Лекция 5:
Операторы
Делегаты и события
Атрибуты
Свойства и индексаторы
Пространство имен

Обзор операторов

- Операторы и методы
- Операторы в C#

Операторы и методы

- **Использование методов**
  - Громоздкая, неудобная запись выражений
  - Увеличивается вероятность возникновения ошибок

```
myIntVar1 = Int.Add(myIntVar2,
                    Int.Add(Int.Add(myIntVar3,
                                    myIntVar4),
                            33));
```

- **Использование операторов**
  - Упрощается запись выражений

```
myIntVar1 = myIntVar2 + myIntVar3 + myIntVar4 + 33;
```

Возможности перегрузки операторов

+, –, !, ~, ++, —, true, false	Унарные операторы, допускающие перегрузку. true и false также являются операторами
+, –, *, /, %, &, |, ^, <<, >>	Бинарные операторы, допускающие перегрузку
==, !=, <, >, <=, >=	Операторы сравнения перегружаются
&&, ||	Условные логические операторы моделируются с использованием ранее переопределенных операторов **&** и |
[]	Оператор доступа к элементам массивов моделируется за счет индексаторов
()	Оператор преобразования реализуются с использованием ключевых слов **implicit** и **explicit**
+=, –=, *=, /=, %=, &=, =, ^=, <<=, >>=	Операторы не перегружаются, т.к невозможно перегрузить оператор присваивания
=, ., ?:, –>, new, is, sizeof, typeof	Операторы не перегружаются

Основы перегрузки операторов

- **Перегрузка операторов**
  - Перегружайте операторы только когда это действительно необходимо
- **Синтаксис перегрузки операторов**
  - operator **op**, где **op** – это оператор, который перегружается
- **Пример**

```
public static Time operator+(Time t1, Time t2)
{
    int newHours = t1.hours + t2.hours;
    int newMinutes = t1.minutes + t2.minutes;
    return new Time(newHours, newMinutes);
}
```

Перегрузка операторов сравнения

- **Операторы сравнения необходимо перегружать попарно**
  - < и >
  - <= и >=
  - == и !=
- **При перегрузке операторов == и != настоятельно рекомендуется переопределять (override) метод Equals**
- **Вместе с методом Equals необходимо переопределить также метод GetHashCode**

Пример перегрузки оператора ++

```
public class Point2D{
    public float x,    y;
    float xTmp, yTmp;
    public Point2D(){
        x    = 0;
        y    = 0;
        xTmp = 0;
        yTmp = 0;
     }
     public static Point2D operator++(Point2D par){
    // Фактическим координатам присваиваются
    //старые значения.
        par.x = par.xTmp++;
        par.y = par.yTmp++;
        return par;
    }
    public float getTempX(){
                return xTmp;
      }
      public float getTempY() {
        return yTmp;
      }
}
```

```
class Program{
    static void Main(string[] args){
        Point2D p = new Point2D();
        int i;
        //унарный плюс всегда постфиксный,
        for (i = 0; i < 10; i++){
                        ++p;
                        Console.WriteLine("{0:F3},{1:F3}",
                                p.x, p.y);
        }
    }
}
```

Перегрузка операторов преобразования типов

- **Перегруженные операторы преобразования типов**

```
public static explicit operator Time (float hours)
{ ... }
public static explicit operator float (Time t1)
{ ... }
public static implicit operator string (Time t1)
{ ... }
```

implicit – неявное преобразование типов

explicit – явное преобразование типов

- **Если в классе используется преобразование типа в строку**
  - В классе должен быть переопределен метод ToString

