Язык программирования C#

Язык C# происходит от двух распространённых языков программирования: C и C++.

- от языка C унаследовал синтаксис, многие ключевые слова и операторы,\
- от C++ – усовершенствованную объектную модель.
- C# близко связан с Java

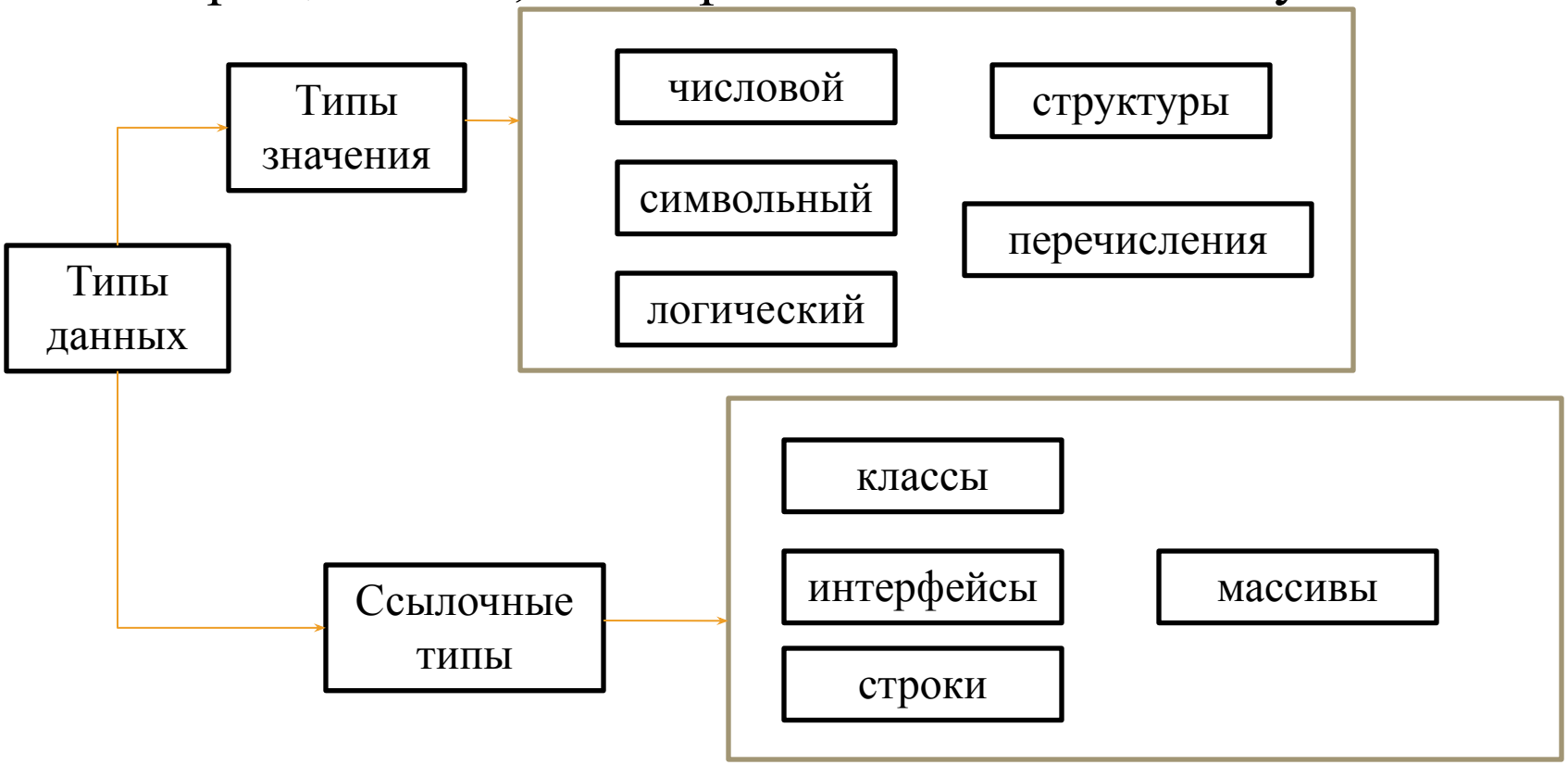
Особенности языка

- ✔ является объектно-ориентированным
- ✔ отсутствуют глобальные переменные и методы
- ✔ простейшие типы являются классами и поддерживают ряд базовых операций
- ✔ язык чувствителен к регистру символов
- ✔ при использовании методов требуется указание после идентификатора метода круглых скобок, даже если метод не имеет параметров
- ✔ переменные могут быть описаны в любом месте программы, при этом область видимости переменных зависит от места (блока программы) их описания
- ✔ все массивы могут изменять размеры (фактически путём создания нового массива)
- ✔ идентификаторы переменной и типа могут совпадать

