент стека ***Stack<T>***
bool TryPeek (out T result)	Возвращает значение, указывающее, имеется ли в начале ***Stack <T>*** объект, и если он присутствует, копирует его в параметр ***result***. Объект не удаляется из ***Stack <T>***
bool TryPop (out T result)	Возвращает значение, указывающее, имеется ли в верхней части ***Stack<T>*** объект, и если он присутствует, копирует его в параметр result и удаляет из ***Stack<T>***
T[] ToArray()	Копирует элементы списка ***Stack <T>*** в новый массив

Рассмотрим пример использования коллекции стек ***Stack <T>,*** по сути, работа с ним осуществляется также как и в коллекции очередь:

```
class Person {
    Ссылок: 3
    public string Name { get; set; }
    Ссылок: 3
    public int Age { get; set; }
}
Stack<Person> mas = new Stack<Person>();
private void button1_Click(object sender, EventArgs e)
{
    // Добавим элементы в конец очереди
    Person p = new Person();
    p.Name = textBox1.Text;
    p.Age = Convert.ToInt32(textBox2.Text);
    mas.Push(p);
    MessageBox.Show("Количество элементов - " + mas.Count);
}
```

```
private void Button_Click_2(object sender, RoutedEventArgs e)
{
    //Отобразим элементы
    Person p;
    p = mas.Peek();
    MessageBox.Show(p.Name + " " + p.Age);

    //Удалить и извлечь 1 элемент
    mas.Pop();

    //Отобразим массив через визуальный элемент,
    //который работает с коллекциями через интрфейсы
    listBox1.Items.Clear();
    Person[] m = mas.ToArray();
    for (int i = 0; i < m.Length; i++)
        listBox1.Items.Add(m[i].Name + " " + m[i].Age);
}
```

Обобщенная коллекция Dictonary<TKey, TValue>

Еще один распространенный тип коллекции представляют словари. Словарь хранит объекты, которые представляют пару ключ-значение. Каждый такой объект является объектом структуры ***KeyValuePair<TKey, TValue>***. Благодаря свойствам ***Key и Value***, которые есть у данной структуры, мы можем получить ключ и значение элемента в словаре.

Таблица 10. Основные методы класса ***Dictonary< TKey,TValue >***

Методы	***Описание***
void Add(TKey key, TValue value)	Добавляет указанные ключ и значение в словарь
void Clear()	Удаляет из списка все элементы
bool ContainsKey (TKey key)	Определяет, содержится ли указанный ключ в словаре ***Dictionary<TKey,TValue>***. Если содержится, возвращает ***true***, иначе возвращает ***false***
bool ContainsValue (TValue value)	Определяет, содержится ли указанное значение в словаре ***Dictionary<TKey,TValue>***. Если содержится, возвращает ***true***, иначе возвращает ***false***
bool Remove (TKey key)	Удаляет значение с указанным ключом из словаря ***Dictionary<TKey,TValue>***
bool Remove (TKey key, out TValue value)	Удаляет значение с указанным ключом из объекта ***Dictionary<TKey,TValue>*** и копирует элемент в параметр ***value***
bool TryAdd (TKey key, TValue value)	Пытается добавить указанную пару "ключ-значение" в словарь. Значение ***true***, если пара "ключ-значение" была успешно добавлена в словарь; в противном случае — значение ***false***
bool TryGetValue (TKey key, out TValue value)	Получает значение, связанное с заданным ключом. ***true***, если ***Dictionary<TKey,TValue>*** содержит элемент с указанным ключом, в противном случае — f***alse***