Пример перегрузки операторов преобразования типов

```
public class Point2D{
    public int x, y;
    public Point3D(){
        x = 0;          y = 0;     }
     public Point3D(int  xKey, int  yKey) {
        x = xKey;      y = yKey;     }
/* Операторная функция, в которой реализуется  алгоритм преобразования значения
типа Point3D в значение типа Point2D.  Это преобразование  осуществляется с
ЯВНЫМ указанием необходимости преобразования.  Принятие решения
относительно присутствия в объявлении ключевого слова explicit вместо implicit
оправдывается тем, что это преобразование сопровождается потерей  информации.
*/
    public static explicit operator Point2D(Point3D p3d){
        Point2D p2d = new Point2D();
        p2d.x = p3d.x;         p2d.y = p3d.y;
        return p2d;      }}
```

```
public class Point3D{
    public int x, y, z;
    public Point3D(){
        x = 0;          y = 0;          z = 0;     }
    public Point3D(int  xKey, int  yKey, int  zKey) {
        x = xKey;      y = yKey;      z = zKey;  }
/* Операторная функция, в которой реализуется алгоритм преобразования
значения типа Point2D в значение типа Point3D.  Это преобразование
осуществляется НЕЯВНО.  Если метод закомментировать, то мы не сможем в
методе main выполнить выражение:  p3d = p2d;*/
    public static implicit operator Point3D(Point2D p2d){
        Point3D p3d = new Point3D();
        p3d.x = p2d.x;      p3d.y = p2d.y;  p3d.z = 1;
        return p3d;
    }}
```

```
class Program{
    static void Main(string[] args){
        Point2D p = new Point2D(125,125);       Console.WriteLine("Point 2d: x = " + p2d.x + "; y=" + p2d.y );
        Point3D p3d;
 /* Этой ссылке присваивается значение в результате НЕЯВНОГО преобразования значения типа Point2D к типу Point3D*/
        p3d = p2d;                      Console.WriteLine("Point 3d: x = " + p3d.x + "; y=" + p3d.y + "; z=" + p3d.z);
        // Изменили значения полей объекта.
        p3d.x = p3d.x * 2;              p3d.y = p3d.y * 2;              p3d.z = 125;
        Console.WriteLine("Point 3d modified: x = " + p3d.x + "; y=" + p3d.y + "; z=" + p3d.z);
      /*Главное – появилась новая информация, которая будет потеряна в случае присвоения значения типа Point3D
значению типа Point2D. Ключевое слово explicit в объявлении соответствующего метода преобразования вынуждает
программиста подтверждать, что он в курсе возможных последствий этого преобразования.
      p2d = (Point2D)p3d;
      Console.WriteLine("Point 2d modified: x = " + p2d.x + "; y=" + p2d.y);        }     }}
```

Многократная перегрузка операторов

- **Один и тот же операторов можно перегрузить несколько раз**

```
public static Time operator+(Time t1, int hours)
{...}

public static Time operator+(Time t1, float hours)
{...}

public static Time operator-(Time t1, int hours)
{...}

public static Time operator-(Time t1, float hours)
{...}
```

Тест: Найдите ошибки

```
public bool operator != (Time t1, Time t2)
{ ... }
```

1

```
public static operator float(Time t1) { ... }
```

2

```
public static Time operator += (Time t1, Time t2)
{ ... }
```

3

```
public static bool Equals(Object obj) { ... }
```

4

```
public static int operator implicit(Time t1)
{ ...}
```

5

Создание и использование делегатов

- Сценарий: Атомная электростанция
- Анализ проблемы
- Создание делегатов
- Использование делегатов

Сценарий: Атомная электростанция

- **Проблема**
  - Как реагировать на изменения температуры на атомной электростанции
  - Если температура активной зоны реактора станет выше определенной температуры, необходимо включить охлаждающие насосы
- **Возможные решения**
  - Все охлаждающие насосы должны постоянно отслеживать температуру активной зоны реактора
  - При критическом изменении температуры специальный компонент, отслеживающий температуру активной зоны, должен включить необходимые насосы

Анализ проблемы

- **Имеющиеся трудности**
  - Имеются различные типы насосов, произведенные различными заводами
  - У каждого насоса свой метод для его активации
- **Возможные трудности в будущем**
  - При добавлении нового насоса, необходимо переписать весь код
  - При каждом таком добавлении существенные накладные расходы
- **Решение**
  - Используйте в своем коде делегаты

Пример

```
public class ElectricPumpDriver{
    public void startElectricPumpRunning(){
        Console.WriteLine("The electric pump is started!");
    }
}
```

```
public class PneumaticPumpdriver{
    public void SwitchOn(){
        Console.WriteLine("The pneumatic pump is started!");
    }
}
```

