Лабораторная работа

8. Работа с Windows-приложениями

Варианты заданий

Выполнить задания 1-2 по предложенным вариантам:

	№ Задания			
	1	2	3	
Вариант 1	1.II	2.III	3.I	
Вариант 2	1.III	2.I	3.III	
Вариант 3	1.I	2.II	3.II	
Вариант 4	1.II	2.I	3.III	
Вариант 5	1.III	2.II	3.II	
Вариант 6	1.I	2.III	3.I	
Вариант 7	1.II	2.I	3.II	
Вариант 8	1.III	2.III	3.I	
Вариант 9	1.I	2.II	3.III	
Вариант 10	1.II	2.II	3.II	

Задание 1.

Создать Windows-приложение для заполнения массива в элементе DataGridView.

I Даны числа n и m. Создайте массив int A[n, m] и заполните следующим образом (пример для n=5, m=6):

```
0  0  0  0  0  0
0  1  2  3  4  5
0  2  4  6  8 10
0  3  6  9 12 15
0  4  8 12 16 20
```

II Даны числа n и m. Создайте массив int A[n, m] и заполните его следующим образом (ниже приведен пример для n=4 и m=6):

```
 0   1   3   6  10  14
 2   4   7  11  15  18
 5   8  12  16  19  21
 9  13  17  20  22  23
```

III Даны числа n и m. Создайте массив int A[n, m] и заполните следующим образом (пример для n=5, m=6):

```
0 1 2 3 4 5
1 0 1 2 3 4
2 1 0 1 2 3
3 2 1 0 1 2
4 3 2 1 0 1
```

Задание 2.

Разработать приложение, демонстрирующее возможности следующих классов:

I Класс «запись», содержащий следующие закрытые поля: ФИО, номер телефона, email, дата рождения. Предусмотреть свойства для получения состояния объекта.

Класс «записная книжка», содержащий закрытый массив записей. Обеспечить:

• вывод на экран информации о человеке, номер телефона которого введен (если такого нет, то выдать соответствующее сообщение);

• поиск людей, день рождения которых сегодня или в заданный день;

• поиск людей, день рождения которых будет в этом месяце;

• поиск людей, номер телефона которых начинается на три заданных цифры.

II Класс «товар», содержащий следующие закрытые поля: название товара, целочисленный код товара, количество, стоимость товара в рублях. Предусмотреть свойства для получения состояния объекта.

Класс «склад», содержащий закрытый массив товаров. Обеспечить:

• вывод информации о товаре по номеру с помощью индекса;

• вывод на экран информации о товаре, название которого введено с клавиатуры;

если таких товаров нет, выдать соответствующее сообщение;

• сортировку товаров по наименованию, по количеству и по цене.

III Класс «самолет», содержащий следующие закрытые поля: название пункта назначения; двухсимвольный код авиакомпании, целочисленный номер рейса; время отправления. Предусмотреть свойства для получения состояния объекта.

Описать класс «аэропорт», содержащий закрытый массив самолетов. Обеспечить возможности:

• вывод информации о самолете по номеру рейса с помощью индекса;

• вывод информации о самолетах, отправляющихся в течение часа после введенного с клавиатуры времени;

• вывод информации о самолетах, отправляющихся в заданный пункт назначения;

Информация должна быть отсортирована по времени отправления.

Задание 3.

I Исходный текст представляет описание класса на C#. Напишите процедуру, удаляющую из этого текста теги summary и комментарии. Для обработки текстов используйте методы классов char [], string и StringBuilder.

II Исходный текст представляет описание класса на C#. Напишите процедуру, создающую массив строк, каждая из которых содержит описание одного из методов класса. Для обработки текстов используйте методы классов char [], string и StringBuilder.

III Исходный текст представляет описание класса на C#. Напишите процедуру, создающую массив строк, каждая из которых содержит описание одного из полей класса. Для обработки текстов используйте методы классов char [], string и StringBuilder.

