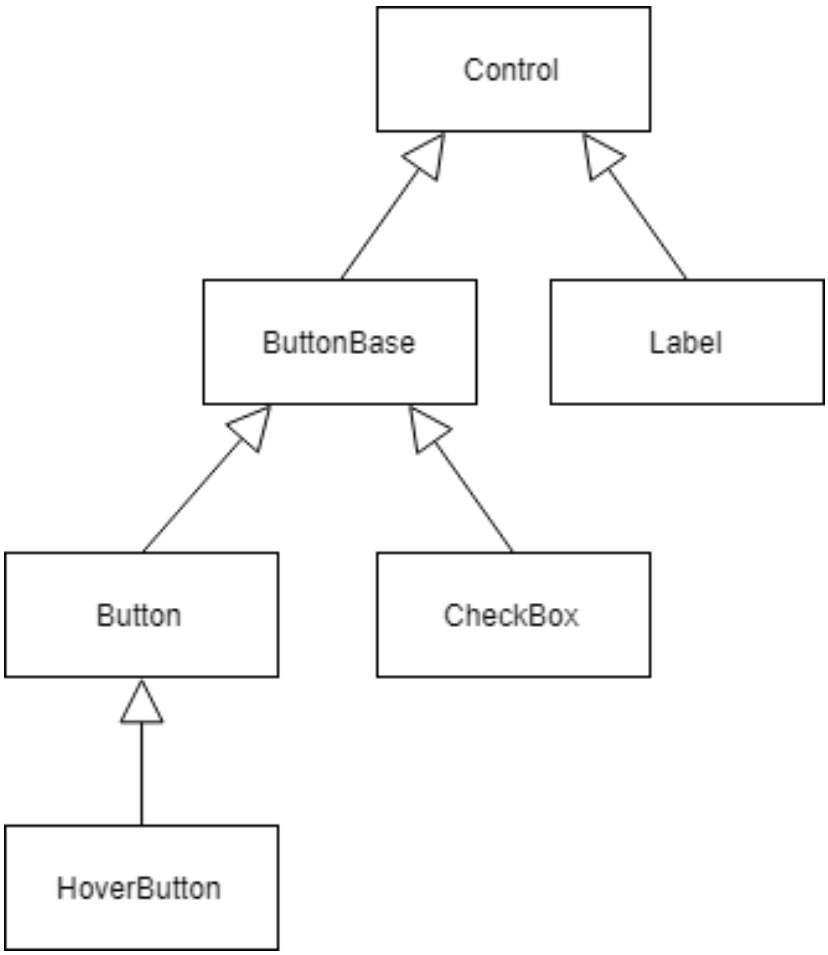
# Лабораторная работа №3: Создание пользовательских элементов управления

**Цель работы:**

- изучить приемы создания пользовательских элементов управления с использованием технологий Windows Forms и WPF;
- ознакомиться с примерами использования наследования, полиморфизма и композиции объектов на практике.

**Создание кнопки с анимацией (hover-эффект) в Windows Forms**

Иерархия элементов управления в Windows Forms представлена следующей диаграммой (данная диаграмма является неполной, на ней представлены лишь некоторые элементы управления, тем не менее, она иллюстрирует основные взаимосвязи между классами).