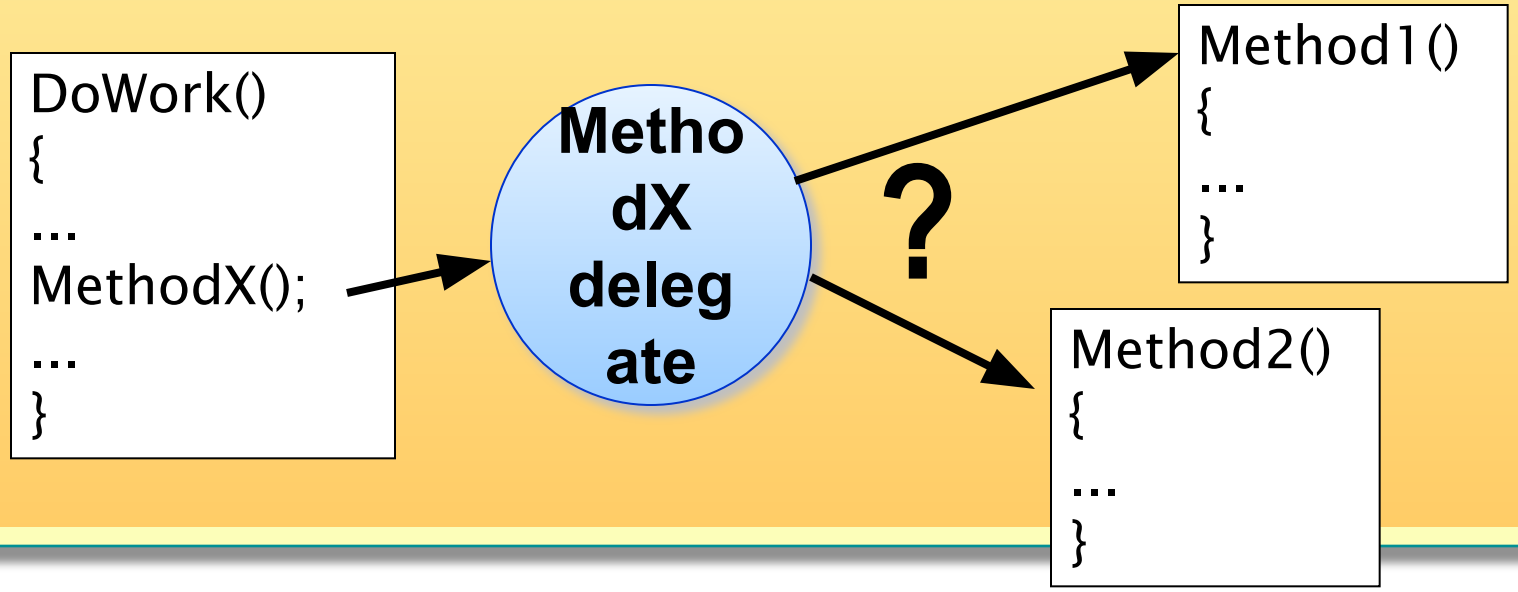
```
public class ExampleOfUse{
    static void Main(string[] args){
        CoreTempMonitor ctm=new CoreTempMonitor();
        ElectricPumpDriver ed1=new ElectricPumpDriver();
        ctm.Add(ed1);
        PneumaticPumpdriver pd1=new PneumaticPumpdriver();
        ctm.Add(pd1);
        ctm.SwitchOnAllPumps(); }
}
```

```
public class CoreTempMonitor{
    private ArrayList pumps = new ArrayList();

    public void Add(object pump){
        pumps.Add(pump);
    }
    public void SwitchOnAllPumps(){
        foreach (object pump in pumps){
                if (pump is ElectricPumpDriver){
                        ((ElectricPumpDriver)pump).StartElectricPumpRunning();
                }
                if (pump is PneumaticPumpdriver){
                        ((PneumaticPumpdriver)pump).SwitchOn();
                }
        }
    }
}
```

Применение делегатов

- **Делегат позволяет вызывать метод косвенно**
  - Содержит ссылку на метод
  - Делегат может вызывать только такие методы, у которых тип возвращаемых значений и список параметров совпадают с соответствующими элементами объявления делегата



Создание делегатов

- **Синтаксис объявления делегата**

```
[<спецификатор доступа>] delegate <тип результата > <имя класса>
(<список аргументов>);
```

- **Размещать объявление делегата :**
  - непосредственно в пространстве имен,
  - внутри другого класса,
    - ✔ такое объявление рассматривается как объявление вложенного класса.

```
public delegate void Del(string message);
```

```
public delegate int PerformCalculation(int x, int y);
```

Свойства делегатов

- Делегаты похожи на указатели функций в C++, но являются типобезопасными.
- Делегаты допускают передачу методов в качестве параметров.
- Делегаты можно использовать для задания функций обратного вызова.
- Делегаты можно связывать друг с другом; например, несколько методов можно вызвать по одному событию.
- Точное соответствие прототипа методов прототипу делегата не требуется (вариативность в делегатах - позволяет методу иметь тип возвращаемого значения, степень наследования которого больше, чем указано в делегате, либо позволяет использовать метод с типами параметров, степень наследования которых меньше, чем у типа делегата).

