Объектно-ориентированное программирование (ООП)

Класс и **Объект**

- **Класс** – это абстрактный тип данных. С помощью класса описывается некоторая сущность (характеристики и возможные действия)
- Описав класс, мы можем создать его экземпляр –**объект**. Объект – это уже конкретный представитель класса.
- **Класс** – это абстрактное представление чего-либо.
- **Объект** - используемый экземпляр того, что представляет класс.

Основные принципы объектно-ориентированного программирования

- Инкапсуляция
- Наследование
- Полиморфизм
- Абстракция

Инкапсуляция

- **Инкапсуляция** – позволяет скрывать внутреннюю реализацию.
- В классе могут быть реализованы внутренние вспомогательные методы, поля, к которым доступ для пользователя необходимо запретить.

Наследование

- **Наследование** – позволяет создавать новый класс на базе другого.
- Класс, на базе которого создается новый класс, называется базовым, а базирующийся новый класс – наследником.
  - Например, есть базовый класс животное. В нем описаны общие характеристики для всех животных (класс животного, вес). На базе этого класса можно создать классы наследники (Собака, Слон) со своими специфическими свойствами.
  - Все свойства и методы базового класса при наследовании переходят в класс наследник.

Полиморфизм

- **Полиморфизм** – это способность объектов с одним интерфейсом иметь различную реализацию.
  - Например, есть два класса, Круг и Квадрат. У обоих классов есть метод **GetSquare()**, который считает и возвращает площадь. Но площадь круга и квадрата вычисляется по-разному, соответственно, реализация одного и того же метода различная.

Абстракция

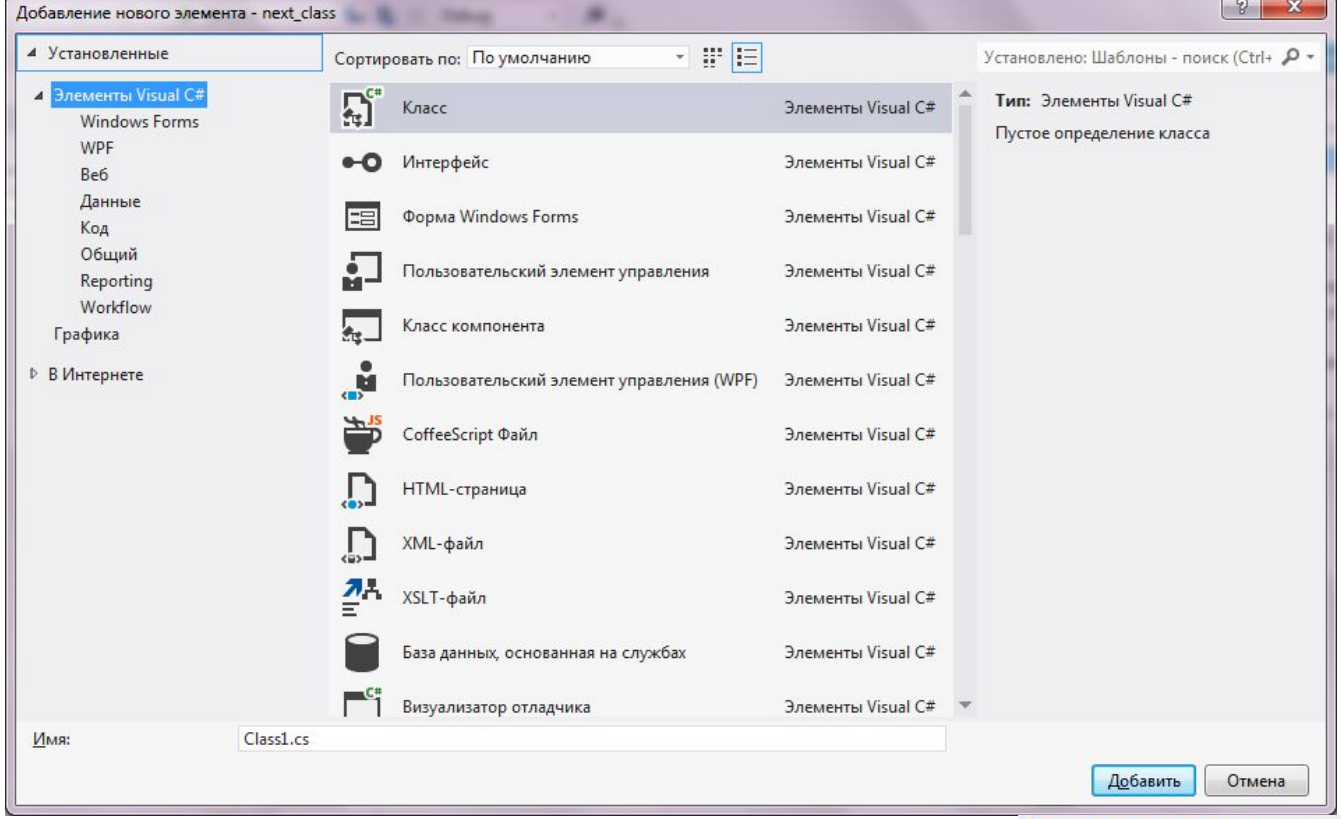
- **Абстракция** – позволяет выделять из некоторой сущности только необходимые характеристики и методы, которые в полной мере (для поставленной задачи) описывают объект.
  - Например, создавая класс для описания студента, мы выделяем только необходимые его характеристики, такие как ФИО, номер зачетной книжки, группа. Здесь нет смысла добавлять поле вес или имя его кота/собаки и т. д.

Класс

- **Класс** – это абстрактный тип данных. Другими словами, класс – это некоторый шаблон, на основе которого будут создаваться его экземпляры – **объекты**.

[модификатор доступа] class [имя_класса]
{
//тело класса
}

- *public* – доступ к классу возможен из любого места одной сборки либо из другой сборки, на которую есть ссылка;
- *internal* – доступ к классу возможен только из сборки, в которой он объявлен

