Лабораторная работа №3

ХАРЬКОВСКИЙ НАЦИОНАЛЬНЫЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И МЕХАТРОНИКИ

Кафедра информационных технологий и мехатроники

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по проведению лабораторных работ по дисциплине
«Объектно-ориентированное программирование»
для студентов специальности 6.050101 “Компьютерные науки”

Разработчик - доцент кафедры информационных технологий и мехатроники
кандидат технических наук, старший научный сотрудник
Тимонин Владимир Алексеевич

Харків 2015

Лабораторная работа №3
Исследование возможностей интегрированной среды разработки Visual C# для создания приложений с классами. Свойства класса.

Цель работы – исследовать возможности интегрированной среды разработки Visual Studio 2010 и получить практические навыки по созданию приложений, использующие свойства класса.

1. Теоретические сведения

Свойство — это специальный тип членов класса, которое включает поле и методы доступа к этому полю. Часто требуется создать поле, которое должно быть доступно для пользователей объекта, но при этом необходимо осуществлять управление операциями, разрешенными к выполнению над этим полем. Например, необходимо ограничить диапазон значений, которые можно присваивать этому полю. Несмотря на то, что этого можно достичь с помощью закрытой переменной и методов доступа к ней, свойство предлагает более удобный и простой способ решения этой задачи.

Концепция инкапсуляции вращается вокруг принципа, гласящего, что внутренние данные объекта не должны быть напрямую доступны через экземпляр объекта. Вместо этого, если вызывающий код желает изменить состояние объекта, то должен делать это через методы доступа (accessor, или метод get) и изменения (mutator, или метод set). В C# инкапсуляция предоставляет способ предохранения целостности данных о состоянии объекта и обеспечивается на синтаксическом уровне с использованием ключевых слов public, private, internal и protected. Вместо определения общедоступных полей (которые легко приводят к повреждению данных), необходимо определять приватные данные, управление которыми осуществляется опосредованно, с применением следующих способов:

- определение пары методов доступа и изменения;
- определение свойства .NET.

1.1. Способы инкапсуляции

1.1.1. Инкапсуляция с использованием традиционных методов доступа и изменения

Пример 1. Рассмотрим инкапсуляцию с использованием традиционных методов доступа и изменения на примере класса, моделирующего обычного сотрудника:

```
class Employee
    {
    // поля данных
    private string empName; // имя
    private int empID;          // код
    private float currPay;      // денежный взнос
    // конструкторы
    public Employee () { }
    public Employee(string name, int id, float pay)
        {
        empName = name;
        empID = id;
        currPay = pay;
        }
    // методы
    public void GiveBonus(float amount)
```

```
        {
         currPay += amount; // количество
        }
      public void DisplayStats ()
        {
         Console.WriteLine("Name: {0}", empName);
         Console.WriteLine ("ID: {0}", empID);
         Console.WriteLine("Pay: {0}", currPay);
        }
    }
```

Так как поля класса Employee определены с ключевым словом private, то поля empName, empID и currPay напрямую через объектную переменную не доступны:

```
    static void Main(string[] args)
      {
       // Ошибка! Невозможно напрямую обращаться к приватным полям объекта!
       Employee emp = new Employee ();
       emp.empName = "Ivanov";
      }
```

Если необходимо взаимодействовать с полным именем сотрудника, то понадобится определить методы доступа (метод get) и изменения (метод set). Роль метода get состоит в возврате вызывающему коду значения лежащих в основе статических данных. Метод set позволяет вызывающему коду изменять текущее значение лежащих в основе статических данных.

Для целей иллюстрации инкапсулируем поле empName. Для этого к существующему классу Employee следует добавить показанные ниже общедоступные члены. Обратите внимание, что метод SetName() выполняет проверку входящих данных, чтобы удостовериться, что строка имеет длину не более 15 символов. Если это не так, на консоль выводится сообщение об ошибке и происходит возврат без изменения значения поля empName.

```
    class Employee
      {
       private string empName;
       public string GetName() // метод доступа (метод get)
        {
          return empName;
        }
       public void SetName(string name) // метод изменения (метод set)
        {
          // Перед присваиванием проверить входное значение
          if (name.Length > 15)
      Console. WriteLine ("Ошибка! Имя должно быть меньше 16 символов!");
          else
            empName = name;
        }
      }
```