Рассмотрим пример использования коллекции очередь ***Dictonary< TKey,TValue >***:

```
Dictionary<int, string> mas = new Dictionary<int, string>();
ссылка: 1
private void button1_Click(object sender, EventArgs e)
{
    // Добавим элементы в словарь
    //Список регионов России
    int Kod = Convert.ToInt32(textBox1.Text);
    string Region = textBox2.Text;
    mas.Add(Kod, Region);
    MessageBox.Show("Количество элементов - " + mas.Count);
}

//Получение элемента по ключу
string region = mas[62];

//Изменение по ключу
mas[77] = "Новая Москва";
mas.TryGetValue(77, out region);
MessageBox.Show("77" + " " + region);
//Удаление по ключу
mas.Remove(62);

//Отобразим массив через визуальный элемент,
listBox1.Items.Clear();
int[] key= new int [mas.Count];
mas.Keys.CopyTo(key, 0);
for (int i = 0; i < mas.Count; i++)
{
    mas.TryGetValue(key[i], out region);
    listBox1.Items.Add(key[i] .ToString() + " " + region);
}

foreach (KeyValuePair<int, string> keyValue in mas)
{
    listBox2.Items.Add(keyValue.Key + " - " + keyValue.Value);
}
```

Практическая работа №10

Коллекции. Обобщённые типы.

1. Использовать коллекцию для выполнения задания. Оформить модули комментариями.
2. Предусмотреть в программе две кнопки «**Выход**» и «**О программе**», где вывести ФИО разработчика, номер работы и формулировку задания.
3. Оформить программу комментариями.

Варианты заданий (Базовый уровень)

1. В первом одномерном массиве хранятся затраты на производство продуктов, во втором — цены на эти продукты. Указать номер первого продукта, затраты на производство которого превышают цены.
2. Составьте программу вычисления в массиве суммы всех чисел, кратных 7.
3. Дан массив в диапазоне [0;10]найти количество значений равных 5 и 7.
4. В одномерном массиве целых чисел найти максимальный среди элементов, являющихся четными, и минимальный среди элементов, кратных А.
5. Дан массив в диапазоне [-100;100] найти количество положительных и отрицательных.
6. В массиве чисел найти наибольший элемент и поменять его местами с первым элементом.
7. Дан массив в диапазоне [-30;100] найти минимальное и максимальное значение и обменять их местами.
8. Составьте программу вычисления в массиве суммы всех чисел, кратных 3.
9. Составьте программу вычисления среднего арифметического отрицательных элементов.
10. Заданы два массива. Проверить, сколько элементов первого массива превосходят соответствующие элементы второго массива.
11. Составьте программу вычисления в массиве максимального среди отрицательных элементов и его номера.
12. Проверить, имеется ли в одномерном массиве хотя бы один элемент, попадающий в интервал [a; b]
13. Дан массив в диапазоне [-100;100] найти количество четных и нечетных.
14. Дан массив в диапазоне [-10;10] найти количество значений равных 5 и -5.
15. Проверить, имеется ли в одномерном массиве два соседних элемента, отличающихся друг от друга не более чем на два.
16. Проверить, состоит ли заданный одномерный массив только из нечетных чисел.

Регулярные выражения

Общие сведения

Класс ***Strin***g предоставляют достаточную функциональность для работы со строками. Однако ***.NET*** предлагает еще один мощный инструмент - регулярные выражения. Регулярные выражения представляют эффективный и гибкий метод по обработке больших текстов, позволяя в то же время существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками.

Комплексная нотация сопоставления шаблонов регулярных выражений позволяет быстро анализировать большие объемы текста в следующих целях:

- поиск определенных шаблонов символов;
- проверка текста на соответствие предопределенному шаблону (например, адресу электронной почты);
- извлечение, изменение, замена или удаление текстовых подстрок;
- добавление извлеченных строк в коллекцию для создания отчета.

Для многих приложений, которые работают со строками или анализируют большие блоки текста, регулярные выражения — незаменимый инструмент.

Основная функциональность регулярных выражений в ***.NET*** сосредоточена в пространстве имен ***System.Text.RegularExpressions***. А центральным классом при работе с регулярными выражениями является класс ***Regex***.