Типы данных

В C# все общие типы данных делятся на два вида:

1) **типы значения** (содержат сами значения)
2) **ссылочные типы** (содержат ссылку на место в памяти, где значения можно найти), может содержать значение null, говорящее о том, что переменная ни на что не указывает



Целочисленные типы данных

- Byte
- Sbyte
- Char
- Short
- Ushort
- Int
- Uint
- Long
- Ulong

Вещественные типы

Float

Double

Десятичный тип

Decimal

Логический тип

Bool

Строковый

String

Методы и поля числовых типов данных:

Parse(s) – преобразует строкуs в число соответствующего типа

TryParse(s, out r) – преобразует строку sb число соответствующего типа и записывает результат в r

MinValue, MaxValue – возвращает минимальное или максимальное значение для заданного типа

Переменные

Для описания переменных используется конструкция:

<тип данных> <идентификатор 1>[=<значение идентификатора 1>] [, <идентификатор2>[=<значение идентификатора 2>] ...];

```
double d;
d = 255;
```

Если при описании переменной ей сразу присваивается значение, и данная строчка выполняется несколько раз (например, в цикле), то значение присваивается переменной при каждом выполнении строки.

Переменные могут быть типизированы неявно. В этом случае вместо типа данных указывается ключевое слово **var**.

```
var d=1.2;
var i=7;
var c='h';
```

В одной строке нельзя выполнить неявную типизацию двух и более переменных, т.е. следующая строка будет ошибочной

- `var a = 5, b = 7;`

Пространство имён

Пространство имён определяет область объявлений, в которой допускается хранить одно множество имён отдельно от другого. По существу, имена, объявленные в одном пространстве имён, не будут вступать в конфликт с аналогичными именами, объявленными в другой области.

Пространства имён помогают предотвратить конфликты имён, но не устранить их полностью. Такой конфликт может, в частности, произойти, когда одно и то же имя объявляется в двух разных пространствах имён и затем предпринимается попытка сделать видимыми оба пространства.

Для каждой программы на C# автоматически предоставляется используемое по умолчанию глобальное пространство имён.

Пространство имён объявляется с помощью ключевого слова namespace:

namespace <имя> { <элементы> }

Для подключения пространства имён используется директива using:

using <имя используемого пространства имен>;

Директиву using необходимо вводить в самом начале каждого файла исходного кода перед любыми другими объявлениями или же в начале тела пространства имен

Директива using может не использоваться вообще, однако в этом случае потребуется каждый раз использовать имя пространства имён при обращении к его членам.

Пример

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

    namespace HelloWorld
    {
     class Program
     {
      static void Main(string[] args)
      {
         Console.WriteLine("Hello, World!");
          Console.ReadKey();
      }
     }
    }
```

По умолчанию консольная программа на языке C# должна содержать как минимум один метод - метод Main, который является точкой входа в приложение:

```
static void Main(string[] args)
{


}
```

Ключевое слово static является модификатором. Далее идет тип возвращаемого значения. В данном случае ключевое слово void указывает на то, что

Далее идет название метода - *Main* и в скобках параметры - *string[] args*. И в фигурные скобки заключено тело метода - все действия, которые он выполняет. В данном случае метод Main пуст, он не содержит никаких операторов и по сути ничего не выполняет метод ничего не возвращает.

В C# есть ключевое слово **static**, которое можно применять к:

полям

- свойствам
- методам
- операторам
- событиям
- конструктору
- Классам

Статика подразумевает, что не нужно создавать экземпляр класса.

- Если весь класс является статическим:
  Нельзя создавать экземпляр класса, используя ключевое слово new.
- Не разрешается использовать не статические члены этого же класса.
- Он не поддерживает наследование.
- Невозможно перегрузить методы

Операторы и конструкции C#

Основным оператором присваивания является оператор =

<идентификатор> = <значение>;

при этом:

- <**идентификатор**> должен быть такого типа данных, который может вместить в себя присваиваемое значение, или который знает, как обработать присваиваемое значение;
- <**значение**> может быть числовой константой, переменной или результатом вычисления выражения.

Если требуется изменить значение некоторой переменной с учётом её предыдущего значения, то могут быть использованы операторы присваивания **+=, –=,*=,/=,%=,&=,|=,^=,<<=,>>=.** Данные операторы выполняют указанную перед символом =.

Выражение a *= b + c; равносильно выражению a = a*(b + c);

Приведение типов

При выполнении операторов присваивания (а также других операторов) в некоторых случаях может выполняться приведение (преобразование) типов.

Автоматическое преобразование возможно, если:

- оба типа совместимы;
- диапазон представления чисел целевого типа шире, чем у исходного типа.

Если требуется выполнить явное преобразование значения переменной или выражения к некоторому типу, то используется конструкция:

(<тип данных>)<преобразуемая величина>.

Операторные скобки {}

Операторные скобки {} применяются в случае, когда необходимо объединить несколько операторов в единый сложный оператор. Необходимость в таких действиях возникает, когда какой-либо оператор может выполнить только один другой оператор, а требуется выполнение нескольких.