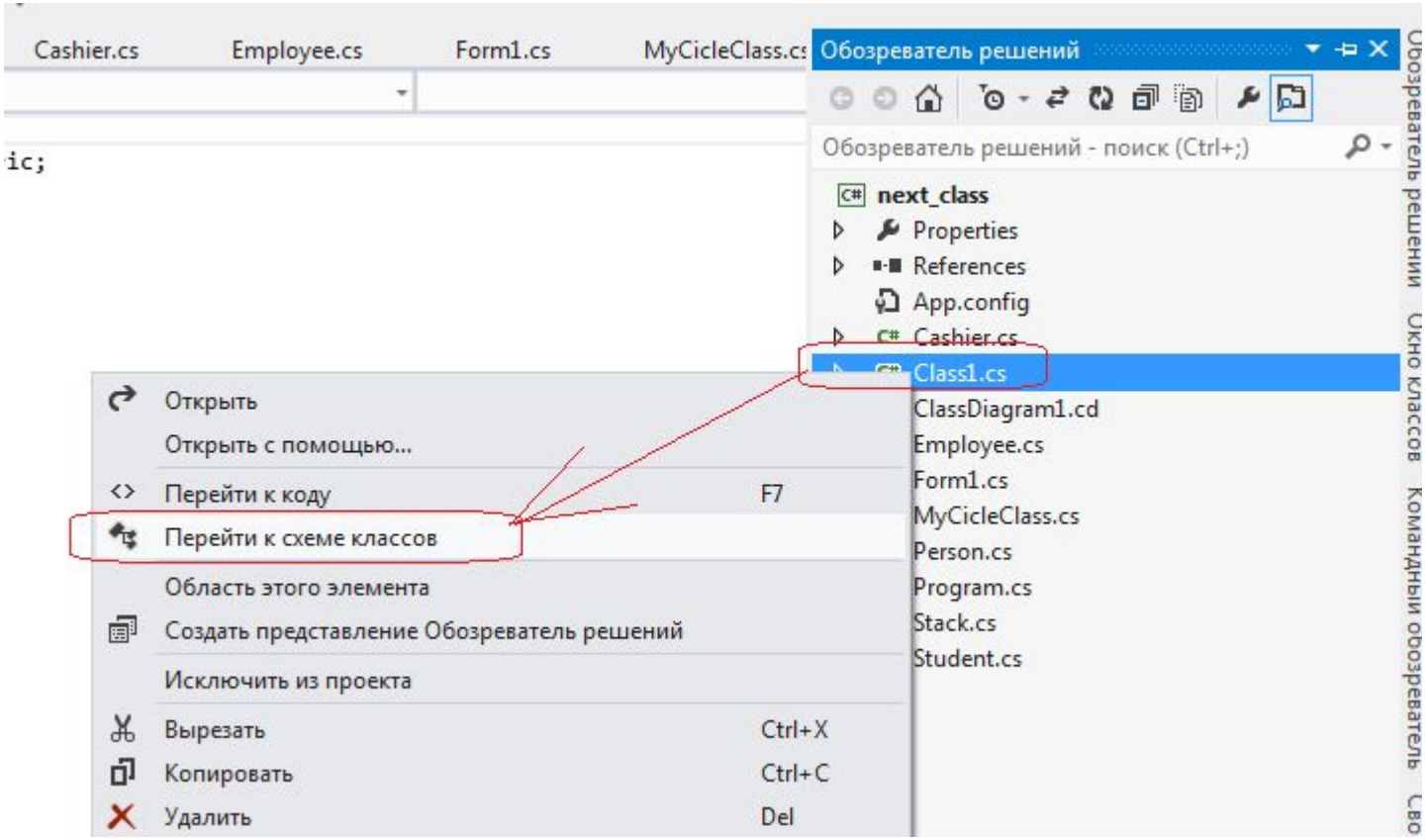


Class1.cs | Student.cs | Person.cs | Cashier.cs

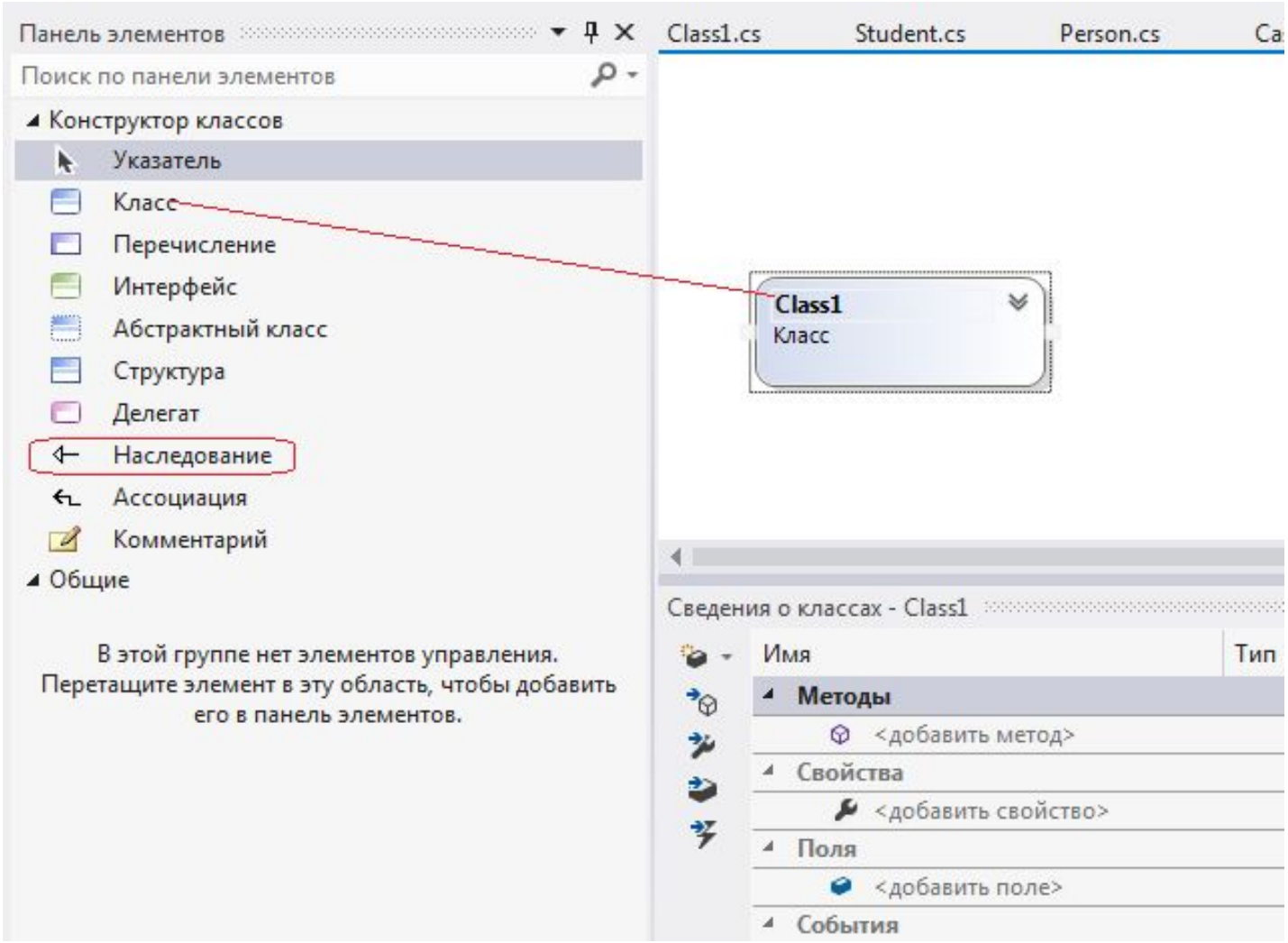
next_class.Class1

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace next_class
8  {
9      class Class1
10     {
11     }
12 }
13
```

Схема Класса



Инструменты схемы классов



Члены класса

- *поля*;
- *константы*;
- *свойства*;
- *конструкторы*;
- *методы*;
- *события*;
- *операторы*;
- *индексаторы*;
- *вложенные типы*.

Поля класса

- **Поля** служат для хранения данных, содержащихся в объекте. Поля аналогичны переменным, т.к. они непосредственно читаются и устанавливаются.
- **Поле** – это переменная, объявленная внутри класса.
  - Как правило, поля объявляются с модификаторами доступа ***private*** либо ***protected***, чтобы запретить прямой доступ к ним.
  - Для получения доступа к полям следует использовать **свойства** или **методы**.

Закрытые, защищенные и открытые поля

- **Private** – "объекты только этого класса могут обращаться к данному полю".
- **Public** – "объекты любого класса могут обращаться к этому полю".
- **Protected** – "только объекты классов-наследников могут обращаться к полю".
  - Если построен класс *Animal*, то другой класс, например, класс *Mammal* (Млекопитающее), может объявить себя наследником класса *Animal*.

Константы

- **Константы-члены класса** ничем не отличаются от простых констант.
- **Константа** – это переменная, значений которой нельзя изменить. Константа объявляется с помощью ключевого слова *const*. Пример объявления константы:

```
class Math
{
   private const double Pi = 3.14;
}
```

Методы

- **Методами** называют действия, которые объект может выполнять.
- Метод позволяет описать порядок выполнения определенных действий.
- Описание метода называется программным кодом или просто кодом.
- Методы позволяют сократить объем кода.

Методы

```
public void SayHello ()
    {
        string fullname;
        fullname ="Hellow, " + FirstName + " " + Surname;
    }
```

Использование слова **void** перед именем метода означает, что, когда завершается выполнение метода, возвращается пустое значение, то есть по завершении определенных действий, которые выполняет метод, он никаких значений не возвращает.

Методы

```
namespace next_class
{
    public class Employee
    {
        //поле
        public string FirstName;
        public string Surname;

        public void SayHello(out string fullname)
        {
            // throw new System.NotImplementedException();

            fullname ="Hello, " + FirstName + " " + Surname;
            return;

        }
```

Вызов из формы

```
private void button2_Click(object sender, EventArgs e)
{
    string EmpName;
    //создание экземпляра класса Employee
    Employee emp = new Employee();
    emp.FirstName = "Nata";
    emp.Surname = "Averina";
      emp.SayHello(out EmpName);
    MessageBox.Show(EmpName);
```

- **Статический метод** – это метод, который не имеет доступа к полям объекта, и для вызова такого метода не нужно создавать экземпляр (объект) класса, в котором он объявлен.
- **Простой метод** – это метод, который имеет доступ к данным объекта, и его вызов выполняется через объект.

Простой метод

- Класс *Телевизор*, у него есть поле *switchedOn*, которое отображает состояние включен/выключен, и два метода – включение и выключение:

```
class TVSet
{
  private bool switchedOn;

  public void SwitchOn()
  {
    switchedOn = true;
  }
  public void SwitchOff()
  {
    switchedOn = false;
  }
}
class Program
{
  static void Main(string[] args)
  {
    TVSet myTV = new TVSet();
    myTV.SwitchOn(); // включаем телевизор, switchedOn = true;
    myTV.SwitchOff(); // выключаем телевизор, switchedOn = false;
  }
}
```

- Чтобы вызвать простой метод, перед его именем, указывается имя объекта. Для вызова статического метода необходимо указывать имя класса.

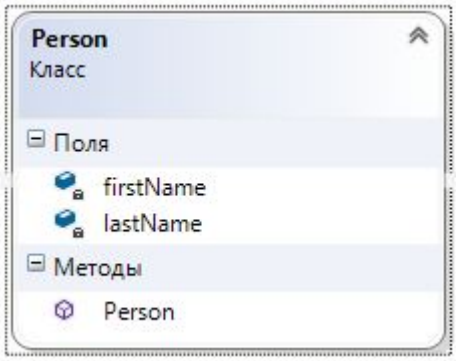
- Пример статического метода, который обрезает строку до указанной длины, и добавляет многоточие:

```
class StringHelper
{
  public static string TrimIt(string s, int max)
  {
    if (s == null)
      return string.Empty;
    if (s.Length <= max)
      return s;

    return s.Substring(0, max) + "...";
  }
}
class Program
{
  static void Main(string[] args)
  {
    string s = "Очень длинная строка, которую необходимо обрезать до указанной длины и добавить многоточие";
    Console.WriteLine(StringHelper.TrimIt(s, 20)); //"Очень длинная строка…"
    Console.ReadLine();
  }
}
```

Конструктор

- **Конструктор** – это метод класса, предназначенный для инициализации объекта при его создании.
- **Инициализация** – это задание начальных параметров объектов/переменных при их создании.
- Особенностью конструктора, как метода, является то, что его имя всегда совпадает с именем класса, в котором он объявляется.
- При этом, при объявлении конструктора, не нужно указывать возвращаемый тип, даже ключевое слово *void*.
- Конструктор следует объявлять как *public*, иначе объект нельзя будет создать

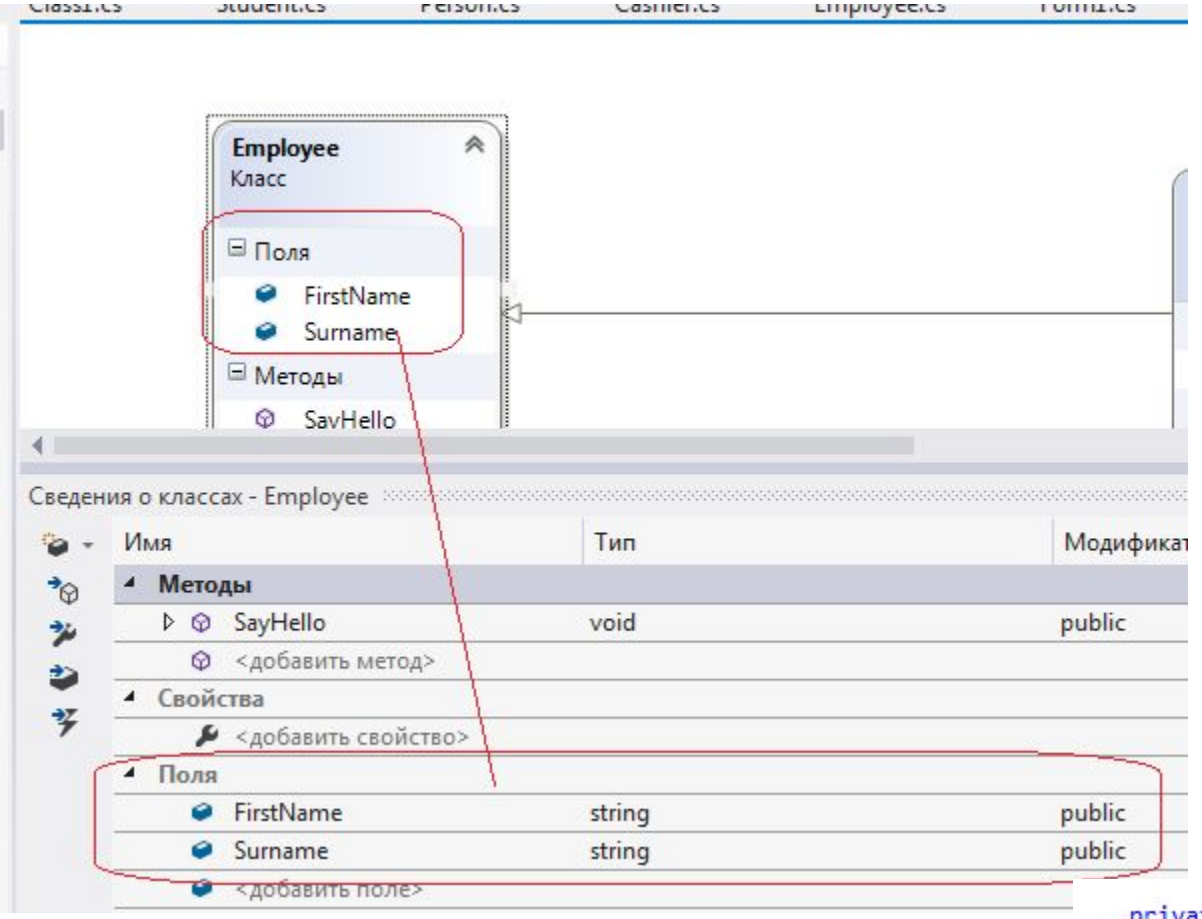



```
namespace next_class
{
    public class Person
    {
        // Поля
        string firstName;
        string lastName;
        // Первый метод-конструктор
        public Person()
        {
            firstName = "Nike";
            lastName = "Black";
        }

        // Второй метод-конструктор
        public Person(string f, string l)
        {
            this.firstName = f;
            this.lastName = l;
        }
    }
}
```

- Указатель **this** - это указатель на объект, для которого был вызван нестатический метод.
- Ключевое слово ***this*** обеспечивает доступ к текущему экземпляру класса.
- Классический пример использования *this*, это как раз в конструкторах, при одинаковых именах полей класса и аргументов конструктора.
- Ключевое слово *this* это что-то вроде имени объекта, через которое мы имеем доступ к текущему объекту.

Поля класса Employee




```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace next_class
7  {
8      public class Employee
9      {
10         //поле
11          public string FirstName;
12          public string Surname;
```

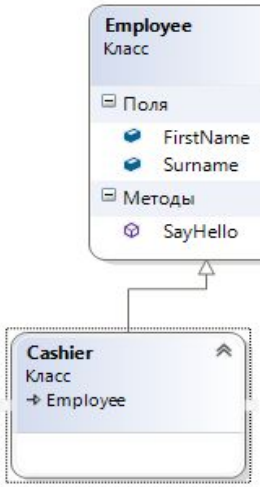
Доступ к полю в форме

```
private void button2_Click(object sender, EventArgs e)
{
    string EmpName;
    //создание экземпляра класса Employee
    Employee emp = new Employee();
    emp.FirstName = "Nata";
    emp.Surname = "Averina";
```

Наследование класса

- class [имя_класса] : [имя_базового_класса]
  {
  // тело класса
  }
- В программировании **наследование** позволяет создавать новый класс на базе другого.
- Класс, на базе которого создается новый класс, называется **базовым**, а базирующийся новый класс – **наследником** или **производным классом**.
- В класс-наследник из базового класса переходят поля, свойства, методы и другие члены класса.

Наследование класса




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace next_class
{
    //Новый класс Cashier потомок класса Employee
    public class Cashier : Employee
    {
     //   Inherits Employee на VB

    }
}
```

```
namespace next_class
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button2_Click(object sender, EventArgs e)
        {
            string EmpName;
            //создание экземпляра класса Employee
            Employee emp = new Employee();
            emp.FirstName = "Nata";
            emp.Surname = "Averina";
              emp.SayHello(out EmpName);
            MessageBox.Show(EmpName);
            //создание экземпляра класса Cashier наследника класса Employee
            Cashier cash = new Cashier();
            cash.FirstName = "Masha";
            cash.Surname = "Belova";
            cash.SayHello(out EmpName);
            MessageBox.Show(EmpName);

        }
    }
}
```

Свойства

- Представляют собой способ доступа к полям объекта
- Для использующих класс программ свойства выглядят как поля с данными, однако внутри класса являются кодом – специальным методом для работы с полями.
- В свойство можно поместить проверку допустимости значения.
- Внутри тела свойства используются специальные процедуры аксессоры (accessors) –**Get** и **Set**.
- Обеспечивают контроль над процессом присваивания или возврата значений полей, что позволяет изолировать и проверять данные до изменения или чтения значений.

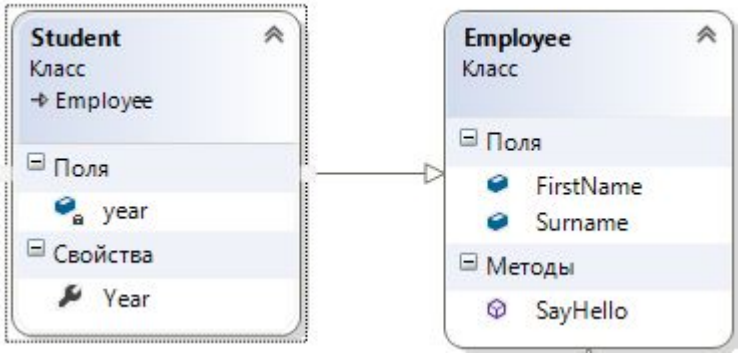
Свойства

- Свойства предоставляют удобный механизм доступа к полю класса (чтение поля и запись). Свойство представляет собой что-то среднее между полем и методом класса.
- **[модификатор доступа] [тип] [имя_свойства]**

```
{
  get
  {
    // тело аксессора для чтения из поля
  }

  set
  {
    // тело аксессора для записи в поле
  }
}
```

Свойства

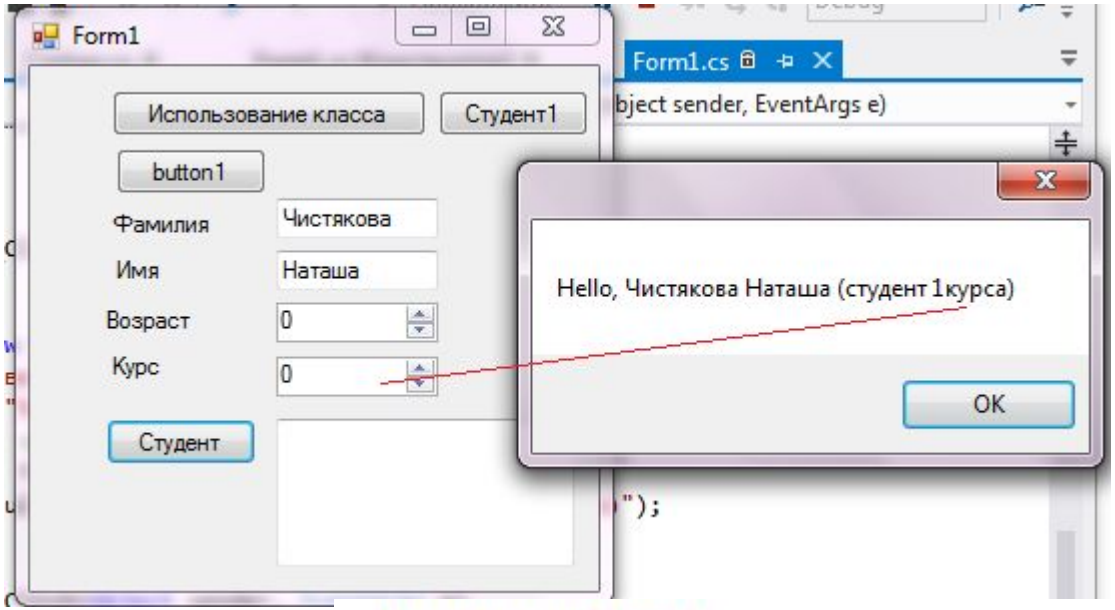



```
public class Student : Employee
{
    private int year;//объявление закрытого поля
    //закрытое поле курс, которое не может быть ниже единицы и больше пяти
    public int Year//объявление свойства
    {
        get// аксессор чтения поля
        {
            //throw new System.NotImplementedException();
            return year;
        }
        set// аксессор записи в поле
        {
            if (value < 1)
                year = 1;
            else if (value > 5)
                year = 5;
            else year = value;
        }
    }
```

```
private void button3_Click(object sender, EventArgs e)
{
    string fullname;
     Student stud = new Student();
     stud.Surname = "Иванов";
     stud.FirstName = "Вася";
     stud.Year = 0; // записываем в поле, используя аксессор set
     stud.SayHello(out fullname);
     MessageBox.Show(fullname + " (студент " + stud.Year + "курса)");
}
```