Таблица 11. Основные методы класса ***Regex***

Методы	***Описание***
`bool IsMatch (string input)`	Указывает, обнаружено ли в указанной входной строке соответствие регулярному выражению, заданному в конструкторе ***Regex***.
`bool IsMatch (string input, int startat)`	Указывает, обнаружено ли в указанной входной строке соответствие (начинающееся с указанной позиции в этой строке) регулярному выражению, заданному в конструкторе ***Regex***
`bool IsMatch (string input, string pattern)`	Указывает, обнаружено ли в указанной входной строке соответствие заданному регулярному выражению.
`bool IsMatch (string input, string pattern, RegexOptions options)`	Указывает, обнаружено ли в указанной входной строке соответствие заданному регулярному выражению, используя указанные параметры сопоставления
`Match Match (string input)`	Ищет в указанной входной строке первое вхождение регулярного выражения, указанного в конструкторе **Regex**. ***Match*** - объект, содержащий сведения о совпадении.
`Match Match (string input, int startat)`	Ищет во входной строке первое вхождение регулярного выражения, начиная с указанной начальной позиции
`Match Match (string input, int beginning, int length)`	Ищет во входной строке первое вхождение регулярного выражения, начиная с указанной начальной позиции и выполняя поиск только по указанному количеству символов
`Match Match (string input, string pattern)`	Ищет в указанной входной строке первое вхождение заданного регулярного выражения
`Match Match (string input, string pattern, RegexOptions options)`	Ищет во входной строке первое вхождение заданного регулярного выражения, используя указанные параметры сопоставления
`MatchCollection Matches (string input)` `MatchCollection Matches`	Ищет в указанной входной строке все вхождения регулярного выражения. ***MatchCollection*** Коллекция объектов ***Match***, найденных при поиске. Если

(string input, int startat) MatchCollection Matches (string input, string pattern) MatchCollection Matches (string input, string pattern, RegexOptions options)	соответствующие объекты не найдены, метод возвращает пустой объект коллекции

Как мы видим ряд методов в качестве одного из параметров принимают перечисление ***RegexOptions***.

Таблица 12. Параметр RegexOptions

Параметр	***Описание***
Compiled	При установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение
IgnoreCase	При установке этого значения будет игнорироваться регистр
IgnorePatternWhitespace	удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака #
Multiline	Указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы "^" и "$" совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста
RightToLeft	Приписывает читать строку справа налево
Singleline	Устанавливает однострочный режим, а весь текст рассматривается как одна строка

Например, у нас есть некоторый текст и нам надо найти в нем все словоформы какого-нибудь слова. С классом ***Regex*** это сделать очень просто:

```
string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";
//Ищем слова туп и далее любые символы
Regex regex = new Regex(@"туп(\w*)");
MatchCollection matches = regex.Matches(s);
if (matches.Count > 0)
{
    object[] mas = new object[matches.Count];
    matches.CopyTo(mas, 0);
    listBox1.ItemsSource = mas;
    // for (int i = 0; i < matches.Count; i++)
    //     listBox1.Items.Add(matches[i]);
}
else
{
    MessageBox.Show("Совпадений не найдено");
}
```

Синтаксис регулярных выражений

Для построения шаблона регулярного выражения необходимо описать строку символов, по которой выполняется сравнение на совпадение регулярного выражения с текстом строки.

Например, найдем в заданной строке слово «Бык»:

```
Regex regex = new Regex("Бык");
string s = "Бык Быкчок тупогуб, тупогубенький бычок, " +
    "у быка губа бела была тупа";
Match match = regex.Match(s);
MessageBox.Show(match.Value + " " + match.Index);
MatchCollection matches = regex.Matches(s);
if (matches.Count > 0)
{
    object[] mas = new object[matches.Count];
    matches.CopyTo(mas, 0);
    listBox1.ItemsSource = mas;
}
```

При выполнении этого примера функция ***regex.Match(s)*** вернет первое найденное слово «Бык» и его позицию. Функция ***regex.Matches(s)*** вернет список найденных слов «Бык», в данном случае это два слова «Бык». Обратите внимание регистр имеет значение.

Метасимволы

Для более сложных регулярных выражений используются метасимволы, которые расширяют возможности построения шаблонов сравнения.