Также операторные скобки применяются для обозначения начала и окончания различных блоков программы, например, тела функции.

Операторы инкремента и декремента

В простейшем случае операцию увеличения можно выполнить с помощью конструкции:

<переменная> = <переменная>+1

в C#:

<переменная>++; постфиксный инкремент

++<переменная>; префиксный инкремент

При выполнении одиночной операции никаких различий между ними нет.

При использовании операторов в выражениях:

- для префиксного инкремента сначала выполняется инкремент, а потом используется переменная в выражении;
- для постфиксного инкремента сначала используется переменная в выражении, а потом выполняется инкремент.

Аналогично операторам инкремента работают и операторы декремента:

<переменная>--; или --<переменная>;

Методы

Для вызова метода указывается его имя, после которого в скобках идут значения для его параметров (если метод принимает параметры.

название_метода (значения_для_параметров_метода);

Преимуществом методов является то, что их можно повторно и многократно вызывать в различных частях программы.

Метод может возвращать значение, какой-либо результат.

Методы с типом void не возвращают никакого значения

Если метод имеет любой другой тип, отличный от void, то такой метод обязан вернуть значение этого типа. Для этого применяется оператор return, после которого идет возвращаемое значение:

return возвращаемое значение;

Константы (литералы)

Для описания констант используется конструкция, аналогичная описанию переменных, но перед указанием типа данных указывается модификатор const

Класс Object

Данный класс является корнем иерархии всех типов и обладает рядом базовых методов, доступных для использования и часто переопределяемых в классах потомках:

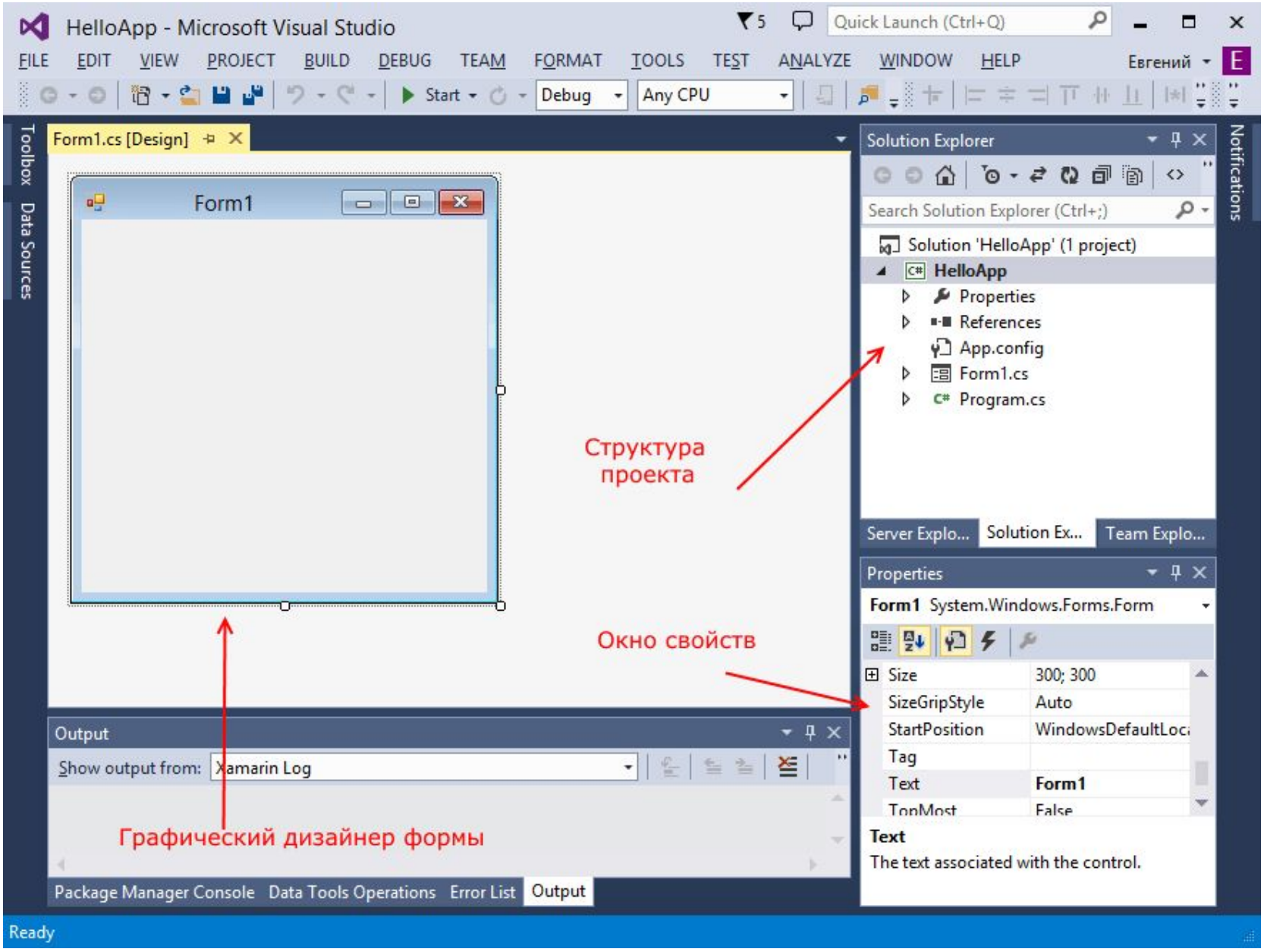
неявно считается базовым классом для всех остальных классов и типов, включая и типы значений. Иными словами, все остальные типы являются производными от object

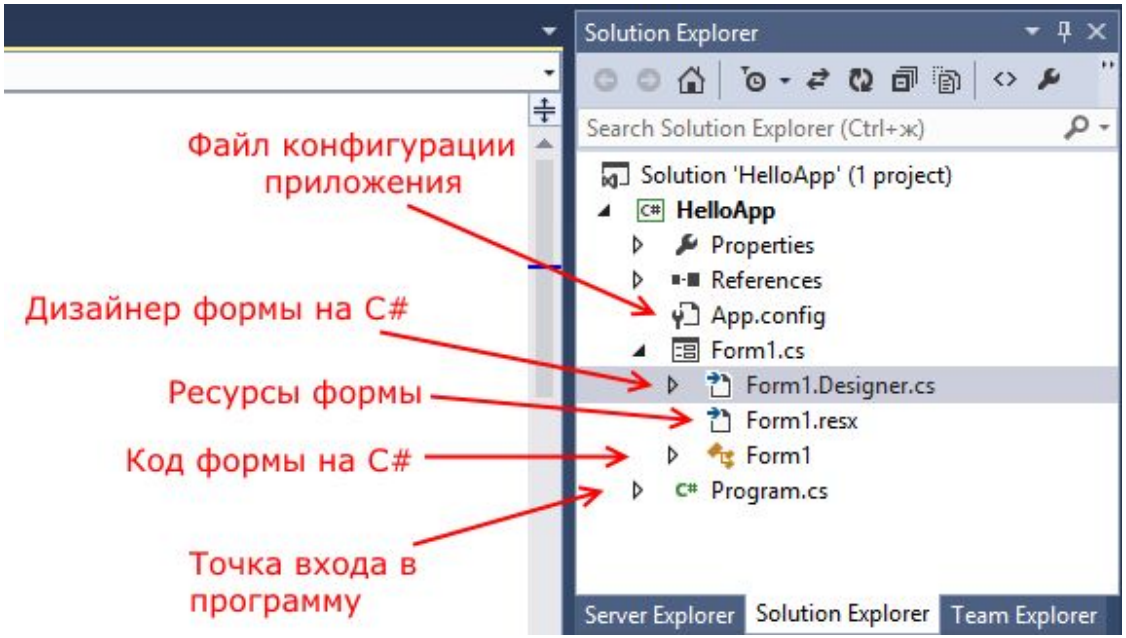
Методы:

ToString()

Метод ToString() возвращает символьную строку, содержащую описание того объекта, для которого он вызывается. Кроме того, метод ToString() автоматически вызывается при выводе содержимого объекта с помощью метода WriteLine(). Этот метод переопределяется во многих классах, что позволяет приспосабливать описание к конкретным типам объектов, создаваемых в этих классах

- **Finalize()**