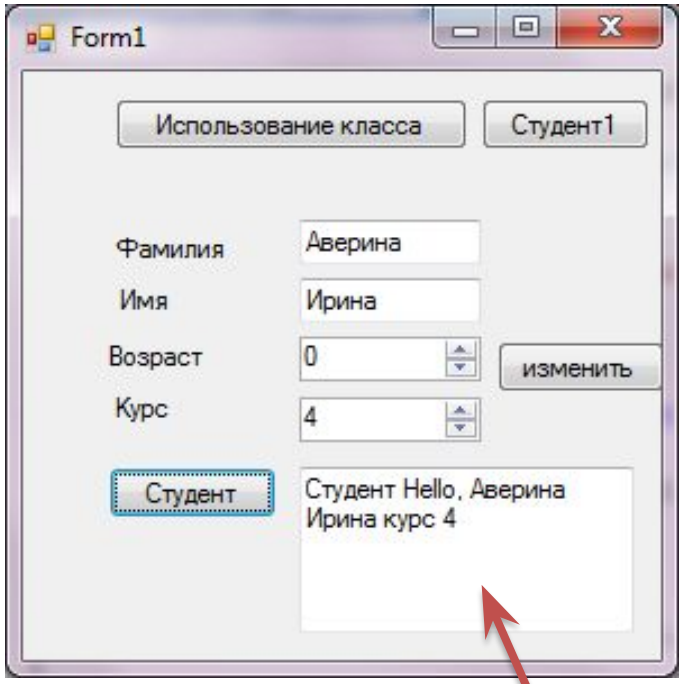
Если бы просто сделали поле **year** открытым и не использовали ни методы, ни свойство для доступа, мы могли бы записать в это поле любое значение, в том числе и некорректное, а так мы можем контролировать чтение и запись.

Свойства




```
// кнопка Студент -
private void button4_Click(object sender, EventArgs e)
{
    string fullname;
    Student stud = new Student();
    stud.Surname = this.textBox1.Text;
    stud.FirstName = this.textBox2.Text;
    stud.Year = Convert.ToInt16(this.YearnumericUpDown.Value);
    stud.SayHello(out fullname);
    MessageBox.Show(fullname + " (студент " + stud.Year + "курса)");
    this.textBox3.Text = "Студент " + fullname + " курс " + stud.Year ;

}
```




```
// кнопка Студент -
private void button4_Click(object sender, EventArgs e)
{
    string fullname;
    Student stud = new Student();
    stud.Surname = this.textBox1.Text;
    stud.FirstName = this.textBox2.Text;
    stud.Year = Convert.ToInt16(this.YearnumericUpDown.Value);
    stud.SayHello(out fullname);
    MessageBox.Show(fullname + " (студент " + stud.Year + "курса)");
    this.textBox3.Text = "Студент " + fullname + " курс " + stud.Year ;

}
```

События

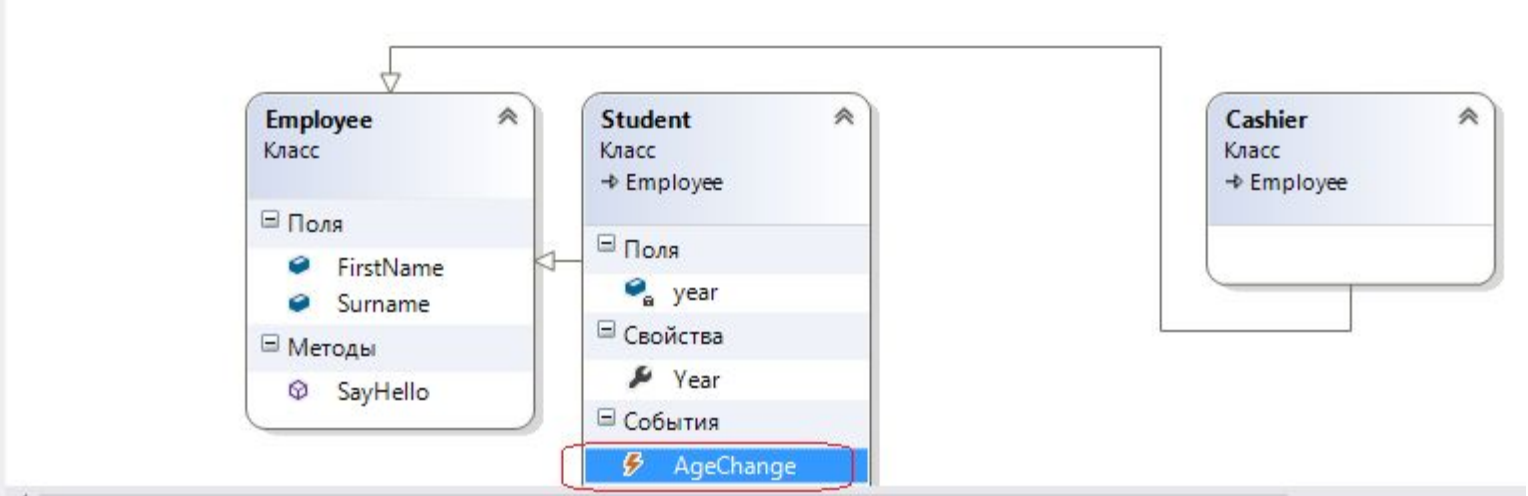
- Применения **событий** позволяет объектам реагировать на те или иные ситуации и выполнять необходимые ответные действия.
- Событие может вызываться только в том классе, где оно объявлено

События и делегаты

- Бывают ситуации, когда программисту требуется написать гибкий код, реагирующий на выполнение конкретных операций.
- Например были созданы различные ЭУ (кнопки, поля и т. д.). Но разработчики не знали, как мы будем их использовать. ЭУ имеют точки взаимодействия , встроенные в код таким образом, чтобы они могли поддерживать связь с вашей программой. Эти точки взаимодействия называются **событиями (events)**. События срабатывают всякий раз, когда пользователь выполняет определенные действия(нажимает на кнопку).
- Программисты пишут код, который связывает эти события с другим кодом в разрабатываемой программе и должен выполняться при наступлении этих событий (нажатие кнопки).
- Для выполнения этой задачи используются **делегаты (delegates**).

События и делегаты

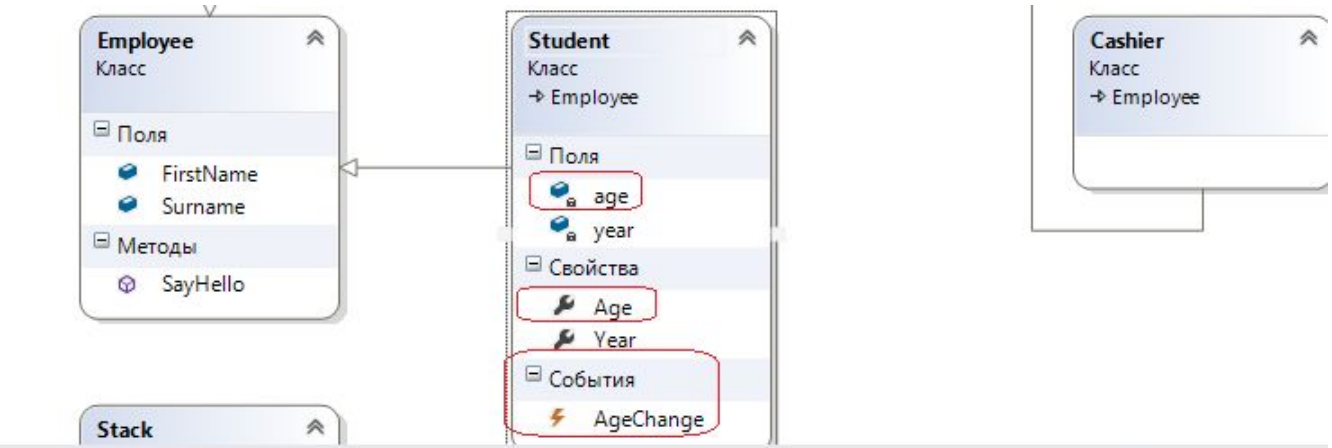
- Событие определяет тип уведомлений, которые могут предоставляться объектом, и делегат позволяет связать событие с тем кодом, которое должен исполняться при его наступлении.
- Событие – это тип класса , который позволяет вашему классу или экземпляру класса уведомлять другой код о том, что происходит в пределах этого класса.