Таблица 13. Классы символов

Символ	Значение	Пример	Соответствует
[...]	Любой из символов, указанных в скобках	[a-z] или [abcd]	В исходной строке может быть любой символ английского алфавита в нижнем регистре или только указанные буквы
[^...]	Любой из символов, не указанных в скобках	[^0-9]	В исходной строке может быть любой символ кроме цифр
.	Любой символ, кроме перевода строки или другого разделителя Unicode-строки	w..k	Соответствует словам work, week
\w	Любой текстовый символ [A-Za-z0-9_], не являющийся пробелом, символом табуляции и т.п.	w\w\wk	Соответствует словам work, week
\W	Любой символ, не являющийся текстовым символом	w\W\Wk	Соответствует слову w22k
\s	Любой пробельный символ из набора Unicode	\w\w\s	Соответствует словам «да », «на »
\S	Любой непробельный символ из набора Unicode. Обратите		

	внимание, что символы \w и \S - это не одно и то же		
\d	Любые ASCII-цифры. Эквивалентно [0-9]	\d\d[:]\d\d	Соответствует слову 12:22
\D	Любой символ, отличный от ASCII-цифр. Эквивалентно [^0-9]		

Рассмотрим примеры использования метасимволов:

Точка (.) представляет собой любой символ. В следующем примере мы ищем символ b, за которым следует любой символ, а за ним символ g:

***Регулярное выражение*: b.g**

***Пример*:** The big bag of bits was bugged.

В примере мы сопоставляем букву l, за которой следуют любые два символа, и за которыми следует e.

***Регулярное выражение*: l..e**

***Пример*:** You can live like a king but make sure it isn't a lie.

В регулярном выражении мы ищем символ t, за которым следует либо символ e, либо o, и за которым следует символ d.

***Регулярное выражение*: t[eo]d**

***Пример*:** When today is over Ted will have a tedious time tidying up.

Допустим, мы хотим найти наличие цифр от 1 до 8.

***Регулярное выражение*: [1-8]**

***Пример*:** Room Allocations: G4 G9 F2 H1 L0 K7 M9

Мы также можем объединить несколько наборов. В регулярном выражении ниже мы ищем 1, 2, 3, 4, 5, a, b, c, d, e, f, x:

***Регулярное выражение*: [1-5a-fx]**

***Пример*:** A random set of characters: y, w, a, r, f, 4, 9, 6, 3, p, x, t

Экранирование

Иногда нам может потребоваться найти один из символов, который является метасимволом. Для этого мы используем то, что называется **экранированием**. Поместив обратную косую черту (\) перед метасимволом, мы можем удалить его особое значение.

Допустим, что нам нужно найти появление слова «this.».

Если бы мы сделали следующее:

***Регулярное выражение*: this.**

***Пример*:** Surely this regular expression should match this.

То нашли бы «this.» как в конце предложения, так и в начале, потому что точка в регулярном выражении обычно соответствует любому символу.

Решением будет следующее:

***Регулярное выражение*: this\.**

***Пример*:** Surely this regular expression should match this.

Квантификаторы

Квантор указывает количество вхождений предшествующего элемента (знака, группы или класса знаков), которое должно присутствовать во входной строке, чтобы было зафиксировано соответствие.

Таблица 14. Квантификаторы

Символ	Значение	Пример	Соответствует
{n,m}	Соответствует предшествующему шаблону, повторенному не менее n и не более m раз	s{2,4}	"Press", "ssl", "progressss"
{n,}	Соответствует предшествующему шаблону, повторенному n или более раз	s{1,}	"Press", "ssl", "progressss"
{n}	Соответствует в точности n экземплярам предшествующего шаблона	s{2}	"Press", "ssl", но не "progressss"
?	Соответствует нулю или одному экземпляру предшествующего шаблона; предшествующий шаблон является необязательным	Эквивалентно {0,1}	
+	Соответствует одному или более экземплярам предшествующего шаблона	Эквивалентно {1,}	
*	Соответствует нулю или более экземплярам предшествующего шаблона	Эквивалентно {0,}	

Рассмотрим примеры использования квантификаторов:

В примере мы ищем символ l, за которым следует символ o ноль или более раз.
Регулярное выражение*: lo
***Пример*:** Are you looking at the lock or the silk?

В примере выше мы ищем символ l, за которым следует символ o ноль или более раз. Вот почему l в слове silk также является совпадением (это l, за которым идёт символ o ноль раз). Возникает вопрос, как избежать этого?