- Назначение этого метода в C# примерно соответствует деструкторам C++, и он вызывается при сборке мусора для очистки ресурсов, занятых ссылочным объектом. Реализация Finalize() из Object на самом деле ничего не делает и игнорируется сборщиком мусора. Обычно переопределять Finalize() необходимо, если объект владеет неуправляемыми ресурсами, которые нужно освободить при его уничтожении. Сборщик мусора не может сделать это напрямую, потому что он знает только об управляемых ресурсах, поэтому полагается на финализацию, определенную вами.
- **GetType()**
- Этот метод возвращает экземпляр класса, унаследованный от System.Type. Этот объект может предоставить большой объем информации о классе, членом которого является ваш объект, включая базовый тип, методы, свойства и т.п. System.Type также представляет собой стартовую точку технологии рефлексии .NET.


HelloApp - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM FORMAT TOOLS TEST ANALYZE WINDOW HELP
Евгений
Start
Debug
Any CPU
Toolbox
Data Sources
Form1.cs [Design]
Form1
Структура проекта
Окно свойств
Графический дизайнер формы
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'HelloApp' (1 project)
HelloApp
Properties
References
App.config
Form1.cs
Program.cs
Server Explo...
Solution Ex...
Team Explo...
Notifications
Properties
Form1 System.Windows.Forms.Form
Size
300; 300
SizeGripStyle
Auto
StartPosition
Tag
Text
Form1
TopMost
False
Text
The text associated with the control.
Output
Show output from:
Xamarin Log
Package Manager Console
Data Tools Operations
Error List
Output
Ready



Form1.resx - хранит ресурсы формы

Form1.cs, который в структуре проекта называется просто Form1, содержит код или программную логику формы

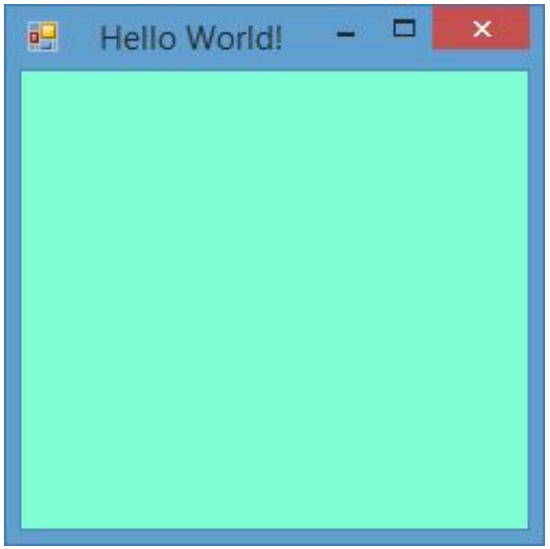
Формы

С помощью специального окна Properties (Свойства) справа Visual Studio предоставляет удобный интерфейс для управления свойствами элемента.

Свойство	Описание
Name	устанавливает имя формы - точнее имя класса, который наследуется от класса Form
BackColor	указывает на фоновый цвет формы. Щелкнув на это свойство, мы сможем выбрать тот цвет, который нам подходит из списка предложенных цветов или цветовой палитры
BackgroundImage	указывает на фоновое изображение формы
Font	задает шрифт для всей формы и всех помещенных на нее элементов управления. Однако, задав у элементов формы свой шрифт, мы можем тем самым переопределить его
Icon	задает иконку формы

Программная настройка свойств

