Теория и практика программирования

Шпилёв Пётр Валерьевич

Санкт-Петербургский государственный университет
Математико-механический факультет
Кафедра статистического моделирования

Лекция 5

Санкт-Петербург
2015 г.

e-mail: pitshp@hotmail.com

Общая форма одномерного индексатора:

```
тип_элемента this[int индекс]
{
    // Аксессор для получения данных.
    get
    {
        // Возврат значения, которое определяет индекс.
    }
    // Аксессор для установки данных.
    set
    {
        // Установка значения, которое определяет индекс.
    }
}
```

Общая форма одномерного индексатора:

```
тип_элемента this[int индекс]
{
    // Аксессор для получения данных.
    get
    {
        // Возврат значения, которое определяет индекс.
    }
    // Аксессор для установки данных.
    set
    {
        // Установка значения, которое определяет индекс.
    }
}
```

Упражнение 5.1

Использовать индексатор для создания отказоустойчивого массива.

Общая форма свойства

```
тип имя
{
    get
    {
        // код аксессора для чтения из поля
    }
    set
    {
        // код аксессора для записи в поле
    }
```

Пример:

```
// Простой пример применения свойства.
using System;
class SimpProp
{
    int prop; // поле, управляемое свойством MyProp
    public SimpProp(){ prop = 0; }
    /* Это свойство обеспечивает доступ к закрытой переменной
    экземпляра prop.
    Оно допускает присваивание только положительных значений. */
    public int MyProp
    {
        get
        {
            return prop;
        }
        set
        {
            if(value >= 0) prop = value;
        }
    }
```

Общая форма свойства

```
тип имя
{
    доступ get
    {
        // код аксессора для чтения из поля
    }
    доступ set
    {
        // код аксессора для записи в поле
    }
```

Общая форма свойства

```
тип имя
{
    доступ get
    {
        // код аксессора для чтения из поля
    }
    доступ set
    {
        // код аксессора для записи в поле
    }
```

Автоматически реализуемые свойства

```
тип имя { доступ get;доступ set; }
```

Общая форма свойства

```
тип имя
{
    доступ get
    {
        // код аксессора для чтения из поля
    }
    доступ set
    {
        // код аксессора для записи в поле
    }
```

Автоматически реализуемые свойства

```
тип имя { доступ get;доступ set; }
```

Пример:

```
public int UserCount { get; private set; }
```

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие модификаторам доступа в свойствах:

- действию модификатора доступа подлежит только один аксессор: set или get, но не оба сразу.
- модификатор должен обеспечивать более ограниченный доступ к аксессору, чем доступ на уровне свойства или индексатора
- модификатор доступа нельзя использовать при объявлении аксессора в интерфейсе или же при реализации аксессора, указываемого в интерфейсе

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие модификаторам доступа в свойствах:

- действию модификатора доступа подлежит только один аксессор: set или get, но не оба сразу.
- модификатор должен обеспечивать более ограниченный доступ к аксессору, чем доступ на уровне свойства или индексатора
- модификатор доступа нельзя использовать при объявлении аксессора в интерфейсе или же при реализации аксессора, указываемого в интерфейсе

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие модификаторам доступа в свойствах:

- действию модификатора доступа подлежит только один аксессор: set или get, но не оба сразу.
- модификатор должен обеспечивать более ограниченный доступ к аксессору, чем доступ на уровне свойства или индексатора
- модификатор доступа нельзя использовать при объявлении аксессора в интерфейсе или же при реализации аксессора, указываемого в интерфейсе

Ограничения, присущие свойствам:

- свойство не может быть передано методу в качестве параметра ref или out.
- свойство не подлежит перегрузке
- Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило
- свойство не должно изменять состояние базовой переменной при вызове аксессора get

Ограничения, присущие модификаторам доступа в свойствах:

- действию модификатора доступа подлежит только один аксессор: set или get, но не оба сразу.
- модификатор должен обеспечивать более ограниченный доступ к аксессору, чем доступ на уровне свойства или индексатора
- модификатор доступа нельзя использовать при объявлении аксессора в интерфейсе или же при реализации аксессора, указываемого в интерфейсе

Упражнение 5.2

Заменить в одной из программ из предыдущих упражнений поля(поле) свойствами(свойством). Обосновать целесообразность данной замены.