Написать шаблон, который соответствует всем трём строкам: abcdefg, abcde, abc.
Регулярное выражение*: abc\w

Написать шаблон, который будет соответствовать всем цифрам: abc123xyz, define "123", var g = 123;.
***Регулярное выражение*: 123**

Написать шаблон, который будет соответствовать первым трем строкам, но не соответствовать последней:

Соответствовать cats.
Соответствовать 8967.
Соответствовать ?=+!.
Пропустить abcd1
***Регулярное выражение*: Напишите сами 2 способами ( )**

Написать шаблон, который в строчках ниже сопоставляет только первые три строки, последние три строки мы сопоставлять не хотим.

Соответствовать can
Соответствовать man
Соответствовать fan
Пропустить dan
Пропустить ran
Пропустить pan
***Регулярное выражение*: Напишите сами ( )**

Написать шаблон, который соответствует только первым двум написаниям этого слова.

Соответствовать wazzzzzup
Соответствовать wazzzup
Пропустить wazup
***Регулярное выражение*: Напишите сами ( )**

Написать шаблон, который соответствует только первым трем написаниям этого слова.

Соответствовать aaaabcc
Соответствовать aabbbbc
Соответствовать aacc
Пропустить a
***Регулярное выражение*: Напишите сами ( )**

Можно использовать метасимвол для квантификации, например:
***Регулярное выражение*: l.*k**
***Пример*:** Are you looking at the lock or the silk?

На первый взгляд такой результат может показаться вам немного странным. Регулярное выражение .* соответствует повторению ноль или более раз любого символа. Вы могли бы подумать, что при нахождении первого k, будет сказано «да, я нашёл совпадение», но на самом деле говориться, что «k является любым символом, поэтому давайте посмотрим, на сколько длинным может быть соответствие», и поиск продолжится, пока не будет найдет последний k в строке.

Это то, что называется **жадным сопоставлением**. Это нормальное поведение — пытаться найти самую большую строку, которая может соответствовать шаблону. Мы можем изменить это поведение и сделать его не жадным, поместив вопросительный знак (?) после квантификатора (что может показаться немного запутанным, поскольку вопросительный знак сам по себе является множителем):

***Регулярное выражение*: l.*?k**
***Пример*:** Are you looking at the lock or the silk?

Метасимволы привязки

При определении регулярных выражений, можно ссылаться на конкретные местоположения в строке — это начало и конец строки или границы слова:

Таблица 16. Метасимволы привязки

Символ	Значение	Пример	Соответствует
^	Соответствует началу строкового выражения или началу строки при многострочном поиске.	^Hello	"Hello, world", но не "Ok, Hello world" т.к. в этой строке слово "Hello"

			находится не в начале
$	Соответствует концу строкового выражения или концу строки при многострочном поиске.	Hello$	"World, Hello Hello", соответствует второму "Hello"
\b	Соответствует границе слова, т.е. соответствует позиции между символом \w и символом \W или между символом \w и началом или концом строки.	\b(my)\b	В строке "Hello my world, mysql." выберет слово "my"
\B	Соответствует позиции, не являющейся границей слов.	\B(ld)\b	Соответствие найдется в слове "World", но не в слове "ld"

Рассмотрим примеры использования квантификаторов:

Найти число, но, при условии, что оно является самым первым элементом в строке:
***Регулярное выражение*: ^\d+**
***Пример*:** 13 cats escaped from the 5 cages at the vet's clinic.

Найти строки, которые содержат только одно слово, которое может быть либо bat, либо bit, либо but:
***Регулярное выражение*: ^b[aiu]t$**
***Пример (строка №1)*:** This line does not match but the next line does.
***Пример (строка №2)*:** bat

Группирование

Мы можем сгруппировать несколько символов в нашем **регулярном выражении**, используя круглые скобки (). Затем мы можем делать разные вещи с этой группой символов (например, добавить мультипликаторы).

Допустим, нам нужно узнать, упоминается ли какой-то конкретный человек. Его могут звать Джон Реджинальд Смит, но второе имя может присутствовать, а может и нет:
***Регулярное выражение*: John (Reginald)?Smith**
***Пример*:** John Reginald Smith is sometime just called John Smith.