Сведения о классах - Student

Имя	Тип	Модификато
Методы		
<добавить метод>		
Свойства		
Year	int	public
<добавить свойство>		
Поля		
year	int	private
<добавить поле>		
События		
AgeChange	EventHandler	public
<добавить событие>		

```
6   namespace next_class
7   {
8       public class Student : Employee
9       {
10          private int year;
11          // Событие
12          public event EventHandler AgeChange;
13          //объявление закрытого поля
```



Сведения о классах - Student

Имя	Тип	Модифик
Методы		
<добавить метод>		
Свойства		
Age	int	public
Year	int	public
<добавить свойство>		
Поля		
age	int	private
year	int	private
<добавить поле>		
События		
AgeChange	EventHandler	public
<добавить событие>		

Чтобы сгенерировать событие, нужно просто вызвать его как простой метод, например:

```
AgeChanged(this, new EventArgs());
```

Событие `AgeChanged` у нас объявлено как делегат `EventHandler`. Этот делегат получает в качестве параметров объект, который сгенерировал событие, и пустой экземпляр класса `EventArgs`. Чтобы передать объект, мы просто передаем в первом параметре `this`, а во втором параметре создаем экземпляр класса `EventArgs`.

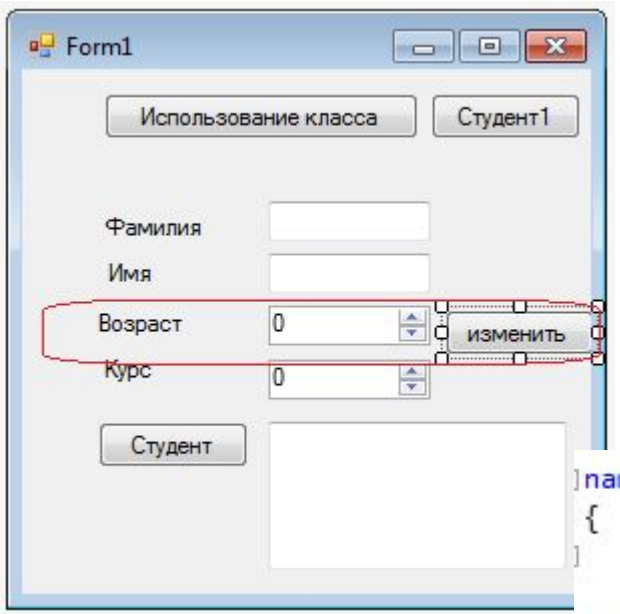
```
public class Student : Employee
{
    //поля
    private int age = 0;
    private int year;
    // Событие
    public event EventHandler AgeChange;
    // поле
    public int Age
    {
        get// аксессор чтения поля
        {
            // throw new System.NotImplementedException();
            return age;
        }
        set// аксессор записи в поле
        {
            if (value < 0)
                throw new Exception("Возраст не может быть отрицательным");
            age = value;
            if (AgeChange != null)
                AgeChange(this, new EventArgs());

        }
    }
```

С помощью ключевого слова event объявляем событие **AgeChange** класса **Event Hander.**

Event Hander представляет собой класс, который принадлежит библиотеки .NET Framework

Чтобы сгенерировать событие нужно вызвать как простой метод




```
namespace next_class
{
    public partial class Form1 : Form
    {
        Student st = new Student();
        public Form1()
        {
            InitializeComponent();
            agenumericUpDown.Value = st.Age;
            //регистрация события
            st.AgeChange += new EventHandler(AgeChange);
        }
        public void AgeChange(Object sender, EventArgs args)
        {
            Student st = (Student)sender;
            MessageBox.Show("Возраст изменился на " + st.Age.ToString());

        }
        //кнопка изменения возраста
        private void ageChangeButton_Click(object sender, EventArgs e)
        {
            st.Age = (int)agenumericUpDown.Value;
        }
```

Чтобы **добавить свой объект в качестве получателя события** , нужно выполнить операцию добавления к текущему значению нового экземпляра обработчика с помощью операции **+=**
Если нужно у**далить** обработчик события, то выполняется операция **-=**

Наше событие является **делегатом EvenHander**, следовательно мы должны добавить экземпляр EvenHander. Передаем делегату метод AgeChange.

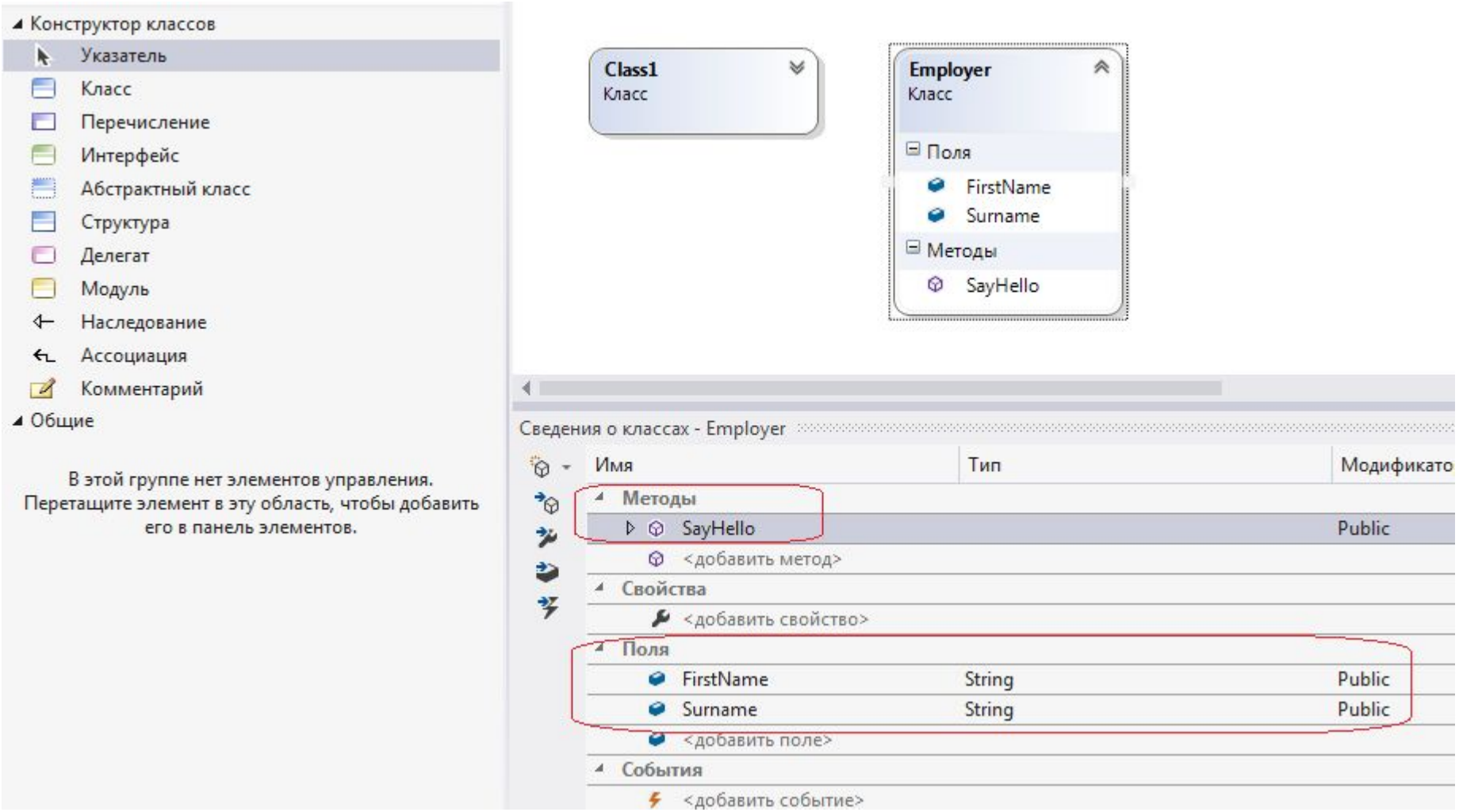
Процедуры и функции – методы класса

- Метод функциональной декомпозиции - декомпозиция главной функции на подфункции, решающие частные задачи.
  - Первыми формами модульности, появившимися в языках программирования, были **процедуры** и **функции**
  - Один раз написанную функцию можно многократно вызывать в программе с разными значениями параметров, передаваемых функции в момент вызова.
  - Важным шагом в автоматизации программирования было появление библиотек процедур и функций, доступных из языка программирования
- Для языков ООП, к которым относится и язык C#, роль архитектурного модуля играет **класс**
  - Процедуры и функции связываются теперь с классом, они обеспечивают требуемую функциональность класса и называются **методами** класса.
  - Главную роль в классе начинают играть его данные – **поля** класса, задающие свойства объектов класса.
  - Прежнюю роль библиотек процедур и функций теперь играют **библиотеки классов**.