Этот способ требует наличия двух уникально именованных методов для управления единственным элементом данных. Для иллюстрации модифицируем метод Main() следующим образом:

```
    static void Main(string [ ] args)
      {
       Employee emp = new Employee("Ivanovich", 456, 30000);
       emp.GiveBonus(1000);
       emp.DisplayStats();
       // использовать методы get/set для взаимодействия с именем объекта
```

```
        emp.SetName("Ivanov");
        Console.WriteLine("Имя сотрудника: {0}", emp.GetName ());
        Console.ReadLine();
      }
```

Благодаря коду в методе SetName (), попытка присвоить строку длиннее 15 символов приводит к выводу на консоль жестко закодированного сообщения об ошибке:

```
    static void Main(string[] args)
      {
        Employee emp2 = new Employee ();
        emp2.SetName("Solodov-Petrovski");
        // длиннее 15 символов! На консоль выводится сообщение об ошибке
        Console.ReadLine();
      }
```

Приватное поле empName инкапсулировано с использованием двух методов GetName() и SetName(). Для дальнейшей инкапсуляции данных в классе Employee понадобится добавить ряд дополнительных методов (например, GetID(), SetID(), GetCurrentPay(), SetCurrentPay()). Каждый метод, изменяющий данные, может иметь в себе несколько строк кода для проверки дополнительных правил.

1.1.2. Инкапсуляция с использованием свойств .NET

К возможности инкапсуляции полей данных с использованием традиционной пары методов get/set, в языках .NET имеется более предпочтительный способ инкапсуляции данных с помощью свойств. Так как свойства — это всего лишь упрощенное представление "реальных" методов доступа и изменения, то разработчик класса может реализовать любую внутреннюю логику, которую нужно выполнить перед присваиванием значения (например, преобразовать в верхний регистр, очистить от недопустимых символов, проверить границы числовых значений и т.д.).

Пример 2. Ниже приведен измененный класс Employee, который обеспечивает инкапсуляцию каждого поля с применением синтаксиса свойств вместо традиционных методов get/set.

```
    class Employee
      {
        // поля данных
        private string empName;
        private int empID;
        private float currPay;
        // свойства
        public string Name // имя
              {
                get { return empName; }
                set
                   {
                     if (value.Length > 15)
          Console. WriteLine ("Ошибка! Имя должно быть меньше 16 символов!");
                       else
                           empName = value;
                   }
              }
        public int ID // код
              {
                get { return empID; }
                set { empID = value; }
              }
```

```
        public float Pay // взнос
              {
               get { return currPay; }
               set { currPay = value; }
              }
    }
```

Свойство C# состоит из определений контекстов чтения get (метод доступа) и set (метод изменения), вложенных непосредственно в контекст самого свойства. Свойство указывает тип данных, которые оно инкапсулирует, как тип возвращаемого значения. Кроме того, в отличие от метода, в определении свойства не используются скобки (даже пустые).

В контексте set свойства используется лексема value, которая представляет входное значение, присваиваемое свойству вызывающим кодом. Эта лексема не является настоящим ключевым словом C#, а представляет собой то, что называется контекстуальным ключевым словом. Когда лексема value находится внутри контекста set, она всегда обозначает значение, присваиваемое вызывающим кодом, и всегда имеет тип, совпадающий с типом самого свойства. Поэтому свойство Name может проверить допустимую длину string следующим образом:

```
public string Name
              {
                get { return empName; }
                set
                   {
                    if (value.Length > 15)
          Console. WriteLine ("Ошибка! Имя должно быть меньше 16 символов!");
                     else
                         empName = value;
                   }
              }
```

При наличии этих свойств вызывающему коду кажется, что он имеет дело с общедоступным элементом данных, однако "за кулисами" при каждом обращении вызывается корректный get или set, обеспечивая инкапсуляцию:

```
static void Main(string [ ] args)
    {
     Employee emp = new Employee("Ivanovich", 456, 30000);
     // Установка свойств Name, ID, Pay
     emp.Name = "Ivanov";
     emp.ID = 3015;
      emp.Pay = (float)396.82;
      // Отображение значений свойств
      Console.WriteLine("Сотрудник: {0} код: {1} взнос: {2}", emp.Name, emp.ID, emp.Pay);
     Console.ReadLine();
    }
```