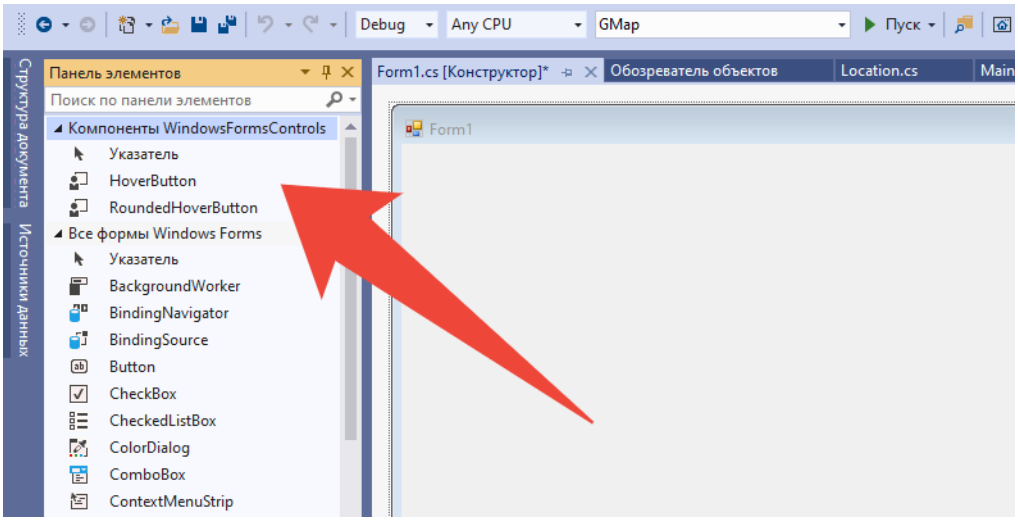

Все элементы управления являются наследниками базового класса `Control`, который реализует общую логику всех элементов управления (отображение, взаимодействие с пользователем и др.). Подклассы `Button`, `CheckBox` и `Label` представляют собой конкретные элементы управления – кнопка, флажок, заголовок. Промежуточный класс `ButtonBase` реализует общую логику для схожих компонентов `Button` и `CheckBox`.

Как правило, при создании пользовательского элемента управления расширяют один из уже существующих компонентов. Создайте проект «Приложение Windows Forms (.NET Framework)». Добавьте новый класс, расширяющий функционал стандартного класса `Button`:

```
class HoverButton : Button
{


}
```

После сборки проекта (Сборка – Собрать решение или F6) компонент будет доступен в панели элементов. Расположите компонент на окне с помощью визуального редактора:



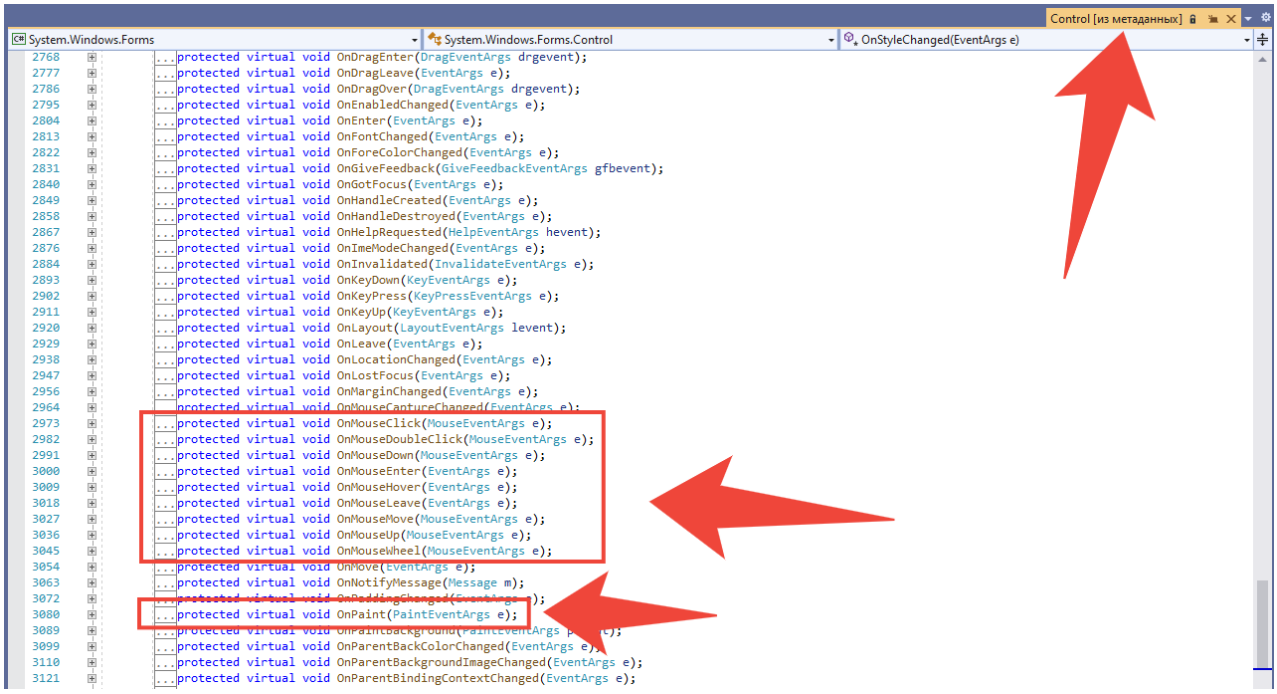

В конструкторе можно задать свойства по умолчанию, например, цвет текста и шрифт.

```
public HoverButton() : base()
{
        ForeColor = Color.White;
        Font = new Font("Microsoft YaHei UI",
            20.25F,
            FontStyle.Bold,
            GraphicsUnit.Point,
            0);
}
```

Hover-эффект представляет собой реакцию на наведение курсора мыши в область кнопки. Таким образом, необходимо изменить поведение кнопки при наведении на нее курсора.