Обратите внимание на пробелы в регулярном выражении выше. Важно помнить, что они являются частью вашего регулярного выражения, и вы должны убедиться, что они находятся в нужных местах. Это является очень важным, так как это распространённый источник проблем, возникающих при использовании регулярных выражений новичками. Например:

Регулярное выражение: John (Reginald)? Smith
Пример: Проблема с этим регулярным выражением заключается в том, что оно найдёт соответствие John Reginald Smith и даже John Smith (два пробела между John и Smith), но не найдёт соответствие в John Smith. Можете ли вы понять, почему?

Вы не ограничены только обычными символами в скобках. Вы можете использовать в скобках и **метасимволы** (включая мультипликаторы).

Например, нам нужно найти экземпляры IP-адресов. IP-адрес представляет собой набор из 4-х чисел (от 0 до 255), разделённых точками (например, 192.168.0.5):

***Регулярное выражение*: \b(\d{1,3}\.){3}\d{1,3}\b**
***Пример*:** The server has an address of 10.18.0.20 and the printer has an address of 10.18.0.116.

Это уже немного сложнее, поэтому давайте разберёмся детальнее:

- \b обозначает границу слова;
- в скобках мы обрабатываем первые 3 части IP-адреса: \d{1,3} указывает на то, что мы ищем от 1 до 3 цифр + точку, следующую за этими цифрами, которую мы не забываем экранировать (\.). Мы ищем ровно 3 таких последовательности, поэтому указываем мультипликатор {3} сразу за скобками;
- наконец, мы ищем 4-е число, указывая \d{1,3}, и обозначаем конечную границу слова (\b).

Обратные ссылки

Всякий раз, когда мы сопоставляем что-либо в круглых скобках, это значение фактически сохраняется в переменной, к которой мы можем обратиться позже в регулярном выражении. Для доступа к этой переменной мы используем escape-символ (\), за которым следует цифра. Первый набор скобок обозначается как \1, второй — \2 и т.д.

Нужно найти строки с двумя упоминаниями о человеке с фамилией Smith. При этом, мы не знаем его имени. Можно сделать следующее:

***Регулярное выражение*: (\b[A-Z]\w+\b) Smith.*\1 Smith**
***Пример*:** Harold Smith went to meet John Smith but John Smith was not there.

В примере выше мы сопоставляем текст между двумя экземплярами Джона Смита, но, в этом случае, это нормально, поскольку мы не слишком обеспокоены тем, **что** было сопоставлено, нам важно только **наличие** совпадения.

Чередование и оператор ИЛИ

При чередовании мы имеем возможность искать что-либо из указанного нами списка. Мы видели очень простой пример чередования с использованием оператора диапазона. Это позволяет нам выполнять поиск одного символа, но иногда нам может быть нужно, выполнить операцию и с большим набором символов. Это делается с помощью **оператора ИЛИ** (|).

Например, нам нужно найти все экземпляры dog или cat:
***Регулярное выражение*: dog|cat**
***Пример*:** Harold Smith has two dogs and one cat.

Мы также можем использовать несколько операторов ИЛИ для использования большего количества вариантов:
***Регулярное выражение*: dog|cat|bird**
***Пример*:** Harold Smith has two dogs, one cat and three birds.

Иногда нам может быть нужно, чтобы чередование происходило только с частью регулярного выражения. Для этого используются круглые скобки:
***Регулярное выражение:* (John|Harold) Smith**
Пример: Harold Smith went to meet John Smith but instead bumped into Jane Smith.

Примеры использования регулярных выражений

Пример 1. Проверить содержит строка время в 24-часовом формате и получить это время.

Регулярное выражение: **(([01]\d)|(2[0-3])):[0-5]\d**

Пример: Время завтрака 07:00, время обеда 12:00, время ужина 20:00, текущее время 8:00

Итак, время в 24-часовом формате может принимать, например, следующие значения: 06:00, 11:15, 23:59 и т.д. Первые две цифры (hh) обозначают часы с ведущим нулем (т.е. пишем 06, а не 6). Значение часов может быть от 0 (т.е. 00) до 23. Поэтому мы пишем следующее выражение:

[01]\d

- что означает, что первый символ может быть 0 или 1, а следующий символ может принимать значение от 0 до 9 (т.е. любая цифра). т.е. любые значения от 00 до 19 подходят под эту регулярку.