Свойства (в противоположность методам доступа и изменения) также облегчают манипулирование типами, поскольку способны реагировать на внутренние операции C#. Для иллюстрации представим, что тип класса Employee имеет внутреннюю приватную переменную-член, хранящую возраст сотрудника. Ниже показаны необходимые изменения (обратите внимание на использование цепочки вызовов конструкторов):

```
class Employee
    {
     // Новое поле и свойство
     private int empAge; // возраст
     public int Age
              {
```

```
            get { return empAge; }
            set { empAge = value; }
          }
        // Обновленные конструкторы
        public Employee () {}
        public Employee(string name, int id, float pay)
            :this(name, 0, id, pay){}
        public Employee(string name, int age, int id, float pay)
          {
            empName = name;
            empID = id;
            empAge = age;
            currPay = pay;
          }
        // Обновленный метод DisplayStats()
        public void DisplayStats()
          {
            Console.WriteLine("Имя: {0}", empName);
            Console.WriteLine ("Код: {0}", empID);
            Console.WriteLine ("Возраст: {0}", empAge);
            Console.WriteLine("Взнос: {0}", currPay);
          }
    }
```

Теперь предположим, что создан объект Employee по имени Petrov. Необходимо, чтобы в день рождения сотрудника возраст увеличивался на 1 год. Используя традиционные методы set /get, пришлось бы написать код вроде следующего:

```
Employee joe = new Employee() ;
Petrov.SetAge(Petrov.GetAge() + 1);
```

Однако если empAge инкапсулируется через свойство по имени Age, можно записать проще:

```
Employee Petrov = new Employee() ;
Petrov.Age++;
```

1.2. Использование свойств внутри определения класса

Свойства, а в особенности их часть set — это общепринятое место для размещения правил класса. В настоящее время класс Employee имеет свойство Name, которое гарантирует длину имени не более 15 символов. Остальные свойства (ID, Pay и Age) также могут быть обновлены с добавлением соответствующей логики.

Необходимо обратить внимание на то, что обычно происходит внутри конструктора класса. Конструктор получает входные параметры, проверяет корректность данных и затем выполняет присваивания внутренним приватным полям. В настоящее время ведущий конструктор не проверяет входные строковые данные на допустимый диапазон, поэтому можно было бы изменить его следующим образом:

```
public Employee (string name, int age, int id, float pay)
    {
     if (value.Length > 15)
       Console. WriteLine ("Ошибка! Имя должно быть меньше 16 символов!");
      else
          empName = value; // это может оказаться проблемой!
     empID = id;
     empAge = age;
     currPay = pay;
    }
```

Свойство Name и ведущий конструктор предпринимают одну и ту же проверку ошибок! В результате получается дублирование кода Чтобы упростить код и разместить всю проверку ошибок в центральном месте, для установки и получения данных внутри класса разумно всегда использовать свойства. Ниже показан соответствующим образом обновленный конструктор:

```
public Employee (string name, int age, int id, float pay)
      {
       Name = name;
       Age = age;
       ID = id;
       Pay = pay;
      }
```

Помимо обновления конструкторов с целью использования свойств для присваивания значений, имеет смысл сделать это повсюду в реализации класса, чтобы гарантировать неукоснительное соблюдение соответствующих правил. Во многих случаях единственное место, где можно напрямую обращаться к приватным данным — это внутри самого свойства. С учетом сказанного модифицируем класс Employee, как показано ниже:

```
class Employee
    {
     // Поля данных
     private string empName;
     private int empID;
     private float currPay;
     private int empAge;
     // Конструкторы
     public Employee () { }
     public Employee(string name, int id, float pay)
       :this(name, 0, id, pay) { }
     public Employee(string name, int age, int id, float pay)
          {
           Name = name;
           Age = age;
           ID = id;
           Pay = pay;
          }
     // Методы
     public void GiveBonus(float amount)
          { Pay += amount; }
     public void DisplayStats ()
          {
           Console.WriteLine("Имя: {0}", Name);
           Console.WriteLine("Код: {0}", ID);
           Console.WriteLine("Возраст: {0}", Age);
           Console.WriteLine("Взнос: {0}", Pay);
          }
     // Свойства остаются прежними...
```

Преимущество свойств состоит в том, что пользователи объектов могут манипулировать внутренними данными, применяя единый именованный элемент.