Методические указания

Оглавление

8.1. DataGridView

См. Lab7 - 7.3.3. Элемент управления DataGridView и отображение массивов

8.2. Свойства

Методы, называемые **свойствами (Properties)**, представляют специальную синтаксическую конструкцию, предназначенную для обеспечения эффективной работы со свойствами. При работе со *свойствами объекта* (*полями*) часто нужно решить, какой *модификатор доступа* использовать, чтобы реализовать нужную стратегию доступа к *полю* класса. Перечислю пять наиболее употребительных стратегий:

- чтение, запись (`Read`, `Write`);
- чтение, запись при первом обращении (`Read`, `Write-once`);
- только чтение (`Read-only`);
- только запись (`Write-only`);
- ни чтения, ни записи (`Not Read`, `Not Write`).

Открытость свойств (атрибут `public`) позволяет реализовать только первую стратегию. В языке C# принято, как и в других объектных языках, свойства объявлять закрытыми, а нужную стратегию доступа организовывать через *методы*. Для эффективности этого процесса и введены специальные *методы-свойства*.

Приведу вначале пример, а потом уточню синтаксис этих *методов*. Рассмотрим класс `Person`, у которого пять *полей*: `fam`, `status`, `salary`, `age`, `health`, характеризующих соответственно фамилию, статус, зарплату, возраст и здоровье персоны. Для каждого из этих *полей* может быть разумной своя стратегия доступа. Возраст доступен для чтения и записи, фамилию можно задать только один раз, статус можно только читать, зарплата недоступна для чтения, а здоровье закрыто для доступа и только специальные *методы* класса могут сообщать некоторую информацию о здоровье персоны. Вот как на C# можно обеспечить эти стратегии доступа к закрытым *полям класса*:

```
public class Person
{
    //поля (все закрыты)
    string fam = "", status = "", health = "";
    int age = 0, salary = 0;
    //методы - свойства
    /// <summary>
    ///стратегия: Read,Write-once (Чтение, запись при
    ///первом обращении)
    /// </summary>
    public string Fam
    {
        set { if (fam == "") fam = value; }
        get { return (fam); }
    }
    /// <summary>
    ///стратегия: Read-only(Только чтение)
    /// </summary>
    public string Status
    {
        get { return (status); }
    }
    /// <summary>
    ///стратегия: Read,Write (Чтение, запись)
    /// </summary>
    public int Age
    {
        set
        {
            age = value;
            if (age < 7) status = "ребенок";
            else if (age < 17) status = "школьник";
            else if (age < 22) status = "студент";
            else status = "служащий";
        }
        get { return (age); }
    }
    /// <summary>
    ///стратегия: Write-only (Только запись)
    /// </summary>
    public int Salary
    {
        set { salary = value; }
    }
}
```

Рассмотрим теперь общий синтаксис методов-свойств. Пусть name - это закрытое свойство. Тогда для него можно определить открытый метод-свойство (функцию), возвращающую тот же тип, что и поле name. Имя метода обычно близко к имени поля (например, Name). Тело свойства содержит два метода - get и set, один из которых может быть опущен. Метод get возвращает значение закрытого поля, метод set - устанавливает значение, используя передаваемое ему значение в момент вызова, хранящееся в служебной переменной со стандартным именем value. Поскольку get и set - это обычные процедуры языка, то программно можно реализовать сколь угодно сложные стратегии доступа. В нашем примере фамилия меняется, только если ее значение равно пустой строке и это означает, что фамилия персоны ни разу еще не задавалась. Статус персоны пересчитывается автоматически при всяком изменении возраста, явно изменять его нельзя. Вот пример, показывающий, как некоторый клиент создает и работает с полями персоны:

```
public void TestPersonProps()
{
      Person pers1 = new Person();
      pers1.Fam = "Петров";
      pers1.Age = 21;
      pers1.Salary = 1000;
      Console.WriteLine ("Фам={0}, возраст={1}, статус={2}",
            pers1.Fam, pers1.Age, pers1.Status);
      pers1.Fam = "Иванов"; pers1.Age += 1;
      Console.WriteLine ("Фам={0}, возраст={1}, статус={2}",
            pers1.Fam, pers1.Age, pers1.Status);
}//TestPersonProps
```

Заметьте, клиент работает с *методами-свойствами* так, словно они являются настоящими *полями*, вызывая их как в правой, так и в левой части оператора присваивания. Заметьте также, что с каждым *полем* можно работать только в *полном* соответствии с той стратегией, которую реализует данное свойство. Попытка изменения фамилии не принесет успеха, а изменение возраста приведет и к одновременному изменению статуса. На рис. 8.1 показаны результаты работы этой процедуры.