Теперь рассмотрим значения часов от 20 до 23. Если первый символ - двойка, то следующий символ может быть цифрой от 0 до 3. Это нас приводит к выражению

2[0-3]

- под него подходят числа 20, 21, 22, 23. Объединяя эти два выражения (с помощью логического "ИЛИ" - |)для часов, получаем первую часть нашей регулярки (для проверки только часов):

([01]\d)|(2[0-3])

Идем дальше: теперь нужно поставить двоеточие и проверять минуты. Первую часть регулярки (проверка часов) заключим в круглые скобки и после них поставим двоеточие:

(([0,1][0-9])|(2[0-3])):(здесь_будет_проверка_минут)

С минутами все просто - минуты принимают значение от 0 (т.е. 00) до 59. Мы это опишем следующей регуляркой:

[0-5]\d

```
Regex regex = new Regex("(([01]\\d)|(2[0-3])):[0-5]\\d");
string s = textBox1.Text;
Match match = regex.Match(s);
if (match.Success)
MessageBox.Show("Время "+match.Value);
```

Пример 2. Проверить является полученная строка временем в 24-часовом формате.

Регулярное выражение: **^(([01]\d|(2[0-3])):[0-5]\d$**

Пример 1: Время завтрака 07:00

Пример 2: 07:00

```
Regex regex = new Regex("^(([01]\\d)|(2[0-3])):[0-5]\\d$");
string s = textBox1.Text;
if (regex.IsMatch(s))
MessageBox.Show("Время "+s + " правильное");
```

Пример 3. Сложить все числа в строке.
Регулярное выражение: **\b(\d+)\b**

Итак, нам нужны числа, которые состоят из одной и более цифр (\d+) расположенных отдельным словом \b.

```
Regex regex = new Regex("\\b(\\d+)\\b");
string s = textBox1.Text;
MatchCollection matches = regex.Matches(s);
int sum = 0;
for (int i = 0; i < matches.Count; i++)
{
    sum = sum + Convert.ToInt32(matches[i].Value);
}
MessageBox.Show("Сумма= " + sum);
//Отобразим найденные числа
object[] mas = new object[matches.Count];
matches.CopyTo(mas, 0);
listBox1.ItemsSource = mas;
```

Практическая работа №11

Использование регулярных выражений.

1. Разработать программу для выполнения задания. Оформить модули комментариями.
2. Предусмотреть в программе две кнопки «**Выход**» и «**О программе**», где вывести ФИО разработчика, номер работы и формулировку задания.
3. Оформить программу комментариями.

Варианты заданий (Базовый уровень)