1.3. Свойства, доступные только для чтения и только для записи

При инкапсуляции данных может понадобиться сконфигурировать свойство, доступное только для чтения. Для этого нужно просто опустить блок set. Аналогично, если требуется создать свойство,

доступное только для записи, следует опустить блок get. Например, нижеприведенный код делает свойство SocialSecurityNumber доступным только для чтения:

```
public string SocialSecurityNumber
      {
        get { return empSSN; }
      }
```

После этого единственным способом модификации номера карточки социального страхования будет передача его в аргументе конструктора Теперь попытка установить новое значение SSN для сотрудника внутри ведущего конструктора приведет к ошибке компиляции:

```
public Employee(string name, int age, int id, float pay, string ssn)
      {
        Name = name;
        Age = age;
        ID = id;
        Pay = pay;
        SocialSecurityNumber = ssn; // Невозможно, т.к. свойство предназначено
                                    // только  для чтения!
}
```

Если сделать это свойство доступным только для чтения, в логике конструктора останется лишь пользоваться лежащей в основе переменной-членом ssn.

1.4. Статические свойства

В C# также поддерживаются статические свойства (статические члены доступны на уровне класса, а не на уровне экземпляра (объекта) этого класса). Например, предположим, что в классе Employee определен статический элемент данных для представления названия организации, нанимающей сотрудников.

Инкапсулировать статическое свойство можно следующим образом:

```
class Employee
    {
      private static string companyName;
      // Статические свойства должны оперировать статическими данными!
      public static string Company
            {
              get { return companyName; }
              set { companyName = value; }
            }
    }
```

Манипулировать статическими свойствами можно точно так же, как статическими методами:

```
static void Main(string[] args)
    {
      Employee.Company = "Авто555"; // взаимодействие со статическим свойством.
      Console.WriteLine("Сотрудники компании: {0}", Employee.Company);
      Employee emp = new Employee("Ivanov", 24, 456, 30000, "11-11-1111");
      emp.GiveBonus(1000);
      emp.DisplayStats() ;
      Console.ReadLine();
    }
```

Так как классы могут поддерживать статические конструкторы, то если нужно гарантировать, что имя в статическом поле companyName всегда будет установлено в Авто555, понадобится написать следующий код:

```
public class Employee
      {
```

```
    private Static companyName As string
    . . .
    static Employee ()
        {
         companyName = "Авто555";
        }
  }
```

Используя такой подход нет необходимости явно вызывать свойство Company для того, чтобы установить начальное значение:

```
// Автоматическая установка значения "Авто555" через статический конструктор.
static void Main(string [ ] args)
  {
   Console.WriteLine("Сотрудники компании: {0}", Employee.Company);
  }
```

1.5. Понятие автоматических свойств

С выходом платформы .NET 3.5 в языке C# появился еще один способ определения простых служб инкапсуляции с минимальным кодом, а именно — синтаксис автоматических свойств. Для целей иллюстрации создадим новый класс Car, в котором инкапсулирована единственная порция данных с использованием классического синтаксиса свойств.

```
class Car
  {
   private string carName = string.Empty;
   public string PetName
        {
         get { return carName; }
         set { carName = value; }
        }
  }
```

Хотя большинство свойств C# содержат в своем контексте определенные правила, не так уж редко бывает, что некоторые свойства не делают буквально ничего, помимо простого присваивания и возврата значений, как в приведенном выше коде. В таких случаях было бы слишком громоздко многократно определять приватные поля и некоторые простые определения свойств. Например, при построении класса, которому нужно 15 приватных элементов данных, в конечном итоге получаются 15 связанных с ними свойств, которые, по сути, представляют собой не более чем тонкие оболочки для инкапсуляции.

Чтобы упростить процесс простой инкапсуляции данных полей, можно применять синтаксис автоматических свойств. Это средство перекладывает работу по определению лежащего в основе приватного поля и связанного свойства C# на компилятор, используя небольшое усовершенствование синтаксиса.

Переделанный класс Car, в котором этот синтаксис применяется для быстрого создания трех свойств, будет иметь вид:

```
class Car
  {
   // Автоматические свойства
   public string PetName { get; set; }
   public int Speed { get; set; }
   public string Color { get; set; }
  }
```

При определении автоматических свойств указывается модификатор доступа, лежащий в основе тип данных, имя свойства и пустые контексты get/set. Во время компиляции тип будет оснащен автоматически сгенерированным полем и соответствующей реализацией логики set/get. В отличие от

традиционных свойств C#, создавать автоматические свойства, предназначенные только для чтения или только для записи, нельзя. Определяемое автоматическое свойство должно поддерживать функциональность и чтения, и записи. Однако можно реализовать автоматическое свойство с более ограничивающими контекстами get или set:

```
public int SomeOtherProperty { get; protected set; }
```