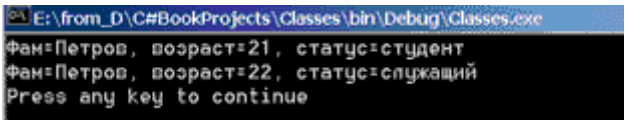

Рис. 8.1. Методы-свойства и стратегии доступа к полям

Индексаторы

Свойства являются частным случаем *метода класса* с особым синтаксисом. Еще одним частным случаем является *индексатор*. Метод-индексатор является обобщением *метода-свойства*. Он обеспечивает доступ к закрытому *полю*, представляющему массив. Объекты класса индексируются по этому *полю*.

Синтаксически объявление *индексатора* - такое же, как и в случае свойств, но *методы* `get` и `set` приобретают аргументы по числу размерности массива, задающего индексы элемента, значение которого читается или обновляется. Важным ограничением является то, что у класса может быть только *индексатор* со стандартным именем `this`. Как и

любые другие методы индексатор может быть перегруженным. Так что если среди *полей* класса есть несколько массивов одной размерности, то индексация объектов может быть выполнена только по одному из них.

Добавим в класс `Person` свойство `children`, задающее детей персоны, сделаем это свойство закрытым, а доступ к нему обеспечит *индексатор*:

```
const int Child_Max = 20; //максимальное число детей
Person[] children = new Person[Child_Max];
int count_children=0; //число детей
public Person this[int i] //индексатор
{
      get {if (i>=0 && i< count_children)return(children[i]);
                  else return(children[0]);}
      set
      {
            if (i==count_children && i< Child_Max)
             {children[i] = value; count_children++;}
      }
}
```

Имя у *индексатора* - `this`, в квадратных скобках в заголовке перечисляются индексы. В *методах* `get` и `set`, обеспечивающих доступ к массиву `children`, по которому ведется индексирование, анализируется корректность задания индекса. Закрытое *поле* `count_children`, хранящее текущее число детей, доступно только для чтения благодаря добавлению соответствующего *метода-свойства*. Надеюсь, текст процедуры-свойства Count_children сумеете написать самостоятельно. Запись в это *поле* происходит в *методе* `set` *индексатора*, когда к массиву `children` добавляется новый элемент.

Протестируем процесс добавления детей персоны и работу *индексатора*:

```
public void TestPersonChildren()
{
      Person pers1 = new Person(), pers2 = new Person();
      pers1.Fam = "Петров"; pers1.Age = 42;
            pers1.Salary = 10000;
      pers1[pers1.Count_children] = pers2;
      pers2.Fam ="Петров"; pers2.Age = 21; pers2.Salary = 1000;
      Person pers3= new Person("Петрова");
      pers1[pers1.Count_children] = pers3;
      pers3.Fam ="Петрова"; pers3.Age = 5;
      Console.WriteLine ("Фам={0}, возраст={1}, статус={2}",
            pers1.Fam, pers1.Age, pers1.Status);
      Console.WriteLine ("Сын={0}, возраст={1}, статус={2}",
            pers1[0].Fam, pers1[0].Age, pers1[0].Status);
      Console.WriteLine ("Дочь={0}, возраст={1}, статус={2}",
            pers1[1].Fam, pers1[1].Age, pers1[1].Status);
}
```

Заметьте, *индексатор* создает из объекта как бы массив объектов, индексированный по соответствующему *полю*, в данном случае по *полю* `children`. На рис. 8.2 показаны результаты вывода.