Где следует размещать объявление делегата?

- **Размещают делегаты:**
  - непосредственно в пространстве имен, наряду с объявлениями других классов, структур, интерфейсов;
  - внутри другого класса, наряду с объявлениями методов и свойств.

Использование делегатов

- **Делегаты вызываются также как и методы**

```
public delegate void StartPumpCallback( );
...
StartPumpCallback callback;
...
callback = new
    StartPumpCallback(ed1.StartElectricPumpRunning);
...
callback( );
```

Нет тела метода

Нет вызова

Вызов

Пример использования делегата

```
public delegate void StartPumpCallback();
```

```
public class ElectricPumpDriver{
    public void startElectricPumpRunning(){
            Console.WriteLine("The electric pump is started!");
    }
}
```

```
public class PneumaticPumpdriver{
    public void SwitchOn(){
            Console.WriteLine("The pneumatic pump is started!");
    }
}
```

```
public class CoreTempMonitor{
    private ArrayList  callbacks = new ArrayList();

    public void Add(StartPumpCallback callback){
        callbacks.Add(callback);
    }
    public void SwitchOnAllPumps(){
        foreach (StartPumpCallback callback in callbacks){
                callback();
        }
    }
```

```
public class ExampleOfUse{
    static void Main(string[] args){
        CoreTempMonitor ctm=new CoreTempMonitor();
        PneumaticPumpdriver pd1=new PneumaticPumpdriver();
        StartPumpCallback  a = new StartPumpCallback(pd1.SwitchOn);
        ctm.Add(a);

        ElectricPumpDriver ed1 = new ElectricPumpDriver();
        ctm.Add(new StartPumpCallback(ed1.StartElectricPumpRunning));
        ctm.SwitchOnAllPumps(); }
}
```

Делегат с именованным методом

- **При создании экземпляра делегата с помощью именованного метода этот метод передается в качестве параметра**

```
// Declare a delegate:
delegate void Del(int x);
// Define a named method:
void DoWork(int k) { /* ... */ }
// Instantiate the delegate using
//the method as a parameter:
Del d = obj.DoWork;
```

Пример делегата с именованным методом

```
public delegate void Del(string message);
```

```
class Print {
        // Create a method for a delegate.
        public static void DelegateMethod(string message){
                System.Console.WriteLine("static"+message);
        }

        public void DelegateMethod2(string message) {
                System.Console.WriteLine("dinamic"+message);
        }
}
```

```
class Program {
      static void Main(string[] args)
      {
            // Instantiate the delegate.
            Del handler = Print.DelegateMethod;
            // Call the delegate.
            handler("Hello");

            handler = (new Print()).DelegateMethod2;
            handler("World");
      }
}
```

Делегат с анонимным методом

- **Создание анонимных методов является, по существу, способом передачи блока кода в качестве параметра делегата**

```
// Create a delegate instance
delegate void Del(int x);
// Instantiate the delegate using an anonymous method
Del d = delegate(int k) { /* ... */ };
```

- **Использование анонимных методов позволяет сократить издержки на кодирование при создании делегатов, поскольку не требуется создавать отдельный метод**

Пример делегата с анонимным методом

```
class TestClass {
        delegate string Printer(string s, int i);

        static void Main(string[] args){
                        Printer p = delegate(string j, int i){
                                System.Console.WriteLine(j + " №" + i);
                                return "hh";
                        };

                        // Results from the anonymous delegate call:
                        p("The delegate using the anonymous method is called.", 1);

                        // The delegate instantiation using a named method "DoWork":
                        p = new Printer(TestClass.DoWork);
                        // Results from the old style delegate call:
                        p("The delegate using the named method is called.", 2);
        }

        // The method associated with the named delegate:
        static string DoWork(string k, int i){
                        System.Console.WriteLine(k + " №" + i);
                        return "ff";
        }
}
```

Определение и использование событий

- Как работают события
- Определения событий
- Передача параметров в события
- Демонстрация: Обработка событий

Как работают события

- **Издатель**
  - Генерирует событие, оповещающее все заинтересованные объекты (подписчики)
- **Подписчик**
  - Предоставляет метод, вызываемый при генерации события
- **Форма объявления события:**

```
event событийный_делегат объект;
```

Определения событий

- **Определение события**

```
public delegate void StartPumpCallback( );
private event StartPumpCallback CoreOverheating;
```