Для решения подобного рода задач хорошо подходит **полиморфизм.** Одной из разновидностей полиморфизма является переопределение виртуальных методов.

В базовом классе Control определены виртуальные методы для управления компонентом, в том числе для управления мышью и для отрисовки:



Переопределим метод `OnPaint`, отвечающий за отрисовку. Для этого добавим в класс `HoverButton` метод `OnPaint` с такой же сигнатурой, но с модификатором `override`:

```
protected override void OnPaint(PaintEventArgs pe)
{
    base.OnPaint(pe);
}
```

Сейчас метод вызывает только базовую реализацию. Для отрисовки кнопки определенного цвета необходимо добавить следующий код:

```
private Color color = Color.SkyBlue;
...
protected override void OnPaint(PaintEventArgs pe)
{
    base.OnPaint(pe);
    // отрисовка прямоугольника
    pe.Graphics.FillRectangle(new SolidBrush(color), ClientRectangle);
    // отрисовка текста
    pe.Graphics.DrawString(Text, Font, new SolidBrush(ForeColor), ClientRectangle);
}
```

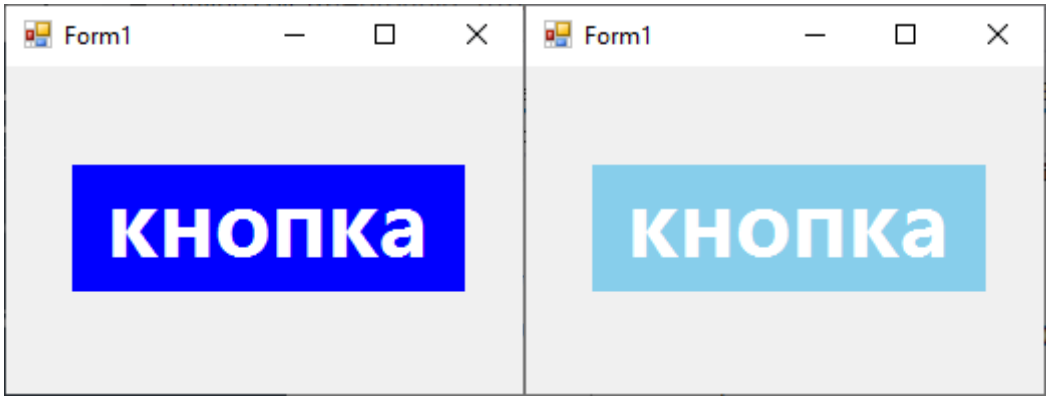
Таким образом, можно управлять цветом кнопки, меняя значение поля `color`.

Один из вариантов hover-эффекта – изменение цвета кнопки при наведении мыши. В базовом классе `Control` есть методы, определяющие нахождение курсора мыши в области элемента управления: `OnMouseEnter` (курсор наведен на элемент управления) и `OnMouseLeave` (курсор перемещен в другую область). При наведении курсора на кнопку сделаем её более темной, а при перемещении курсора в другую область вернем исходный цвет:

```
protected override void OnMouseEnter(EventArgs e)
{
    base.OnMouseEnter(e);
    color = Color.Blue;
}
```

```
protected override void OnMouseLeave(EventArgs e)
{
    base.OnMouseLeave(e);
    color = Color.SkyBlue;
}
```

Во время выполнения это будет выглядеть так:




Для достижения других эффектов можно аналогичным образом дополнить реализацию других методов, например `OnMouseDown` (переключение кнопки вниз), `OnMouseUp` (переключение кнопки вверх), `OnMouseWheel` (прокрутка колесика). Для изменения формы кнопки можно переопределить метод `OnResize`, отвечающий за расчёт границ элемента управления. Например, можно сделать кнопку овальной:

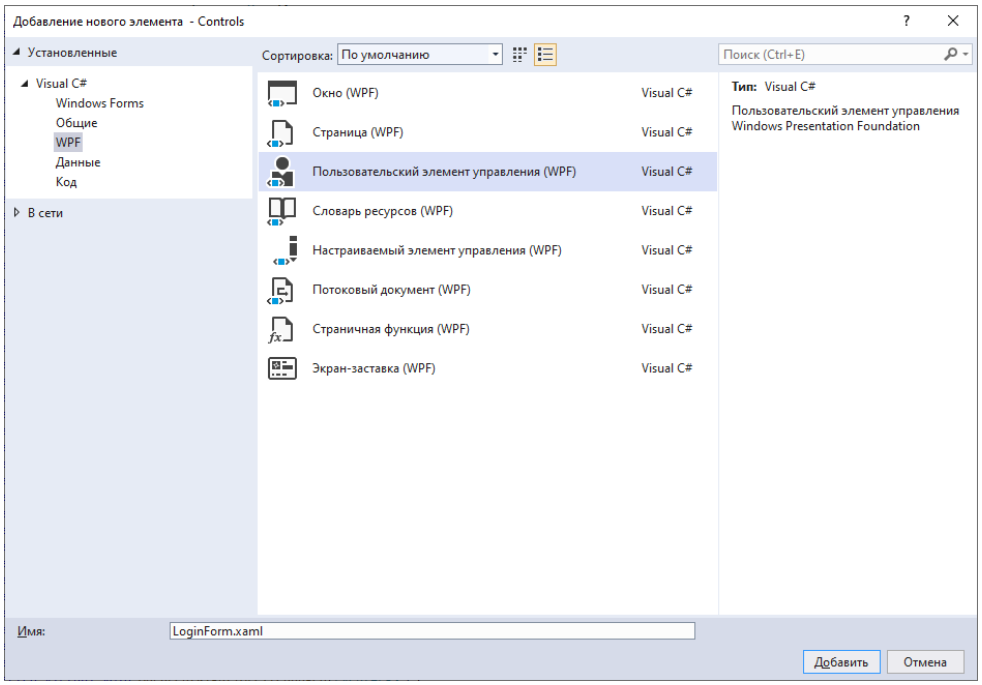
```
protected override void OnResize(EventArgs e)
{
    base.OnResize(e);
    GraphicsPath graphicsPath = new GraphicsPath();
    graphicsPath.AddEllipse(new Rectangle(0, 0, Width - 1, Height - 1));
    Region = new Region(graphicsPath);
}
```

Во время выполнения это будет выглядеть так:


**Создание формы авторизации на WPF:**

Пользовательские элементы управления на WPF представляют собой, как правило, композицию нескольких стандартных элементов управления, приведенных к единому стилю.

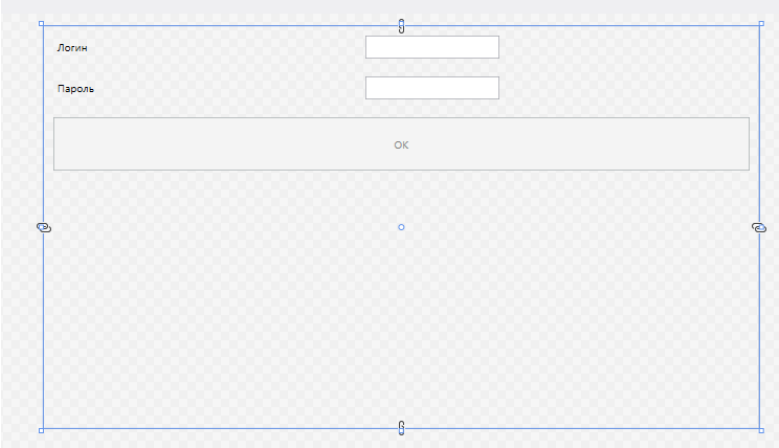
Создайте проект «Приложение WPF (.NET Framework)». В контекстном меню проекта выберите «Добавить – Создать элемент», а затем, в диалоговом окне, «Пользовательский элемент управления». Введите название и нажмите «Добавить».




После этого в обозревателе решений будут доступны 2 файла: .xaml-файл для создания макета и .cs-файл для определения логики пользовательского элемента управления.

Для создания разметки воспользуйтесь визуальным конструктором или отредактируйте .xaml-файл вручную. В качестве корневого элемента можно использовать **Grid** или **StackPanel**.