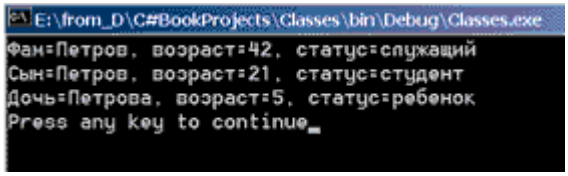

Рис. 8.2. Работа с индексатором класса

Персоны и профессии

Рассмотрим еще один пример работы с перечислениями, приближенный к реальности. Рассмотрим класс `Person (см. Lab6 – 6.2 Классы и перечисления)`. Сделаем поле `Profession` закрытым, а доступ к нему обеспечим соответствующим свойством:

```
Profession prof;
public Profession Prof
 {
   get {return (prof);}
   set {prof = value;}
 }
```

Рассмотрим перечисление `Status`, элементы которого задают возможный статус персоны:

```
    public enum Status
    {
        ребенок, школьник,
        студент, работник, пенсионер
    }
```

Зададим в классе `Person` поле `status`, принадлежащее перечислению `Status`:

```
        Status status = Status.студент;
```

Предположим, что статус персоны изменяется с возрастом, определив метод-свойство для поля `age` следующим образом:

```
/// <summary>
///стратегия: Read,Write (Чтение, запись)
/// </summary>
public int Age
{
  set
  {
     age = value;
     //Изменение статуса
     if (age < 7) status = Status.ребенок;
     else if (age < 17) status = Status.школьник;
     else if (age < 22) status = Status.студент;
     else if (age < 65) status = Status.работник;
     else status = Status.пенсионер;
  }
```

```
  get { return (age); }
}
```

Соответствующий тест, демонстрирующий работу с полем `status`, имеет вид:

```
public void TestStatus()
  {
     Person pers = new Person("Кузнецов");
     pers.Age = 5;
     Console.WriteLine("возраст = {0}, статус = {1}",
         pers.Age, pers.GetStatus);
     pers.Age = 20;
     Console.WriteLine("возраст = {0}, статус = {1}",
         pers.Age, pers.GetStatus);
     pers.Age = 35;
     Console.WriteLine("возраст = {0}, статус = {1}",
         pers.Age, pers.GetStatus);
  }
```

В этом примере `GetStatus` - это метод-свойство, обеспечивающий доступ к закрытому полю `status`. Результаты работы с объектами перечислений, полученные при вызове тестов `TestProfession` и `TestStatus`, показаны на рис. 8.3.

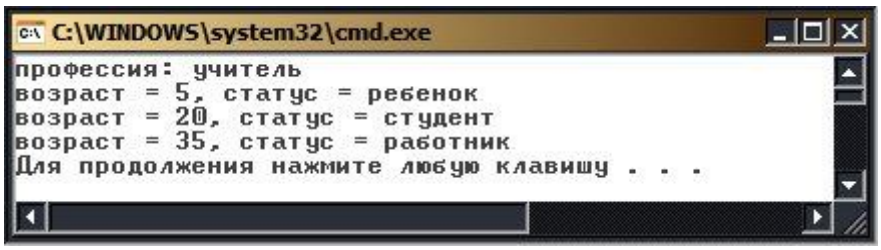

Рис. 8.3. Результаты работы с перечислениями Profession и Status

8.3. Класс StringBuilder - построитель строк

Рассмотрим подробнее возможности класса StringBuilder (см. раздел 5.2.3).