Упражнение 5.2

Заменить в одной из программ из предыдущих упражнений поля(поле) свойствами(свойством). Обосновать целесообразность данной замены.

Упражнение 5.3

С помощью свойств и индексаторов реализовать класс для вычисления миноров матриц

Общая форма объявления класса наследования

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

Общая форма объявления класса наследования

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

Пример:

```
// Класс для двумерных объектов.
class TwoDShape
{
    public double Width;
    public double Height;
    public void ShowDim()
    {
        Console.WriteLine("Ширина и высота равны " + Width + " и "
        + Height);
    }
}
```

Пример:

```
// Класс для прямоугольников, производный от класса TwoDShape.
class Rectangle : TwoDShape
{
     // Возвратить логическое значение true, если
     // прямоугольник является квадратом.
    public bool IsSquare()
    {
        if(Width == Height) return true;
        return false;
    }
     // Возвратить площадь прямоугольника.
    public double Area()
    {
        return Width * Height;
    }
}
```

Пример:

```
// Класс для прямоугольников, производный от класса TwoDShape.
class Rectangle : TwoDShape
{
     // Возвратить логическое значение true, если
     // прямоугольник является квадратом.
    public bool IsSquare()
    {
        if(Width == Height) return true;
        return false;
    }
     // Возвратить площадь прямоугольника.
    public double Area()
    {
        return Width * Height;
    }
}
```

Замечание

Закрытый член класса остается закрытым в своем классе. Он не доступен из кода за пределами своего класса, включая и производные классы.

Пример:

```
//Продемонстрировать применение модификатора доступа protected.
using System;
class B
{
    //члены, закрытые для класса B, но доступные для класса D
    protected int i, j;
}
class D : B
{
    int k; //закрытый член
    public void Setk()
    {
        k = i * j; //члены i и j класса B доступны для класса D
    }
    public void Showk()
    {
        Console.WriteLine(k);
    }
}
```

Общая форма объявления конструктора производного класса

```
конструктор_производного_класса(список_параметров) :
base(список_аргументов)
{
// тело конструктора
}
```

Общая форма объявления конструктора производного класса

```
конструктор_производного_класса(список_параметров) :
base(список_аргументов)
{
// тело конструктора
}
```

Упражнение 5.4

Для любого из ранее реализованных классов создать производный класс по своему усмотрению и сделать для него конструктор общего вида.

Пример:
```
// Пример сокрытия имени с наследственной связью.
using System;
class A { public int i = 0;} // Создать производный класс.
class B : A
{
    new int i; // этот член скрывает член i из класса A
    public B(int b)
    {
        i = b; // член i в классе B
    }
    public void Show()
    {Console.WriteLine("Член i в производном классе: " + i);}
}
class NameHiding
{
    static void Main()
    {
        B ob = new B(2);
        ob.Show();
    }
}
```

Пример: // Пример сокрытия имени с наследственной связью.

```
using System;
class A { public int i = 0;} // Создать производный класс.
class B : A
```

Замечание

Доступ к скрытому имени базового класса осуществляется с помощью ключевого слова base:

base.имя_члена

```
    }
    public void Show()
    {Console.WriteLine("Член i в производном классе: " + i);}
}
class NameHiding
{
    static void Main()
    {
        B ob = new B(2);
        ob.Show();
    }
}
```

Пример: // Пример сокрытия имени с наследственной связью.

```
using System;
class A { public int i = 0;} // Создать производный класс.
class B : A
```

Замечание

Доступ к скрытому имени базового класса осуществляется с помощью ключевого слова base:

base.имя_члена

```
    }
```

Упражнение 5.5

Продемонстрировать порядок вызова конструкторов на примере многоуровневой иерархии производных классов (не менее двух от базового).

```
    static void Main()
    {
        B ob = new B(2);
        ob.Show();
    }
}
```