```
namespace HelloApp
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            Text = "Hello World!";
            this.BackColor = Color.Aquamarine;
            this.Width = 250;
            this.Height = 250;
        }
    }
}
```



Элементы управления представляют собой визуальные классы, которые получают введенные пользователем данные и могут инициировать различные события. Все элементы управления наследуются от класса Control и поэтому имеют ряд общих свойств.

Название	Описание
Anchor:	Определяет, как элемент будет растягиваться
Font	Устанавливает шрифт текста для элемента
BackColor	Определяет фоновый цвет элемента
Tag	Позволяет сохранять значение, ассоциированное с этим элементом управления
Cursor	Представляет, как будет отображаться курсор мыши при наведении на элемент
Name	Имя элемента управления
BackgroundImage	Определяет фоновое изображение элемента

Кнопка

Наиболее используемый элемент управления. При нажатии на кнопку на форме открывается код обработчика события Click, который будет выполняться при нажатии:

```
private void button1_Click(object sender, EventArgs e)
{
   MessageBox.Show("Hello World");
}
```

Чтобы управлять внешним отображением кнопки, можно использовать свойство **FlatStyle**. Оно может принимать следующие значения:

Flat - Кнопка имеет плоский вид

Popup - Кнопка приобретает объемный вид при наведении на нее указателя, в иных случаях она имеет плоский вид

Standard - Кнопка имеет объемный вид (используется по умолчанию)

System - Вид кнопки зависит от операционной системы

Для кнопки можно:

- задавать изображение с помощью свойства BackgroundImage

-управлять размещением текста и изображения на кнопки с помощью свойства TextImageRelation:

а) изображение располагается над текстом – *ImageAboveText*

б) наложение текста на изображение – *Overlay*

в) изображение располагается перед текстом - *ImageBeforeText*

В окне Свойств есть поле Image, где можно устанавливать изображение для кнопки.

TextBox

Для ввода и редактирования текста предназначены текстовые поля - элемент TextBox. Так же как и у элемента Label текст элемента TextBox можно установить или получить с помощью свойства Text.

По умолчанию при переносе элемента с панели инструментов создается однострочное текстовое поле. Для отображения больших объемов информации в текстовом поле нужно использовать его свойства Multiline и ScrollBars. При установке для свойства Multiline значения true, все избыточные символы, которые выходят за границы поля, будут переноситься на новую строку.

Кроме того, можно сделать прокрутку текстового поля, установив для его свойства **ScrollBars** одно из значений:

None: без прокруток (по умолчанию)

Horizontal: создает горизонтальную прокрутку при длине строки, превышающей ширину текстового поля

Vertical: создает вертикальную прокрутку, если строки не помещаются в текстовом поле

Both: создает вертикальную и горизонтальную прокрутку.

Элемент TextBox обладает достаточными возможностями для создания автозаполняемого поля. Для этого нам надо привязать свойство **AutoCompleteCustomSource** элемента TextBox к некоторой коллекции, из которой берутся данные для заполнения поля

ListBox

Элемент ListBox представляет собой простой список. Ключевым свойством этого элемента является свойство Items, которое как раз и хранит набор всех элементов списка.

Элементы в список могут добавляться как во время разработки, так и программным способом.

Все элементы списка входят в свойство *Items*, которое представляет собой коллекцию: для добавления нового элемента в эту коллекцию, а значит и в список, надо использовать метод Add, например: listBox1.Items.Add("Новый элемент"), причем каждый новый элемент добавляется в конец списка.

Вставка элементов

```
string[] fruits = { “orange", “kivi", “banana"};
listBox1.Items.AddRange(fruits);
```

Удаление элемента - метод Remove:

```
listBox1.Items.Remove(“kivi");
```

Удаление элемента по его индексу в списке -метод RemoveAt:

```
listBox1.Items.RemoveAt(1);
```

Очистка списка:

```
listBox1.Items.Clear();
```

Первый элемент списка:

```
string firstElement = listBox1.Items[0];
```

Количество элементов в списке – метод Count:

```
int n = listBox1.Items.Count();
```

Свойства элемента ListBox для выделения элементов:

- *SelectedIndex*: возвращает или устанавливает номер выделенного элемента списка. Если выделенные элементы отсутствуют, тогда свойство имеет значение -1;
- *SelectedIndices*: возвращает или устанавливает коллекцию выделенных элементов в виде набора их индексов
- *SelectedItem*: возвращает или устанавливает текст выделенного элемента
- *SelectedItems*: возвращает или устанавливает выделенные элементы в виде коллекции

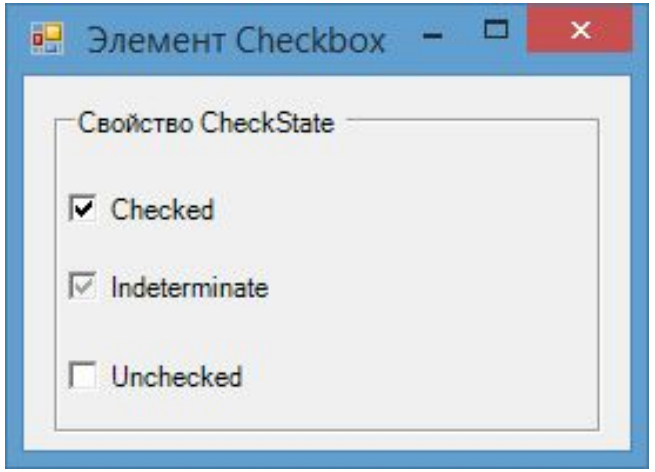
CheckBox

Элемент CheckBox (флажок) предназначен для установки одного из двух значений: отмечен или не отмечен.

Чтобы отметить флажок, надо установить у его свойства Checked значение true.

Свойство AutoCheck - если оно имеет значение false, то нельзя изменять состояние флажка. По умолчанию - true.

```
private void checkBox_CheckedChanged(object sender, EventArgs e)
{
   CheckBox checkBox = (CheckBox)sender; // приводим отправителя к
    элементу типа CheckBox
   if (checkBox.Checked == true)
   {
      MessageBox.Show(" Да " + checkBox.Text );
   }
}
```



Radiobutton

Переключатели располагаются группами, и включение одного переключателя означает отключение всех остальных.

Чтобы установить у переключателя включенное состояние, надо присвоить его свойству Checked значение true.

Для создания группы переключателей, из которых можно бы было выбирать, надо поместить несколько переключателей в какой-нибудь контейнер, например, в элементы GroupBox или Panel. Переключатели. находящиеся в разных контейнерах, будут относиться к разным группам.

Переключение переключателей в группе можно производить, обрабатывая событие CheckedChanged. Связав каждый переключатель группы с одним обработчиком данного события, можно получитьтот переключатель, который выбран