- **Подписка на событие**

```
PneumaticPumpDriver pd1 = new PneumaticPumpDriver( );
...
CoreOverheating += new StartPumpCallback(pd1.SwitchOn);
```

- **Уведомление подписчиков о событии**

```
public void SwitchOnAllPumps( ) {
  if (CoreOverheating != null) {
     CoreOverheating( );
  }
}
```

Пример определения события

```
public delegate void StartPumpCallback();
```

```
public class ElectricPumpDriver{
    public void startElectricPumpRunning(){
            Console.WriteLine("The electric pump is started!");
    }
}
```

```
public class PneumaticPumpdriver{
    public void SwitchOn(){
            Console.WriteLine("The pneumatic pump is started!");
    }
}
```

```
public class CoreTempMonitor{
    public event StartPumpCallback     CoreOverheating;

    public void AddOnCoreOverHeating
                (StartPumpCallback startpump){
        CoreOverheating+=startpump;
    }
    public void SwitchOnAllPumps(){
        if (CoreOverheating!=null)
                {CoreOverheating();}
} }
```

```
public class ExampleOfUse{
    static void Main(string[] args){
        CoreTempMonitor ctm=new CoreTempMonitor();

        PneumaticPumpdriver pd1=new PneumaticPumpdriver();
        ctm.CoreOverheating+=new StartPumpCallback(pd1.SwitchOn);
        ctm.AddOnCoreOverHeating(new StartPumpCallback(pd1.SwitchOn));

        ElectricPumpDriver ed1 = new ElectricPumpDriver();
        ctm.CoreOverheating+=new StartPumpCallback(ed1.StartElectricPumpRunning);
        ctm.AddOnCoreOverHeating(new StartPumpCallback(ed1.StartElectricPumpRunning));

        ctm.SwitchOnAllPumps(); }
}
```

Передача параметров в события

- **Параметры в события должны передаваться как EventArgs**
  - Создайте класс, унаследованный от EventArgs, который будет служить контейнером для параметров события
- **Один и тот же метод-подписчик может вызываться несколькими событиями**
  - Первым параметром, передаваемым в метод, всегда должен быть издатель события (sender)

Пример передачи параметров в события

```
public delegate void StartPumpCallback(object sender, CoreOverHeatingEventArgs args);
```

```
public class ElectricPumpDriver{
    public void startElectricPumpRunning (object  sender, CoreOverHeatingEventArgs  args){
        int currentTemperature=args.GetTemperature();
        if (currentTemperature>900){ Console.WriteLine("The electric pump is started at T " + currentTemperature);  }
        else{  Console.WriteLine("The electric pump is not started!")}}}
```

```
public class PneumaticPumpdriver{
    public void SwitchOn(object sender, CoreOverHeatingEventArgs args){
        int currentTemperature=args.GetTemperature();
        if (currentTemperature>1200){ Console.WriteLine(""The pneumatic pump is started at T " +currentTemperature);  }
        else{  Console.WriteLine("The pneumatic pump is not started!")}}}
```

```
public class CoreOverHeatingEventArgs:EventArgs {
        private readonly int temperature;
        public CoreOverHeatingEventArgs(int temperature)          {this.temperature=temperature;}
        public int GetTemperature()                               {return temperature;}}
```

```
public class CoreTempMonitor{
     public event StartPumpCallback     CoreOverheating;
     public void AddOnCoreOverHeating
                (StartPumpCallback startpump){
        CoreOverheating+=startpump;
     }
    public void SwitchOnAllPumps(int   i){
       if (CoreOverheating!=null){
        CoreOverheating(
                this,new CoreOverHeatingEventArgs(t));}
} }
```

```
public class ExampleOfUse{
     static void Main(string[] args){
        CoreTempMonitor ctm=new CoreTempMonitor();
        ElectricPumpDriver ed1=new ElectricPumpDriver();
        ctm.AddOnCoreOverHeating(
                new StartPumpCallback(ed1.StartElectricPumpRunning));
        PneumaticPumpdriver pd1=new PneumaticPumpdriver();
        ctm.AddOnCoreOverHeating(
                new StartPumpCallback(pd1.SwitchOn));
        ctm.SwitchOnAllPumps(1300); //1100 or 1300   }}
```

Атрибуты

Понятие атрибутов

- **Атрибуты - это:**
  - Описательные тэги в программном коде, передающие информацию во время выполнения программы
  - Хранятся вместе с метаданными элемента
- **.NET Framework содержит множество встроенных атрибутов**
  - Среда выполнения содержит код, проверяющий значения атрибутов и меняет свое поведение в соответствии с этими значениями