Пример: //По ссылке на объект базового класса можно обращаться
//к объекту производного класса.

```
using System;
class X { public int a=1; }
class Y : X{ public int b=2; }
class BaseRef
{
  static void Main()
  {
    X x = new X();
    X x2;
    Y y = new Y();
    x2 = x; //верно, т.к. оба объекта относятся к одному и тому же типу
    Console.WriteLine("x2.a: " + x2.a);
    x2 = y; //верно, поскольку класс Y является производным от класса X
    Console.WriteLine("x2.a: " + x2.a);
     //ссылкам на объекты класса X известно только о членах класса X
    x2.a = 19; //верно
    // x2.b = 27;//неверно, поскольку член b отсутствует у класса X
  }
}
```

Пример: //По ссылке на объект базового класса можно обращаться
//к объекту производного класса.

```
using System;
class X { public int a=1; }
```

Упражнение 5.6

Реализовать конструктор производного класса, позволяющий создавать копию объекта того же класса.

```
  {
    X x = new X();
    X x2;
    Y y = new Y();
    x2 = x; //верно, т.к. оба объекта относятся к одному и тому же типу
    Console.WriteLine("x2.a: " + x2.a);
    x2 = y; //верно, поскольку класс Y является производным от класса X
    Console.WriteLine("x2.a: " + x2.a);
     //ссылкам на объекты класса X известно только о членах класса X
    x2.a = 19; //верно
    // x2.b = 27;//неверно, поскольку член b отсутствует у класса X
  }
}
```

Пример:

```
// Продемонстрировать виртуальный метод.
using System;
class Base
{
    // Создать виртуальный метод в базовом классе.
    public virtual void Who()
    {
        Console.WriteLine("Метод Who() в классе Base");
    }
}
class Derivedl : Base
{
    // Переопределить метод Who() в производном классе.
    public override void Who()
    {
        Console.WriteLine("Метод Who() в классе Derivedl");
    }
}
```

Замечание

Выбор виртуального метода который следует вызывать, осуществляется исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения.

Замечание

Выбор виртуального метода который следует вызывать, осуществляется исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения.

Замечание

Свойства и индексаторы также подлежат модификации ключевым словом virtual и переопределению ключевым словом override.

Замечание

Выбор виртуального метода который следует вызывать, осуществляется исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения.

Замечание

Свойства и индексаторы также подлежат модификации ключевым словом virtual и переопределению ключевым словом override.

Упражнение 5.7

Продемонстрировать вызовы виртуальных методов для различных производных классов от общего базового класса.

Замечание

Выбор виртуального метода который следует вызывать, осуществляется исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения.

Замечание

Свойства и индексаторы также подлежат модификации ключевым словом virtual и переопределению ключевым словом override.

Упражнение 5.7

Продемонстрировать вызовы виртуальных методов для различных производных классов от общего базового класса.

Упражнение 5.8

Реализовать виртуальные методы для случая многоуровневой иерархии производных классов.

Замечание

Выбор виртуального метода который следует вызывать, осуществляется исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения.

Замечание

Свойства и индексаторы также подлежат модификации ключевым словом virtual и переопределению ключевым словом override.

Упражнение 5.7

Продемонстрировать вызовы виртуальных методов для различных производных классов от общего базового класса.

Упражнение 5.8

Реализовать виртуальные методы для случая многоуровневой иерархии производных классов.

Упражнение 5.9

Продемонстрировать удобство использования виртуальных методов и производных классов на примере базового класса для 2D фигур и различных производных от него (круги, треугольники, прямоугольники и т.п.).

Общая форма абстрактного класса:

```
abstract class имя_класса
{
    //в классе должен быть хотя бы один абстрактный метод
    abstract тип имя(список_параметров);
    ...
}
```

Замечание

Можно создавать ссылки на объекты абстрактного класса, но объявлять(инициализировать) эти объекты уже нельзя.