Операция конкатенации (+) не определена над строками класса `StringBuilder`, ее роль играет метод `Append`, дописывающий новую строку в хвост уже существующей. Семантика операций частично изменилась. Присваивание для строк этого класса является полноценным ссылочным присваиванием, так что изменение значения строки сказывается на всех экземплярах, ссылающихся на строку в динамической памяти. Эквивалентность теперь является проверкой ссылок, а не значений. Со строкой этого класса можно работать как с массивом, но, в отличие от класса `string`, здесь уже все делается как надо: допускается не только чтение отдельного символа, но и его изменение. Рассмотрим пример работы со строками, используя строки класса `StringBuilder`:

```
/// <summary>
  /// Операции над строками StringBuilder
  /// </summary>
  public void TestStringBuilder()
```

```
{
    string DEL = "->";
    StringBuilder s1 = new StringBuilder("ABC"),
        s2 = new StringBuilder("CDE");
    StringBuilder s3 = s2.Insert(0,s1.ToString());
    s3.Remove(3, 3);
    bool b1 = (s1 == s3);
    char ch1 = s1[2];
    string s = s1.ToString() + DEL + s2.ToString() +
     DEL + s3.ToString() + DEL +
     b1.ToString() + DEL + ch1.ToString();
    Console.WriteLine(s);

    s2.Replace("ABC", "Zenon");
    s1 = s2;
    s2[0] = 'L';
    s1.Append(" - это музыкант!");
    Console.WriteLine(s1.ToString() +
        " -> " + s2.ToString());
}
```

Результаты работы этого метода показаны на рис. 8.4.

Рис. 8.4. Тип StringBuilder - это изменяемый тип

Этот пример демонстрирует возможность выполнения над строками класса `StringBuilder` тех же операций, что и над строками класса `string`. Обратите внимание, теперь методы, изменяющие строку, `Replace`, `Insert`, `Remove`, `Append` реализованы как процедуры, а не как функции. Они изменяют значение строки непосредственно в буфере, отводимом для хранения строки. Появляется новая возможность - изменять отдельные символы строки.

Основные методы

У класса `StringBuilder` методов значительно меньше, чем у класса `string`. Это и понятно: класс создавался с целью дать возможность изменять значение строки. По этой причине у класса есть основные методы, позволяющие выполнять такие операции над строкой, как вставка, удаление и замена подстрок, но нет методов, подобных поиску вхождения, которые можно выполнять над обычными строками. Технология работы обычно такова: создается обычная строка; из нее конструируется строка класса `StringBuilder`; выполняются операции, требующие изменение значения; полученная строка преобразуется в строку класса `string`; над этой строкой выполняются операции, не требующие изменения значения строки.

Давайте чуть более подробно рассмотрим основные методы класса StringBuilder:

- public StringBuilder Append(<объект>); К строке, вызвавшей метод, присоединяется строка, полученная из объекта, который передан методу в качестве параметра. Метод перегружен и может принимать на входе объекты всех простых типов, начиная от char и bool до string и long. Поскольку объекты всех этих типов имеют метод ToString, всегда есть возможность преобразовать объект в строку, которая и присоединяется к исходной строке. В качестве результата возвращается ссылка на объект, вызвавший метод. Поскольку возвращаемую ссылку ничему присваивать не нужно, то правильнее считать, что метод изменяет значение строки;
- public StringBuilder Insert(int location,<объект>); Метод вставляет строку, полученную из объекта, в позицию, указанную параметром location. Метод Append является частным случаем метода Insert;
- public StringBuilder Remove(int start, int len); Метод удаляет подстроку длины len, начинающуюся с позиции start;
- public StringBuilder Replace(string str1,string str2); Все вхождения подстроки str1 заменяются на строку str2;
- public StringBuilder AppendFormat(<строка форматов>, <объекты>); Метод является комбинацией метода Format класса string и метода Append. Строка форматов, переданная методу, содержит только спецификации форматов. В соответствии с этими спецификациями находятся и форматируются объекты. Полученные в результате форматирования строки присоединяются в конец исходной строки.

За исключением метода `Remove`, все рассмотренные методы являются перегруженными. В представленном описании приведен основной вариант вызова метода, не отражающий точный синтаксис всех перегруженных реализаций.

Емкость буфера

Каждый экземпляр строки класса `StringBuilder` имеет буфер, в котором хранится строка. Объем буфера - его емкость - может меняться в процессе работы со строкой. Объекты класса имеют две характеристики емкости - текущую и максимальную. В процессе работы текущая емкость изменяется, естественно, в пределах максимальной емкости, которая реально достаточно высока. Если размер строки увеличивается, то соответственно автоматически растет и текущая емкость. Если же размер строки уменьшается, то емкость буфера остается на том же уровне. По этой причине иногда разумно уменьшать емкость. Следует помнить, что попытка уменьшить емкость до величины, меньшей длины строки, приведет к ошибке.