Применение атрибутов

- **Синтаксис: Для использования атрибута необходимо указать его имя в квадратных скобках**

```
[attribute(positional_parameters,named_parameter=value, ...)]
element
```

- **Можно указать несколько атрибутов для одного элемента:**
  - Заключить каждый из атрибутов в отдельные квадратные скобки
  - Использовать одни квадратные скобки и перечислить атрибуты через запятую
  - В некоторых случаях необходимо явно указать имя элемента, которому принадлежит атрибут

Использование стандартных атрибутов

- **В .NET определено большое количество стандартных атрибутов**
  - Пример: Использование атрибута Conditional

Использование атрибута Conditional

- **Используется как инструмент отладки**
  - Производит условную компиляцию вызовов метода в зависимости от значения параметра, определяемого программным путем
  - Не производит условную компиляцию самих методов
- **Ограничения на методы:**
  - Должны возвращать тип **void**
  - Не должны быть объявлены как override
  - Не должны быть методами наследуемыми от интерфейса

```
using System.Diagnostics;
...
class MyClass
{
  [Conditional ("DEBUGGING")]
  public static void MyMethod( )
  {
    ...
  }
}
```

Создание пользовательских атрибутов

- Определение области действия пользовательского атрибута
- Создание класса атрибута
- Обработка пользовательского атрибута
- Использование нескольких атрибутов

Определение области действия пользовательского атрибута

- Для определения области действия используйте тэг атрибута AttributeUsage

```
[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute: System.Attribute
{ ... }
```

- Для определения нескольких элементов необходимо использовать оператор «|»

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttribute: System.Attribute
{ ... }
```

- Спецификация используемости атрибута

```
[AttributeUsage(доступные_элементы, AllowMultiple=true_или_false,
  Inherited=наследуемость )]
```

Создание класса атрибута

- **Наследование класса атрибута**
  - Все классы атрибутов должны наследоваться от System.Attribute
  - Добавляйте к имени класса атрибута слово “Attribute”
- **Компоненты класса атрибута**
  - Для каждого класса атрибута определите один конструктор, устанавливающий обязательную информацию
  - Создайте свойства для передачи дополнительных именованных параметров.

Обработка пользовательского атрибута

Процесс компиляции

1. **Поиск класса атрибута**
2. **Проверка области действия атрибута**
3. **Проверка конструктора атрибута**
4. **Создание экземпляра объекта**
5. **Проверка именованных параметров**
6. **Установка для поля или свойства значения именованного параметра**
7. **Сохраняется текущее состояние класса атрибута**

Свойства и индексаторы

Зачем использовать свойства?

- **Преимущества использования свойств:**
  - Удобный способ инкапсуляции информации внутри класса
  - Прозрачный синтаксис

  Пример: o.SetValue(o.GetValue()+1);

  o.Value++;

Использование аксессоров

- **Можно работать со свойством как с открытой переменной-членом класса**
  - **get** –аксессор предназначен для чтения свойств
  - **set** – аксессор предназначен для установки свойства

```
class Button
{
    public string Caption // Property
    {
        get { return caption; }
        set { caption = value; }
    }
    private string caption; // Field
}
```

Типы свойств

- **Для чтения и записи**
  - Реализуют **get** - и **set** -аксессоры
- **Только для чтения**
  - Реализован только **get** –аксессор
  - Не константы
- **Только для записи**
  - Реализован только **set** –аксессор
- **Статические свойства**
  - Для обращения к статическим данным, хранящим информацию на уровне всего класса

Сравнение свойств с полями

- **Свойства – это «умные поля»**
  - **get** –аксессор может возвращать расчетное значение
- **Сходства**
  - Одинаковый синтаксис создания и использования
- **Различия**
  - Свойства не определяют область памяти
  - Свойства нельзя передавать в методы как **ref** или **out**

Сравнение свойств с методами

- **Сходства**
  - И те, и другие содержат исполняемый код
  - И те, и другие можно использовать для инкапсуляции данных
  - И те, и другие могут быть virtual, abstract или override
- **Различия**
  - Синтаксические – для работы со свойствами не используются круглые скобки
  - Семантические – свойства не могут быть **void** или принимать параметры

Что такое индексатор?

- **Индексатор позволяет получать доступ к объекту по индексу подобно тому, как это реализовано в массивах**
  - Удобно, если свойство может принимать различные значения
- **Создание индексатора**
  - Создайте свойство с именем *this*
  - Определите тип индекса
- **Использование индексатора**
  - Для чтения или записи проиндексированного свойства используйте синтаксис для массивов

Создание одномерных индексаторов