```
private void radioButton_CheckedChanged(object sender, EventArgs e)
{
      RadioButton radioButton = (RadioButton)sender; // тип Radiobutton
   if (radioButton.Checked)
   {
      MessageBox.Show("Вы выбрали " + radioButton.Text);
   }
}
```

PictureBox

- предназначен для показа изображений. Он позволяет отобразить файлы в формате bmp, jpg, gif, а также метафайлы изображений и иконки

Image:	устанавливает объект типа Image
ImageLocation:	устанавливает путь к изображению на диске или в интернете
InitialImage:	некоторое начальное изображение, которое будет отображаться во время загрузки главного изображения, которое хранится в свойстве Image
ErrorImage:	изображение, которое отображается, если основное изображение не удалось загрузить в PictureBox
SizeMode	Размер изображения: - *Normal*: изображение позиционируется в левом верхнем углу PictureBox, и размер изображения не изменяется. Если PictureBox больше размеров изображения, то по справа и снизу появляются пустоты, если меньше - то изображение обрезается *StretchImage*: изображение растягивается или сжимается таким образом, чтобы вместиться по всей ширине и высоте элемента PictureBox AutoSize: элемент PictureBox автоматически растягивается, подстраиваясь под *размеры* изображения *CenterImage*: если PictureBox меньше изображения, то изображение обрезается по краям и выводится только его центральная часть. Если же PictureBox больше изображения, то оно позиционируется по центру. *Zoom*: изоражение подстраивается под размеры PictureBox, сохраняя при

ToolStrip

представляет панель инструментов. Каждый отдельный элемент на этой панели является объектом ToolStripItem.

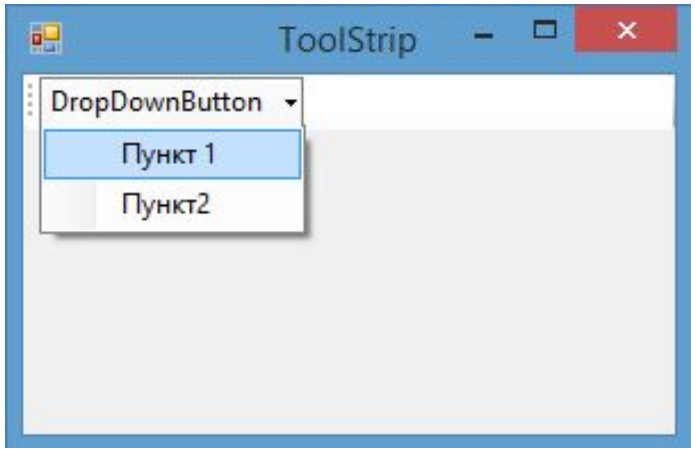
- Dock: прикрепляет панель инструментов к одной из сторон формы
- LayoutStyle: задает ориентацию панели на форме (горизонтальная, вертикальная, табличная)
- ShowItemToolTips: указывает, будут ли отображаться всплывающие подсказки для отдельных элементов панели инструментов
- Stretch: позволяет растянуть панель по всей длине контейнера

В зависимости от значения свойства LayoutStyle панель инструментов может располагаться по горизонтали, или в табличном виде:

- HorizontalStackWithOverflow: расположение по горизонтали с переполнением - если длина панели превышает длину контейнера, то новые элементы, выходящие за границы контейнера, не отображаются, то есть панель переполняется элементами
- StackWithOverflow: элементы располагаются автоматически с переполнением
- VerticalStackWithOverflow: элементы располагаются вертикально с переполнением
- Flow: элементы располагаются автоматически, но без переполнения - если длина панели меньше длины контейнера, то выходящие за границы элементы переносятся, а панель инструментов растягивается, чтобы вместить все элементы
- Table: элементы позиционируются в виде таблицы

Панель ToolStrip может содержать объекты следующих классов:

- ToolStripLabel: текстовая метка на панели инструментов, представляет функциональность элементов Label и LinkLabel
- ToolStripButton: аналогичен элементу Button.Также имеет событие Click, с помощью которого можно обработать нажатие пользователя на кнопку
- ToolStripSeparator: визуальный разделитель между другими элементами на панели инструментов
- ToolStripToolStripComboBox: подобен стандартному элементу ComboBox
- ToolStripTextBox: аналогичен текстовому полю TextBox
- ToolStripProgressBar: индикатор прогресса, как и элемент ProgressBar
- ToolStripDropDownButton: представляет кнопку, по нажатию на которую открывается выпадающее меню, К каждому элементу выпадающего меню дополнительно можно прикрепить обработчик нажатия и обработать клик по этим пунктам меню
- ToolStripSplitButton: объединяет функциональность ToolStripDropDownButton и ToolStripButton



Сериализация

представляет процесс преобразования какого-либо объекта в поток байтов. После преобразования мы можем этот поток байтов или записать на диск или сохранить его временно в памяти. А при необходимости можно выполнить обратный процесс - *десериализацию*, то есть получить из потока байтов ранее сохраненный объект.

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом *Serializable*

```
[Serializable]
class Person
{
    public string Name { get; set; }
    public int Year { get; set; }

    public Person(string name, int year)
    {
        Name = name;
        Year = year;
    }
}
```

При отстутствии данного атрибута объект Person не сможет быть сериализован, и при попытке сериализации будет выброшено исключение SerializationException

Сериализация применяется к свойствам и полям класса. Если для какого-либо поля класса не нужна сериализация, то его помечают атрибутом NonSerialized:

```
[Serializable]
class Person
{
   public string Name { get; set; }
   public int Year { get; set; }

   [NonSerialized]
   public string accNumber;

   public Person(string name, int year, string acc)
   {
      Name = name;
      Year = year;
      accNumber = acc;
   }
}
```