1. Дана строка 'ahb acb aeb aeeb adcb axeb'. Напишите регулярное выражение, которое найдет строки ahb, acb, aeb.
   Дана строка 'aa a1a a22a a333a a4444a a55555a aba aca'. Напишите регулярное выражение, которое найдет строки, в которых по краям стоят буквы 'a', а между ними любое количество цифр (в том числе и ноль цифр, то есть строка 'aa').
2. Дана строка 'aba aca aea abba adca abea'. Напишите регулярное выражение, которое найдет строки abba adca abea.
   Дана строка 'ave a#b a2b a$b a4b a5b a-b acb'. Напишите регулярное выражение, которое найдет строки следующего вида: по краям стоят буквы 'a' и 'b', а между ними - не буква и не цифра.
3. Дана строка 'aba aca aea abba adca abea'. Напишите регулярное выражение, которое найдет строки abba и abea, не захватив adca.
   Напишите регулярное выражение, которое найдет строки следующего вида: по краям стоят буквы 'a', а между ними - буква от a до f и от j до z.
4. Дана строка 'aa aba abba abbba abca abea'. Напишите регулярное выражение, которое найдет строки aba, abba, abbba.
   Дана строка '23 2+3 2++3 2+++3 445 677'. Напишите регулярное выражение, которое найдет строки 23, 2+3, 2++3, 2+++3, не захватив остальные.
5. Дана строка 'aa aba abba abbba abca abea'. Напишите регулярное выражение, которое найдет строки aa, aba, abba, abbba.
   Дана строка 'a1a a2a a3a a4a a5a aba aca'. Напишите регулярное выражение, которое найдет строки, в которых по краям стоят буквы 'a', а между ними одна цифра.
6. Дана строка 'aa aba abba abbba abca abea'. Напишите регулярное выражение, которое найдет строки aa, aba.
   Напишите регулярное выражение, которое найдет строки следующего вида: по краям стоят буквы 'a', а между ними - цифра от 3-х до 7-ми.
7. Дана строка 'aa aba abba abbba abca abea'. Напишите регулярное выражение, которое найдет строки aa, aba, abba, abbba, не захватив abca abea.
   Напишите регулярное выражение, которое найдет строки следующего вида: по краям стоят буквы 'a', а между ними - буква от a до f .
8. Дана строка 'ab abab abab abababab abea'. Напишите регулярное выражение, которое найдет строки 'ab' повторяется 1 или более раз.
   Дана строка 'a.a aba aea'. Напишите регулярное выражение, которое найдет строку a.a, не захватив остальные.
9. Дана строка '2+3 223 2223'. Напишите регулярное выражение, которое найдет строку 2+3, не захватив остальные.
   Дана строка 'aa aba abba abbba abbbba abbbbba'. Напишите регулярное выражение, которое найдет строки abba, abbba, abbbba и только их.

10. Дана строка '23 2+3 2++3 2+++3 345 567'. Напишите регулярное выражение, которое найдет строки 2+3, 2++3, 2+++3, не захватив остальные (+ может быть любое количество).
    Дана строка 'aa aba abba abbba abbbba abbbbba'. Напишите регулярное выражение, которое найдет строки вида aba, в которых 'b' встречается менее 3-х раз (включительно).
11. Дана строка '23 2+3 2++3 2+++3 445 677'. Напишите регулярное выражение, которое найдет строки 23, 2+3, 2++3, 2+++3, не захватив остальные.
    Дана строка '*+ *q+ *qq+ *qqq+ *qqq qqq+'. Напишите регулярное выражение, которое найдет строки *q+, *qq+, *qqq+, не захватив остальные.
12. Дана строка 'a1a a22a a333a a4444a a55555a aba aca'. Напишите регулярное выражение, которое найдет строки, в которых по краям стоят буквы 'a', а между ними любое количество цифр.
    Дана строка 'aba aea aca aza axa a-a a#a'. Напишите регулярное выражение, которое найдет строки следующего вида: по краям стоят буквы 'a', а между ними - не 'e' и не 'x'.
13. Дана строка 'aa aba abba abbba abbbba abbbbba'. Напишите регулярное выражение, которое найдет строки вида aba, в которых 'b' встречается более 4-х раз (включительно).
    Дана строка 'avb a1b a2b a3b a4b a5b abb acb'. Напишите регулярное выражение, которое найдет строки следующего вида: по краям стоят буквы 'a' и 'b', а между ними - не число.
14. Дана строка 'ave a#b a2b a$b a4b a5b a-b acb'. Напишите регулярное выражение, которое найдет строки следующего вида: по краям стоят буквы 'a' и 'b', а между ними - не буква и не цифра.
    Напишите регулярное выражение, м найдет строки следующего вида: по краям стоят буквы 'a', а между ними - буква от a до g.
15. Дана строка 'aba aea aca aza axa'. Напишите регулярное выражение, которое найдет строки aba, aea, axa, не затронув остальных.
    Напишите регулярное выражение, которое найдет строки следующего вида: по краям стоят буквы 'a', а между ними - буква от a до f и от j до z.
16. Дана строка 'aba aea aca aza axa a.a a+a a*a'. Напишите регулярное выражение, которое найдет строки aba, a.a, a+a, a*a, не затронув остальных.
    Дана строка 'aAXa aeffa aGha aza ax23a a3sSa'. Напишите регулярное выражение, которое найдет строки следующего вида: по краям стоят буквы 'a', а между ними - маленькие латинские буквы, не затронув остальных. Скрыть решение.