```
тип_элемента this[int индекс]
{
// Аксессор считывания данных,
get {
// Возврат значения, заданного  элементом индекс

// Аксессор установки данных,
set {
// Установка значения, заданного элементом индекс
     }

}
```

- ***тип_элемента* — базовый тип индексатора**
- **Параметр *индекс* получает индекс опрашиваемого (или устанавливаемого) элемента**

Использование параметров при определении индексаторов

- **При создании индексатора**
  - Необходимо определить хотя бы один индекс
  - Укажите значение для каждого из параметров
  - Не используйте модификаторы **ref** или **out**

```
class MultipleParameters
{
public string this[int one, int two]
{
get { ... }
set { ... }
}
...
}
...
MultipleParameters mp = new MultipleParameters( );
string s = mp[2,3];
...
```

Сравнение индексаторов с массивами

- **Сходства**
  - И те, и другие используют синтаксис для массивов
- **Различия**
  - Индексаторы могут использовать индексы различных типов
  - Индексаторы можно перегружать – можно создать несколько индексаторов с индексами различного типа
  - Индексаторы – это не переменные, они не определяют область памяти. Их нельзя передавать в качестве параметров, используя **ref** или **out**

Сравнение индексаторов со свойствами

- **Сходства**
  - И те, и другие используют **get**- и **set** -аксессоры
  - Ни те, ни другие не определяют область памяти
  - Ни те, ни другие не могут быть void
- **Различия**
  - Индексаторы можно перегружать
  - Индексаторы не могут быть статическими

Пространства имен

Пространство имен

- **Пространство имен определяет декларативную область, которая позволяет отдельно хранить множества имен.**
  - платформа .NET Framework использует пространства имен для организации большинства классов.

    ✔Пример: библиотека .NET Framework использует пространство имен **System**.

```
System.Console.WriteLine("Hello World!");
```

  - объявление собственного пространства имен поможет в управлении областью действия имен классов и методов в крупных программных проектах

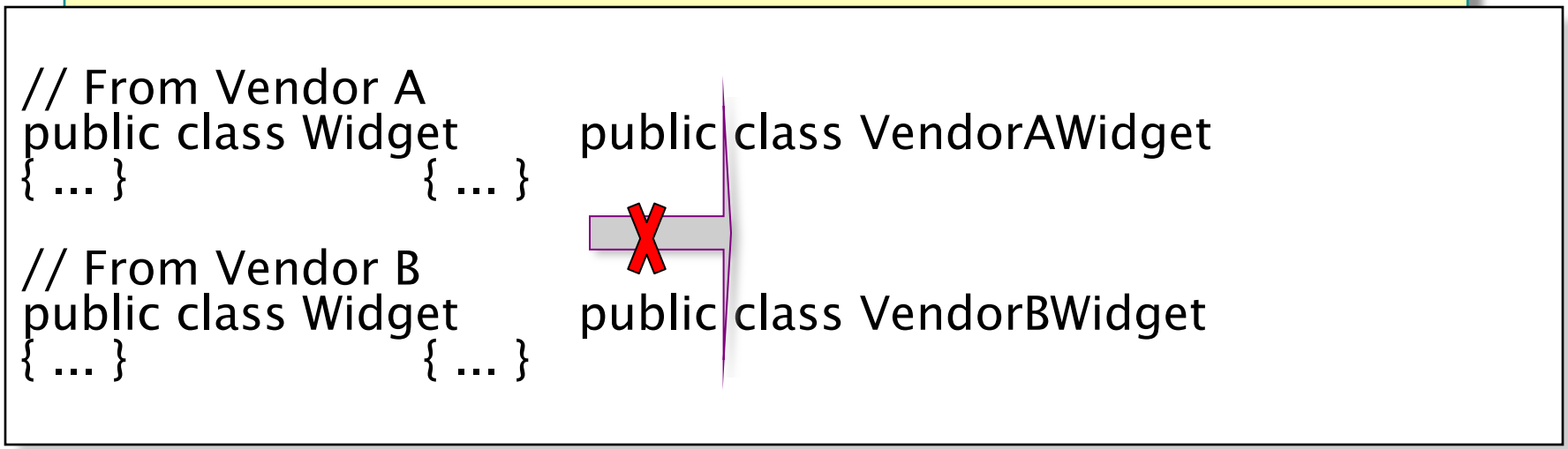
Область видимости

- **Область видимости имени определяется той частью программы, в которой вы можете использовать имя без уточнения**