Процедуры и функции – методы класса

- имя_метода([список_фактических_аргументов])
- синтаксис объявления формального аргумента:
- **[ref | out | params] тип_аргумента имя_аргумента.**
- Выходные аргументы всегда должны сопровождаться ключевым словом **out,** обновляемые - **ref.**
- Входные аргументы, как правило, задаются без ключевого слова, хотя иногда их полезно объявлять с параметром **ref.**

Классы в VB


Конструктор классов
Указатель
Класс
Перечисление
Интерфейс
Абстрактный класс
Структура
Делегат
Модуль
Наследование
Ассоциация
Комментарий
Общие
В этой группе нет элементов управления.
Перетащите элемент в эту область, чтобы добавить его в панель элементов.
Class1
Класс
Employer
Класс
Поля
FirstName
Surname
Методы
SayHello
Сведения о классах - Employer
Имя
Тип
Модификато
Методы
SayHello
Public
<добавить метод>
Свойства
<добавить свойство>
Поля
FirstName
String
Public
Surname
String
Public
<добавить поле>
События
<добавить событие>

Добавление методов

```
Sub MyMetod(ByVal MyMethParam As String)
……..
End Sub

Function MyFunc(ByVal MyFuncParam As Integer) as String
…….
MyFunc=выражение
End Functon
```

Методы

- Методы являются обычными процедурами или функциями
- Функция всегда возвращает результат
- Подпрограмма для возвращения результата может использовать параметры
  - Передача параметров по значению (ByVal). Режим по умолчанию. Значения переменных , которые используются в качестве параметров, невозможно изменить в теле подпрограммы
  - Передача параметров по ссылке (ByRef). В теле вызываемой подпрограммы можно изменять значения тех переменных, которые ей передаются в качестве параметров

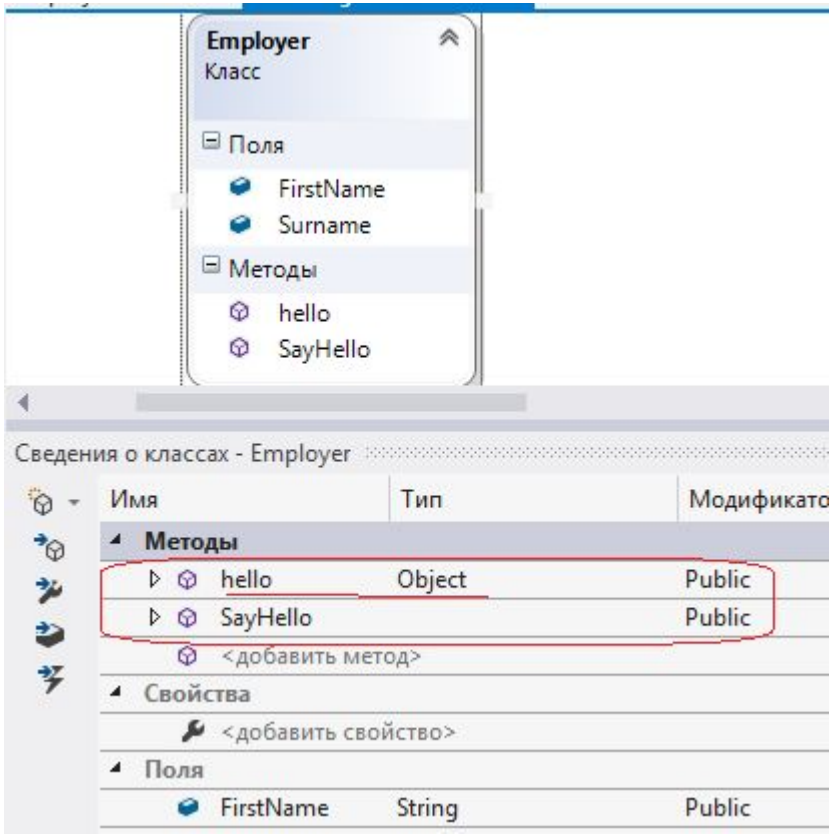
Классы в VB

Employer.vb* | ClassDiagram1.cd* | Class1.vb

(Общие)

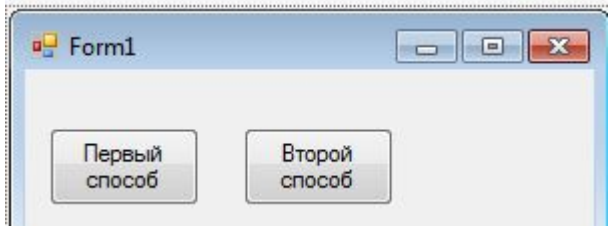
```
1
2  Public Class Employer
3      Public FirstName As String
4      Public Surname As String
5
6      Public Sub SayHello()
7
8      End Sub
9
10 End Class
11
```

```
1
2  Public Class Employer
3      'Описание полей
4      Public FirstName As String
5      Public Surname As String
6      'Создание метода при помощи подпрограмы
7      Public Sub SayHello(ByRef fullname As String)
8          fullname = Surname & " " & FirstName
9      End Sub
10     'Создание метода при помощи функции
11     Public Function hello()
12         hello = Surname & " " & FirstName
13     End Function
14 End Class
```



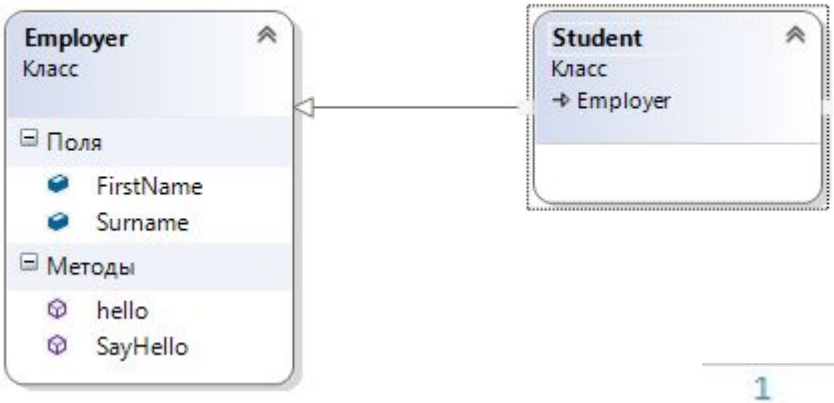
Классы в VB

```vb
Public Class Employer
    'Описание полей
    Public FirstName As String
    Public Surname As String
    'Создание метода при помощи подпрограмы
    Public Sub SayHello(ByRef fullname As String)
        fullname = Surname & " " & FirstName
    End Sub
    'Создание метода при помощи функции
    Public Function hello()
        hello = Surname & " " & FirstName
    End Function
End Class
```




```vb
Public Class Form1
    'Кнопка первый способ
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim EmpName As String
        'создание и объявление нового объекта
        Dim Emp As Employer = New Employer()
        Emp.FirstName = "Nata"
        Emp.Surname = "Averina"
        Emp.SayHello(EmpName)
        MessageBox.Show(EmpName)
    End Sub
    'Кнопка певторой способ
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim EmpName As String
        'создание и объявление нового объекта
        Dim Emp As New Employer
        Emp.FirstName = "Nata"
        Emp.Surname = "Averina"
        EmpName = Emp.hello()
        MessageBox.Show(EmpName)
    End Sub
End Class
```

Наследование




```
1
2  Public Class Student
3      Inherits Employer
4
5  End Class
6
```

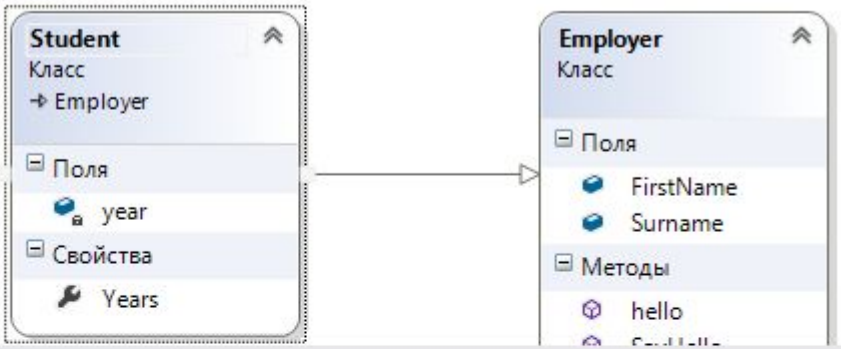
Свойство

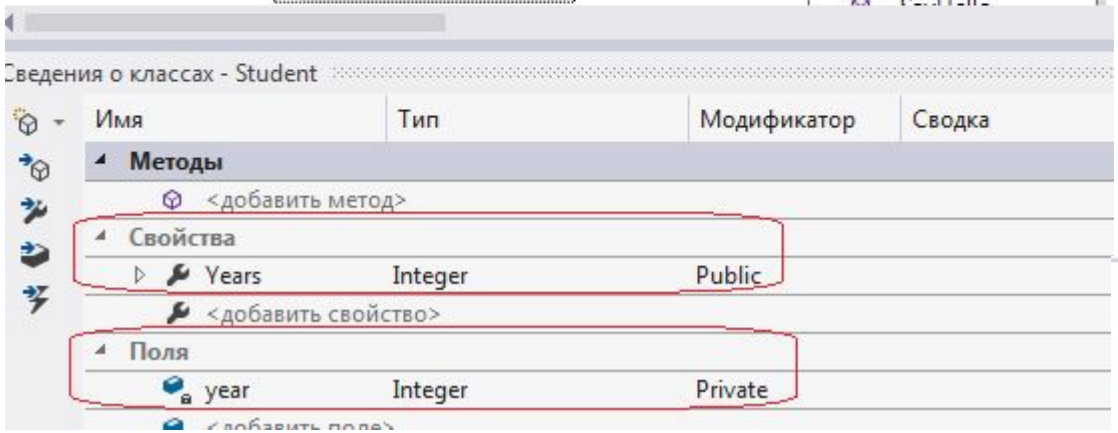
```
Public Property MyProperty() as String
    Get
        Return FmyField
    End Get
    Set(ByVal Value As String)
        FmyField=Value
    End Set
    End Property
```