1.5.1. Взаимодействие с автоматическими свойствами

Поскольку компилятор будет определять лежащие в основе приватные поля во время компиляции, класс с автоматическими свойствами всегда должен использовать синтаксис свойств для установки и чтения значений. Например, если бы класс Car включал метод DisplayStats(), то он должен был бы реализовать этот метод, используя имя свойства:

```
class Car
    {
     public string PetName { get; set; }
     public int Speed { get; set; }
     public string Color { get; set; }
     public void DisplayStats()
          {
           Console.WriteLine("Название: {0}", PetName);
           Console.WriteLine("Скорость: {0}", Speed);
           Console.WriteLine("Цвет: {0}", Color);
          }
    }
```

Пример 3. При использовании объекта, определенного с автоматическими свойствами, можно присваивать и получать значения, используя ожидаемый синтаксис свойств:

```
static void Main(string[ ] args)
     {
      Car c = new Car() ;
      c.PetName = "Lada";
      c.Speed = 120;
      c.Color = "Red";
      Console.WriteLine("Название авто: {0}", c.PetName);
      c.DisplayStats();
      Console.ReadLine();
     }
```

При использовании автоматических свойств для инкапсуляции числовых и булевских данных можно сразу применять автоматически сгенерированные свойства типа прямо в своей кодовой базе, поскольку их скрытым полям будут присвоены безопасные значения по умолчанию, которые могут быть использованы непосредственно. Однако, если синтаксис автоматического свойства применяется для упаковки переменной другого класса, то скрытое поле ссылочного типа также будет установлено в значение по умолчанию, т.е. null.

Пример 4. Рассмотрим следующий новый класс по имени Garage, в котором используются два автоматических свойства:

```
class Garage
    {
     public int NumberOfCars { get; set; } // Скрытое поле int установлено в 0!
     public Car MyAuto { get; set; } // Скрытое поле Car установлено в null!
    }
```

Имея установленные C# значения по умолчанию для полей данных, значение NumberOfCars можно вывести в том виде, как оно есть (поскольку ему автоматически присвоено значение 0). Однако если напрямую обратиться к MyAuto, то во время выполнения сгенерируется исключение ссылки на null, потому что лежащей в основе переменной-члену типа Car не был присвоен новый объект:

```
static void Main(string[] args)
    {
     Garage g = new Garage();
     // печатаем значение по умолчанию, равное 0.
     Console.WriteLine("Номер автомобиля: {0}", g.NumberOfCars);
     Console.WriteLine(g.MyAuto.PetName); // Ошибка! Поле равно null!
     Console.ReadLine();
    }
```

Учитывая, что приватные поля создаются во время компиляции, в синтаксисе инициализации полей C# нельзя непосредственно размещать экземпляр ссылочного типа с помощью new. Это должно делаться конструкторами класса, что обеспечит создание объекта безопасным образом. Например:

```
class Garage
    {
     public int NumberOfCars { get; set; } // Скрытое поле установлено в 0!
     public Car MyAuto { get; set; } // Скрытое поле установлено в null!
     // Для переопределения значений по умолчанию, присвоенных
     // скрытым полям, должны использоваться конструкторы,
     public Garage()
           {
            MyAuto = new Car();
            NumberOfCars = 1;
           }
      public Garage (Car car, int number)
           {
            MyAuto = car;
            NumberOfCars = number;
           }
    }
```

После этой модификации объект Car можно поместить в объект Garage, как показано ниже:

```
static void Main(string[] args)
    {
     Car c = new Car(); // Создать автомобиль
     c.PetName = "Lada";
     c.Speed =120;
     c.Color = "Red";
     c.DisplayStats();
     Garage g = new Garage();
     g.MyAuto = c; // Поместить автомобиль в гараж
     Console.WriteLine("Номер автомобиля в гараже: {0}", g.NumberOfCars);
     Console.WriteLine("Имя автомобиля: {0}", g.MyAuto.PetName);
     Console.ReadLine();
    }
```

2. Рабочее задание

Задание 1. Руководствуясь теоретическим материалом раздела 1 изучить возможности языка C# по созданию приложений, использующие свойства класса, и выполнить примеры, описанные в этом разделе.