```
public class Bank
{
    public class Account
    {
        public void Deposit(decimal amount)
        {
            balance += amount;
        }
        private decimal balance;
    }
    public Account OpenAccount( ) { ... }
}
```

Разрешение конфликтов имен

- Реальные проекты могут насчитывать в своем составе тысячи классов
- Что будет, если имена каких-то двух классов совпадут?
- Не добавляйте приставку ко всем именам классов



Объявление пространств имен

```
namespace VendorA
{
    public class Widget
    { ... }
}
```

```
namespace VendorB
{
    public class Widget
    { ... }
}
```

```
namespace Microsoft
{
   namespace Office
   {
      ...
   }
}
```

Краткая запись

```
namespace Microsoft.Office
{

}
```

Полностью определенные имена

- Полностью определенное имя класса включает в себя пространство имен
- Неопределенные имена классов могут использоваться только в пределах их области видимости

```
namespace VendorA
{
    public class Widget { ... }
    ...
}
class Application
{
    static void Main( )
    {
        Widget w = new Widget( );
        VendorA.Widget w = new VendorA.Widget( );
    }
}
```

Объявление директивы using

- **Позволяет вернуть имена в область их видимости**

```
namespace VendorA.SuiteB
{
    public class Widget { ... }
}
```

```
using VendorA.SuiteB;

class Application
{
    static void Main( )
    {
        Widget w = new Widget( );
    }
}
```

Использование альтернативных имен в директиве using

- Можно создавать альтернативные имена (псевдонимы) для вложенных пространств имен и типов

```
namespace VendorA.SuiteB
{
    public class Widget { ... }
}
```

```
using Widget = VendorA.SuiteB.Widget;

class Application
{
    static void Main( )
    {
        Widget w = new Widget( );
    }
}
```

Рекомендации по именованию пространств имен

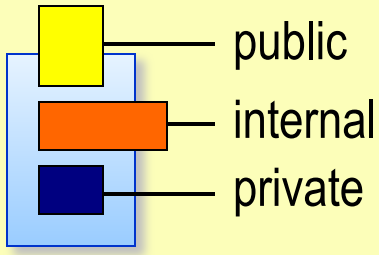
- **Для логического разделения имени используйте технику «Паскаль»**
  - Пример: VendorA.SuiteB
- **Рекомендуется для пространств имен использовать префикс с именем компании**
  - Пример: Microsoft.Office
- **При необходимости используйте имена во множественном числе**
  - Example: System.Collections
- **Избегайте конфликтов имен между пространствами имен и классами**

Использование внутренних (internal) классов, методов и данных

- Зачем нужен модификатор доступа internal?
- Модификатор доступа internal
- Синтаксис
- Пример использования модификатора доступа internal

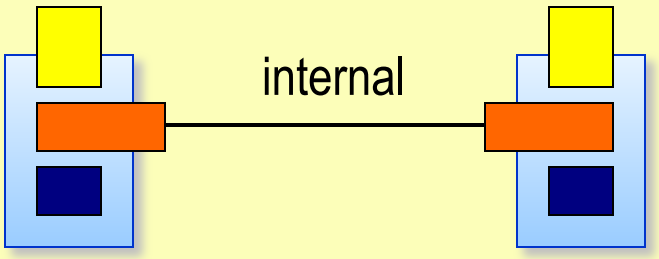
Зачем нужен модификатор доступа internal?

- Маленькие объекты редко используются сами по себе
- Объекты должны взаимодействовать между собой, образуя более крупные объекты
- Необходимо организовать доступ между отдельными объектами



Модификатор доступа internal

- **Сравнение модификаторов доступа**
  - Модификатор доступа public логический
  - Модификатор доступа private логическийl
  - Модификатор доступа internal физический



Синтаксис

```
internal class <outername>
{
    internal class <nestedname> { ... }
    internal <type> field;
    internal <type> Method( ) { ... }

    protected internal class <nestedname> { ... }
    protected internal <type> field;
    protected internal <type> Method( ) { ... }
}
```

protected internal означает **protected** *или* **internal**

Пример использования модификатора доступа internal

```
public interface IBankAccount { ... }

internal abstract class CommonBankAccount { ... }

internal class DepositAccount: CommonBankAccount,
                               IBankAccount { ... }
public class Bank
{
    public IBankAccount OpenAccount( )
    {
        return new DepositAccount( );
    }
}
```