Обязательный оператор для возврата значения

Указание значения

Свойства

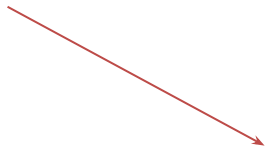




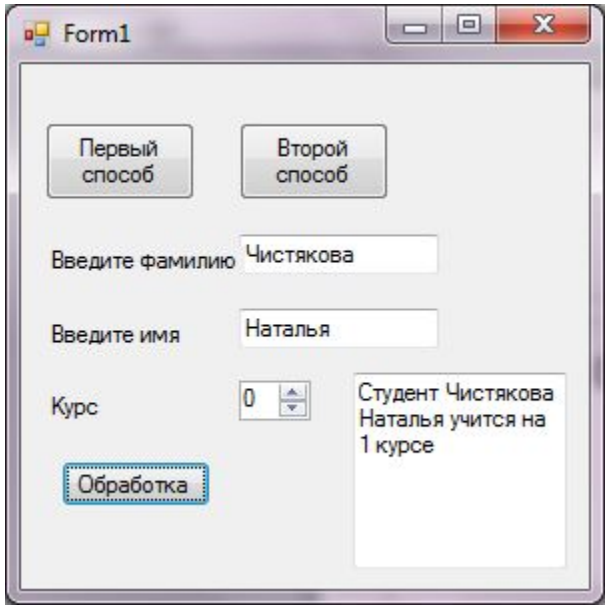

```
1
2   Public Class Student
3       Inherits Employer
4       Private year As Integer
5
6       Public Property Years As Integer
7           Get
8
9           End Get
10          Set(value As Integer)
11
12          End Set
13      End Property
14
15  End Class
```

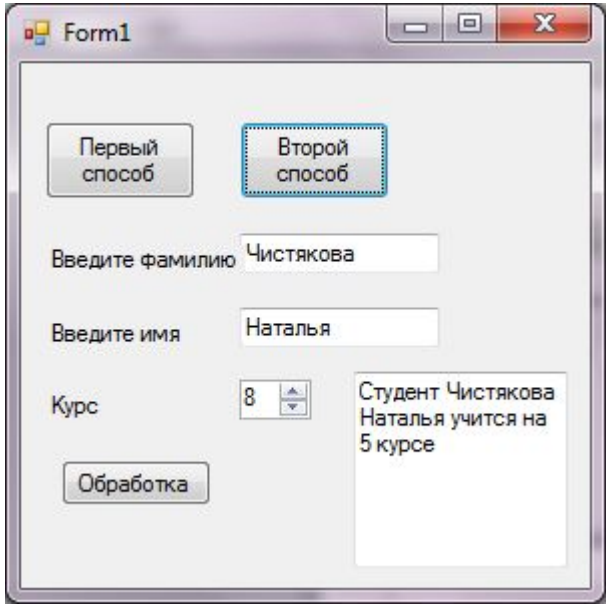
Свойства

```
1
2  Public Class Student
3      Inherits Employer
4      Private year As Integer
5
6      Public Property Years As Integer
7          Get
8
9          End Get
10         Set(value As Integer)
11
12         End Set
13     End Property
14
15 End Class
```



```
1
2  Public Class Student
3      Inherits Employer
4      Private year As Integer
5
6      Public Property Years As Integer
7          'аксессор чтения поля
8          Get
9              Return year
10         End Get
11         'аксессор записи поля
12         Set(value As Integer)
13             If value < 1 Then
14                 year = 1
15             ElseIf value > 5 Then
16                 year = 5
17             End If
18         End Set
19     End Property
20
21 End Class
```

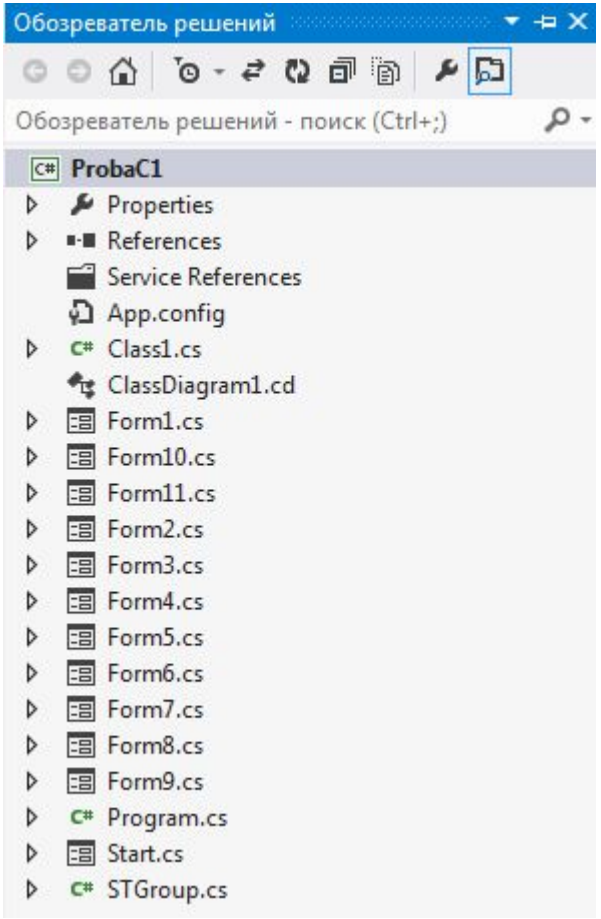





```
    'Кнопка Обработка
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        Dim StName As String
        'создание и объявление нового объекта
        Dim Stud As New Student
        Stud.FirstName = Me.TextBox2.Text
        Stud.Surname = Me.TextBox1.Text
        Stud.Years = Convert.ToInt16(Me.YearNumericUpDown.Value)
        StName = Stud.hello()
        'MessageBox.Show(StName)
        Me.TextBox3.Text = "Студент " & StName & " учится на " & Stud.Years & " курсе"
    End Sub
```

Структура проекта

- Проект представляет собой совокупность файлов, которые компилятор использует для создания выполняемого файла.
- Основными элементами проекта являются:
  - главный модуль приложения (файл Program.css);
  - модули форм.



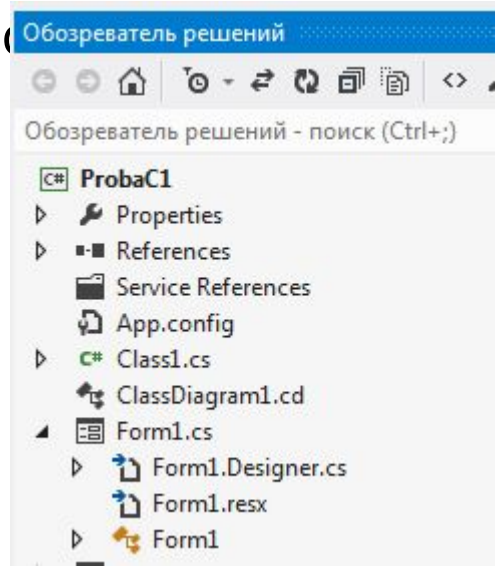
Главный модуль

- В главном модуле находится функция **Main**, с которой начинается выполнение программы. Функция Main создает стартовую форму (имя класса стартовой формы указывается в качестве параметра метода **Run**)

```
    {
        /// <summary>
        /// Главная точка входа для приложения.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form11());
        }
    }
```

Модуль формы

- Модуль формы содержит объявление класса (
  модуль формы разделен на два файла:
  - Form1.cs
  - Form1.Designer.cs



Form1.Designer.cs

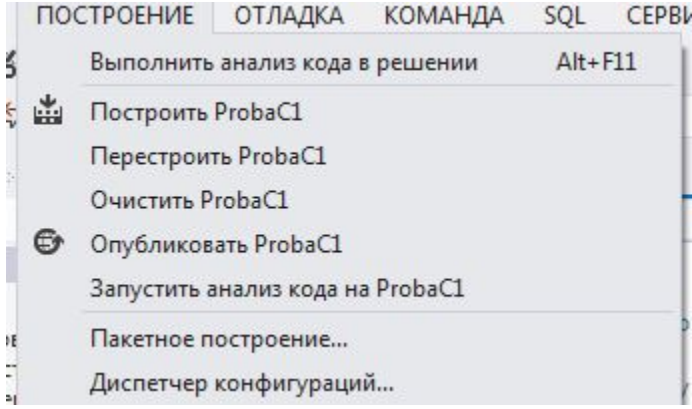
```
1  namespace ProbaC1
2  {
3      partial class Form1
4      {
5          /// <summary>
6          /// Требуется переменная конструктора.
7          /// </summary>
8          private System.ComponentModel.IContainer components = null;
9
10         /// <summary>
11         /// Освободить все используемые ресурсы.
12         /// </summary>
13         /// <param name="disposing">истинно, если управляемый ресурс до
14         protected override void Dispose(bool disposing)
15         {
16             if (disposing && (components != null))
17             {
18                 components.Dispose();
19             }
20             base.Dispose(disposing);
21         }
22
23         Код, автоматически созданный конструктором форм Windows
71
72         private System.Windows.Forms.Label label1;
73         private System.Windows.Forms.Button button1;
74     }
75 }
76
```