Добавьте в макет поле ввода логина **TextBox** и поле ввода пароля **PasswordBox**, а так же другие вспомогательные элементы по необходимости: кнопку подтверждения **Button** и заголовки **Label.** Один из вариантов макета:




Для использования элементов макета в .cs-файле необходимо задать им имена: например, для поля ввода логина **Name=«Login»**.

После сборки проекта элемент управления также будет доступен в панели элементов.

**Передача параметров элемента управления из разметки:**

Некоторые параметры элемента управления удобно задавать из разметки, например, визуальные параметры (цвет фон, параметры шрифта и др.) или значения по умолчанию. В данном случае, требуется задавать значение по умолчанию в поле ввода логина.

Данный функционал реализуется с помощью служебного класса `DependencyProperty`. В .cs-файле создайте объект класса `DependencyProperty`:

```
public static readonly DependencyProperty LoginProperty = DependencyProperty.Register(
            "Login", // имя параметра в разметке
            typeof(string), // тип данных параметра
            typeof(LoginForm), // тип данных элемента управления
            new PropertyMetadata(string.Empty, LoginChanged)); // метаданные - значение параметра по умолчанию и обработчик изменения параметра
```

Для удобства использования значения параметра создайте свойство, делегирующее работу с объектом `LoginProperty`:

```
public string Login
{
       get { return (string)GetValue(LoginProperty); }
       set { SetValue(LoginProperty, value); }
}
```

В обработчике изменения параметра `LoginChanged` необходимо добавить код, который реализует добавление текста в поле ввода логина:

```
private static void LoginChanged(DependencyObject obj, DependencyPropertyChangedEventArgs args)
{
       var loginForm = obj as LoginForm;
       loginForm.LoginTextBox.Text = loginForm.Login;
}
```

После этого можно передать значение по умолчанию через разметку. Для этого в разметку поля ввода логина необходимо добавить **Login=«admin»**.

Также необходимо реализовать логику работы элемента управления, например, осуществлять валидацию введенных значений логина и пароля, а по нажатию на кнопку подтверждения

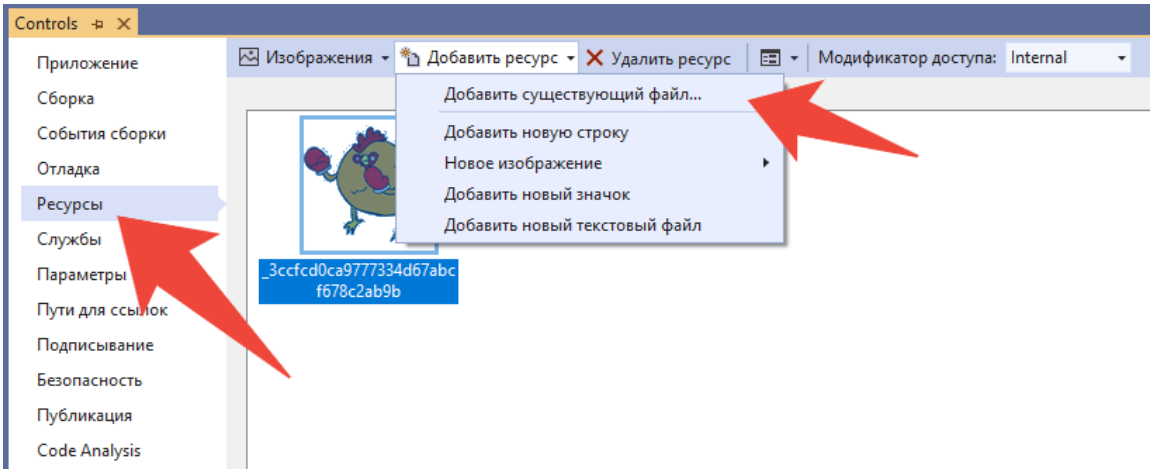
осуществлять некоторое действие. На каждое действие необходимо создать и назначить свой обработчик:

```
LoginTextBox.TextChanged += LoginTextBox_TextChanged; // назначение обработчика изменения логина
PasswordBox.PasswordChanged += PasswordBox_PasswordChanged; // назначение обработчика изменения пароля
OkButton.Click += OkButton_Click; // назначение обработчика нажатия кнопки подтверждения
```