Пример 1:

```
sealed class A
{ // ...}
class  B : A // ОШИБКА! Наследовать класс A нельзя
{ // ... }
```

Пример 1:

```
sealed class A
{ // ...}
class  B : A // ОШИБКА! Наследовать класс A нельзя
{ // ... }
```

Пример 2:

```
class B
{
    public virtual void MyMethod()
    { /* ... */ }
}
class D : B
{
    // Здесь герметизируется метод MyMethod() и
    // предотвращается его дальнейшее переопределение.
    sealed public override void MyMethod() { /* ... */ }
}
class X : D
{
    // Ошибка! Метод MyMethod() герметизирован!
    public override void MyMethod() { /* ... */ }
}
```

Таблица 1. Методы класса object

Метод	Назначение
`public virtual bool Equals(object ob)`	Определяет, является ли вызывающий объект таким же, как и объект, доступный по ссылке *ob*
`public static bool Equals(object objA, object objB)`	Определяет, является ли объект, доступный по ссылке *objA*, таким же, как и объект, доступный по ссылке *objB*
`protected Finalize()`	Выполняет завершающие действия перед "сборкой мусора". В C# метод `Finalize()` доступен посредством деструктора
`public virtual int GetHashCode()`	Возвращает хеш-код, связанный с вызывающим объектом
`public Type GetType()`	Получает тип объекта во время выполнения программы
`protected object MemberwiseClone()`	Выполняет неполное копирование объекта, т.е. копируются только члены, но не объекты, на которые ссылаются эти члены
`public static bool ReferenceEquals(obj objA, object objB)`	Определяет, делаются ли ссылки *objA* и *objB* на один и тот же объект
`public virtual string ToString()`	Возвращает строку, которая описывает объект

Таблица 1. Методы класса object

Метод	Назначение
`public virtual bool Equals(object ob)`	Определяет, является ли вызывающий объект таким же, как и объект, доступный по ссылке *ob*
`public static bool Equals(object objA, object objB)`	Определяет, является ли объект, доступный по ссылке *objA*, таким же, как и объект, доступный по ссылке *objB*
p	...у- ...м
p	...м
`GetHashCode()`	объектом
`public Type GetType()`	Получает тип объекта во время выполнения программы
`protected object MemberwiseClone()`	Выполняет неполное копирование объекта, т.е. копируются только члены, но не объекты, на которые ссылаются эти члены
`public static bool ReferenceEquals(obj objA, object objB)`	Определяет, делаются ли ссылки *objA* и *objB* на один и тот же объект
`public virtual string ToString()`	Возвращает строку, которая описывает объект

Упражнение 5.10

Продемонстрировать применение метода ToString() для любого своего класса

Упаковка

Присваивание ссылки на объект класса object переменной типа значения называется упаковкой. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект.

Упаковка

Присваивание ссылки на объект класса object переменной типа значения называется упаковкой. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект.

Распаковка

Распаковка представляет собой процесс извлечения упакованного значения из объекта. Это делается с помощью явного приведения типа ссылки на объект класса object к соответствующему типу значения.

Упаковка

Присваивание ссылки на объект класса object переменной типа значения называется упаковкой. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект.

Пример 1:

```
// Простой пример упаковки и распаковки.
using System;
class BoxingDemo
{
    static void Main()
    {
        int x;
        object obj;
        x = 10;
        obj = x; // упаковать значение переменной x в объект
        int y = (int)obj; // распаковать значение из объекта, доступного по
        // ссылке obj, в переменную типа int
        Console.WriteLine(y);
    }
}
```

Упаковка

Присваивание ссылки на объект класса object переменной типа значения называется упаковкой. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект.

Пример 2:

```
// Благодаря упаковке становится возможным вызов методов по значению!
using System;
class MethOnValue
{
    static void Main()
    {
        Console.WriteLine(10.ToString());
    }
}
```

Упаковка

Присваивание ссылки на объект класса object переменной типа значения называется упаковкой. Упаковка приводит к тому, что значение простого типа сохраняется в экземпляре объекта, т.е. "упаковывается" в объекте, который затем используется как и любой другой объект.

Пример 2:

```
// Благодаря упаковке становится возможным вызов методов по значению!
using System;
class MethOnValue
{
    static void Main()
    {
        Console.WriteLine(10.ToString());
    }
}
```

Упражнение 5.11

Использовать класс object для создания массива "обобщенного" типа