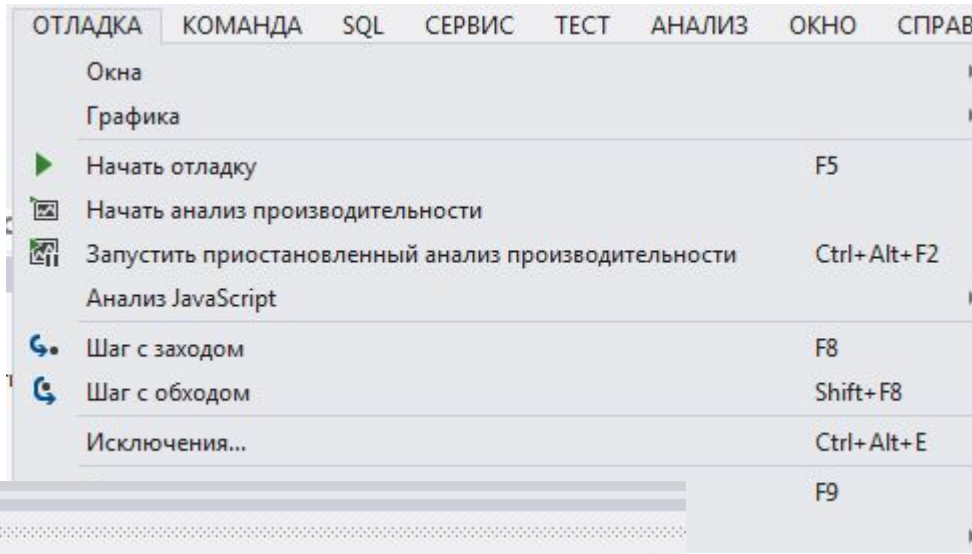
- находится объявление класса формы, в том числе сформированная дизайнером формы функция **InitializeComponent**, обеспечивающая создание и настройку компонентов

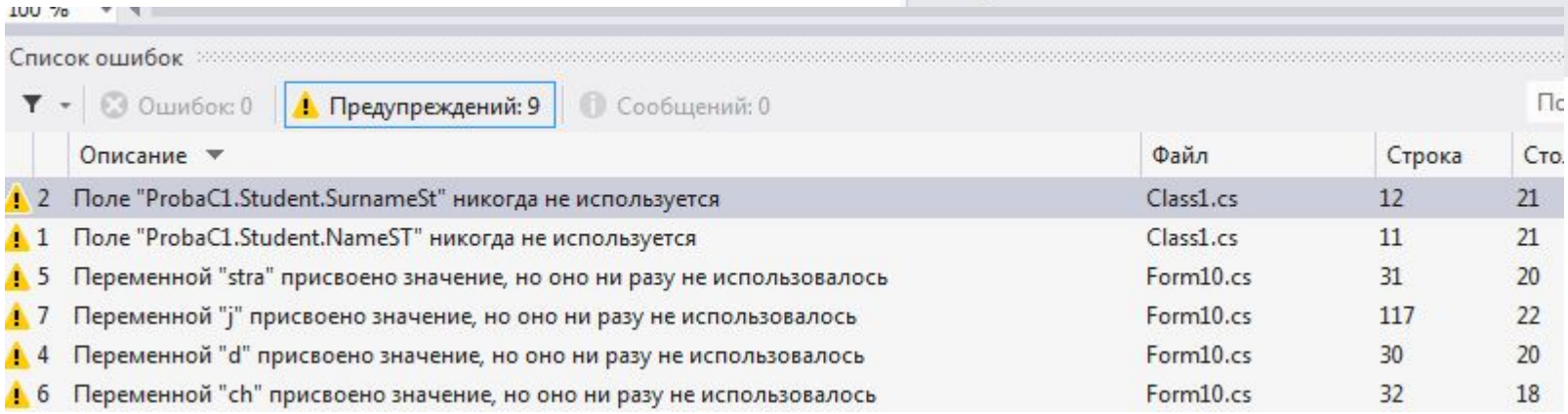
Компиляция

- Процесс преобразования исходной программы в выполняемую называется ***компиляцией*** или построением (**build**).
- Укрупненно процесс построения программы можно представить как последовательность двух этапов:
  - компиляция и
  - компоновка.
- На этапе компиляции выполняется перевод исходной программы (модулей) в некоторое внутреннее представление.
- На этапе компоновки — объединение модулей в единую программу.

- Процесс построения программы активизируется в результате выбора в меню
  - Построение-Построить
  - Отладка –Начать отладку, если с момента последней компиляции в программу были внесены изменения
- Результат компиляции отражается в окне **Список ошибок**

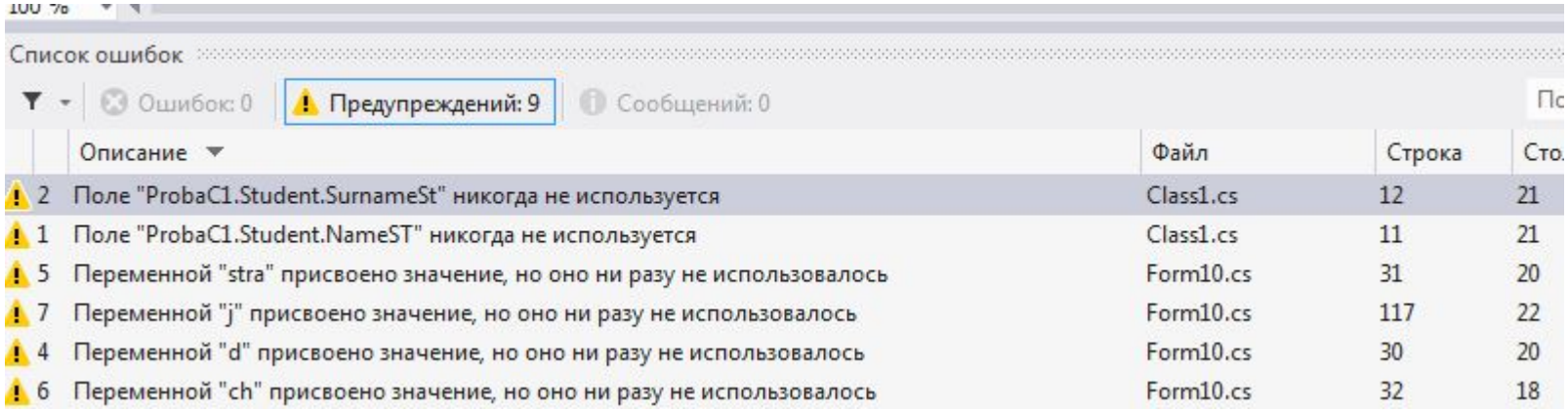






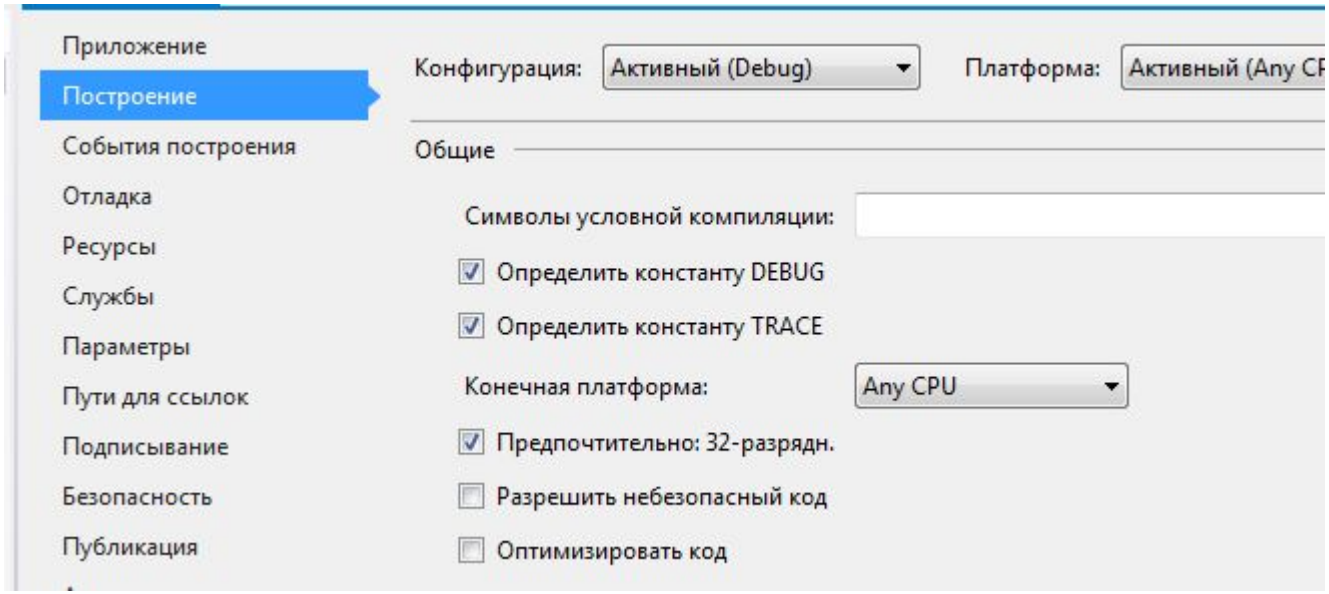
Ошибки и предупреждения

- Компилятор генерирует выполняемую программу (exe-файл) только в том случае, если в исходной программе (в тексте) нет ошибок.



Компиляция приложений

- Построение решений и проектов
  - Во время стандартного построения будут перестраиваться только те проекты в которые были внесены изменения с момента последней компиляции
  - Самая быстрая
- Перестройка решений и проектов
  - Добавление нового кода
  - Создание новой сборки
  - Ситуации, когда в процессе стандартного построения произошли ошибки
  - Перестраиваются все элементы проекта, увеличение времени
- Очистка решений ипроектов
  - Удаление вех выходных файлов
  - Получение более компактной версии приложения


Приложение
Построение
События построения
Отладка
Ресурсы
Службы
Параметры
Пути для ссылок
Подписывание
Безопасность
Публикация
Конфигурация:
Активный (Debug)
Платформа:
Активный (Any CP
Общие
Символы условной компиляции:
Определить константу DEBUG
Определить константу TRACE
Конечная платформа:
Any CPU
Предпочтительно: 32-разрядн.
Разрешить небезопасный код
Оптимизировать код