Задание 2. Разработать приложение «Свойство», с помощью которого, используя класс Property, демонстрируется действие свойства Chislo, предназначенное для доступа к полю chislo (один из вариантов работы приложения представлен на рис.1)

Класс Property содержит поле chislo, конструктор по умолчанию, действие которого заключается в присвоении полю chislo значения 0, и метода, с помощью которого выдается на экран монитора значение поля chislo. Свойство допускает присваивание только положительных значений (если вводится отрицательное число, то выдается соответствующее сообщение, например, «Попытка ввода отрицательного числа»).

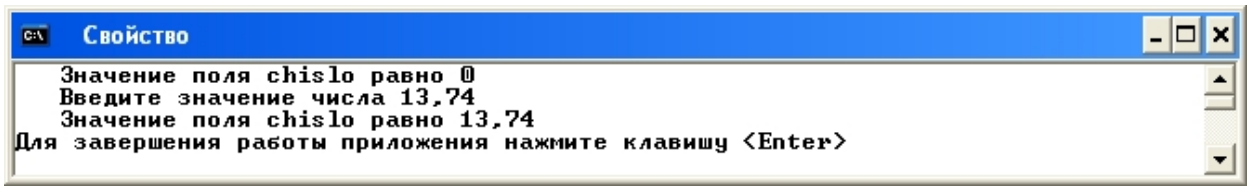

Рис. 1. Пример работы приложения «Свойство»

Задание 3. Разработать приложение «Возраст студента», с помощью которого, используя класс Student, можно определять возраст студента. Класс Student должен содержать в виде свойств следующие данные: фамилия, имя, день, месяц и год рождения.
Свойства должны содержать проверки на достоверность вводимых данных.

После запуска приложения должны вводиться данные студента. После чего вводится дата, на которую необходимо рассчитать возраст студента, и производится расчет (один из вариантов работы приложения представлен на рис.2).

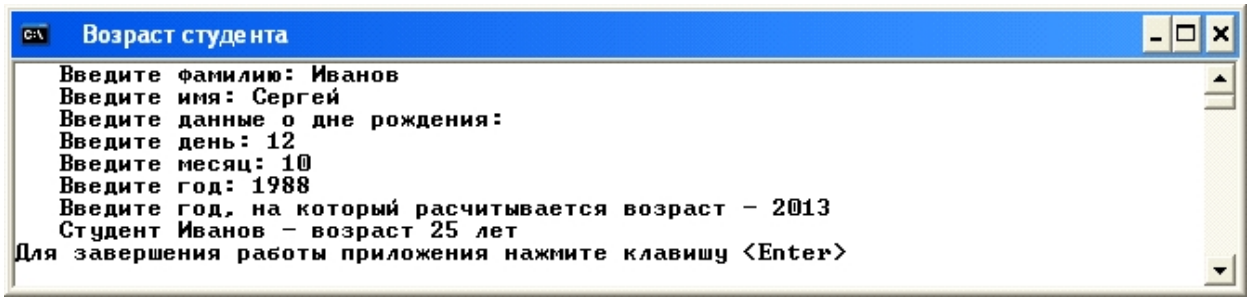

Рис. 2. Пример работы приложения «Возраст студента»

Задание 4. Модифицировать приложение «Возраст студента» таким образом, чтобы можно было вводить данные 3-х студентов и выводить возраст студента или всех вместе на любую заданную дату, не выходя из приложения.

3. Контрольные вопросы

1. Что такое свойство?
2. Перечислите способы доступа к данным объекта.
3. Как осуществляется доступ к данным объекта с использованием традиционных методов?
4. Как осуществляется доступ к данным объекта с использованием свойств .NET?
5. Как определить свойства только для чтения или только для записи?
6. Что такое статические свойства?
7. Что такое автоматические свойства?

Литература

1. Голощапов А.Л. Microsoft Visual Studio 2010. – СПб.:БХВ-Петербург, 2011. – 544 с.: ил.
2. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 1. Пер. с англ. - М.: «Русская Редакция», 2002.- 576 с.: ил.
3. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 2. Пер. с англ. - М.: «Русская Редакция», 2002.- 624 с.: ил.
4. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0. Пер. с англ. - М.: Издательский дом "Вильямс", 2011. — 1392 с.: ил.
5. Фленов М.Е. Библия C#. - СПб.: БХВ-Петербург, 2011.– 560с.: ил.
6. Шилдт Г. C# Учебный курс . – СПб.: Питер, Издательская группа BHV, 2003. – 512 с.: ил.