У класса `StringBuilder` имеется 2 свойства и один метод, позволяющие анализировать и управлять емкостными свойствами буфера. Напомню, что этими характеристиками можно управлять также еще на этапе создания объекта, - для этого имеется соответствующий конструктор. Рассмотрим свойства и метод класса, связанные с емкостью буфера:

- свойство Capacity - возвращает или устанавливает текущую емкость буфера;
- свойство MaxCapacity - возвращает максимальную емкость буфера. Результат один и тот же для всех экземпляров класса;
- метод int EnsureCapacity(int capacity) - позволяет убедиться, что емкость буфера не меньше емкости, заданной параметром capacity; если текущая емкость меньше, то она увеличивается до значения capacity, иначе не изменяется. Максимум текущей емкости и capacity возвращается в качестве результата работы метода.

Приведу код, в котором проводятся различные эксперименты с емкостью буфера:

```
/// <summary>
/// Анализ емкости буфера
/// </summary>
public void TestCapacity()
{
  string txt = "А это пшеница, которая в темном чулане хранится," +
  " в доме, который построил Джек!";
    string str = "А роза упала на лапу Азора";
    StringBuilder strbuild = new StringBuilder(100, 1000);
    StringBuilder txtbuild = new StringBuilder(txt);
    strbuild.Append(str);
    //Емкость буфера
    Console.WriteLine("strbuild: емкость буфера = {0}, " +
    "максимальная емкость = {1}",
    strbuild.Capacity, strbuild.MaxCapacity);
    Console.WriteLine("txtbuild: емкость буфера = {0}, " +
    "максимальная емкость = {1}",
    txtbuild.Capacity, txtbuild.MaxCapacity);
    //Изменение емкости
    //Ошибка периода выполнения!
    //попытка установить емкость меньше длины строки
    //txtbuild.Capacity = 75;
    int sure = txtbuild.EnsureCapacity(75);
    Console.WriteLine("sure= {0}", sure);
    // увеличим строку за пределы буфера
    // емкость автоматически увеличится!
    txtbuild.Append(txtbuild.ToString());
    Console.WriteLine("txtbuild: емкость буфера = {0}",
    txtbuild.Capacity);
}
```

В этом фрагменте кода анализируются и изменяются емкостные свойства буфера двух объектов. Демонстрируется, как меняется емкость при работе с объектами. Результаты работы этого фрагмента кода показаны на рис. 8.5.

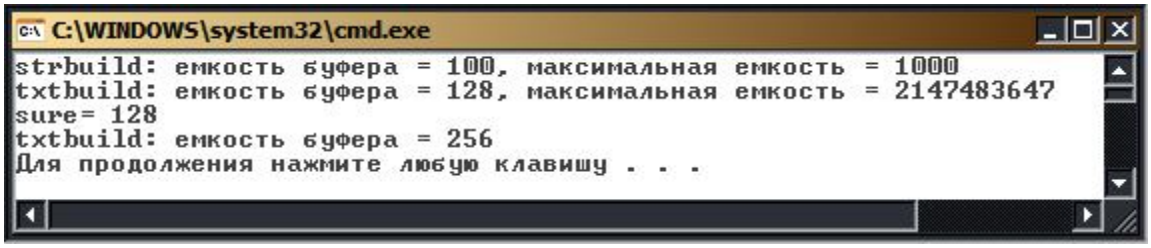

Рис. 8.5. Анализ емкостных свойств буфера

Источники:

1. Биллиг В. Основы программирования на C#. Режим доступа: http://www.intuit.ru/studies/courses/2247/18/info
2. Биллиг В. Основы программирования на C# 3.0. Ядро языка. Режим доступа: http://www.intuit.ru/studies/courses/1094/428/info