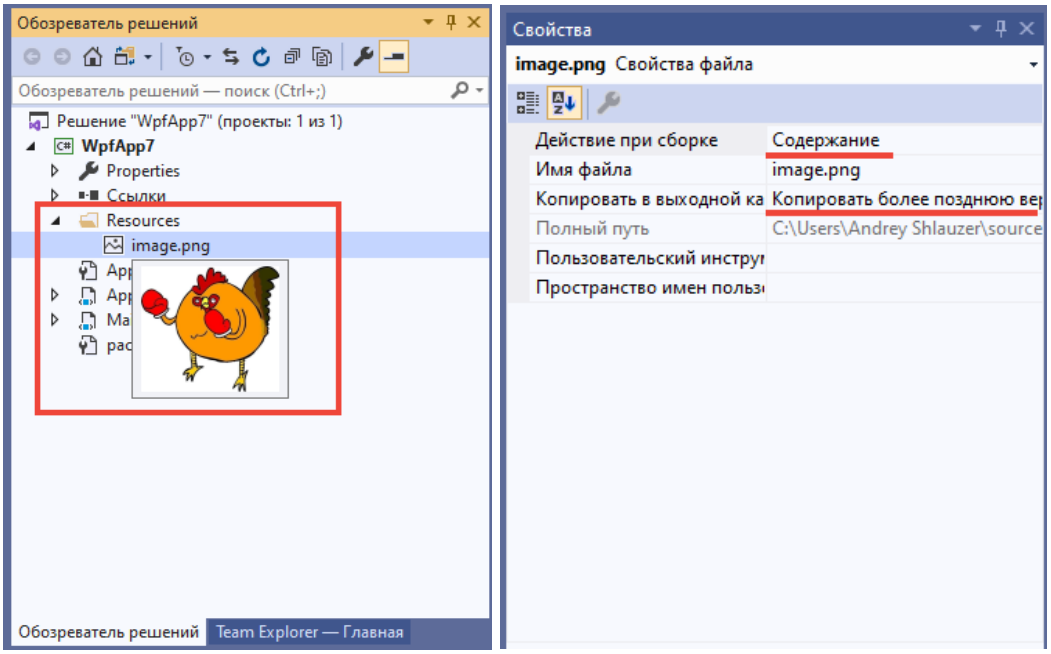
```
private void OkButton_Click(object sender, RoutedEventArgs e)
{
	// TODO : need action
}
private void PasswordBox_PasswordChanged(object sender, RoutedEventArgs e)
{
	// TODO : need action
}
private void LoginTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
	// TODO : need action
}
private void validateData()
{
	// длина логина должна быть не менее 4 символов,
	// а длина пароля - не менее 8 символов
	bool isDataValid = Login.Length >= 4 && Password.Length >= 8;
	if (isDataValid)
		OkButton.IsEnabled = true;
	else
		OkButton.IsEnabled = false;
}
```

**Работа с изображениями:**

В приложениях Windows Forms и WPF с изображением удобно работать как с ресурсом. Чтобы загрузить изображение, перейдите в свойства проекта (вкладка «Ресурсы») и добавьте файл с изображением в ресурсы.



После добавления изображения в ресурсы оно появится в обозревателе решений. Выделите его и установите в панели свойств такие же значения атрибутов, как на скриншоте (это нужно для того, чтобы изображение копировалось в выходную директорию с .exe-файлом в процессе сборки).




Затем можно обращаться к изображению из кода. В стандартном классе `Resources` будет сгенерировано одноименное поле для получения картинки. Пример получения и отрисовки картинки в Windows Forms может выглядеть так:

```
protected override void OnPaint(PaintEventArgs pe)
{
    base.OnPaint(pe);
    // получение картинки из ресурсов
    Bitmap bitmap = new Bitmap(Resources.image);
    // отрисовка картинки в точке (0,0)
    pe.Graphics.DrawImage(bitmap, 0, 0);
}
```

Если поместить этот код в классе `HoverButton`, то картинка будет установлена в качестве фона.

В WPF достаточно прописать путь к картинке по следующему шаблону:

```
image1.Source = new BitmapImage(new Uri("pack://application:,,,/Resources/image.jpg"));
```

В данном примере картинка будет отображена в стандартном компоненте `Image` (`image1` – имя конкретного компонента в разметке).

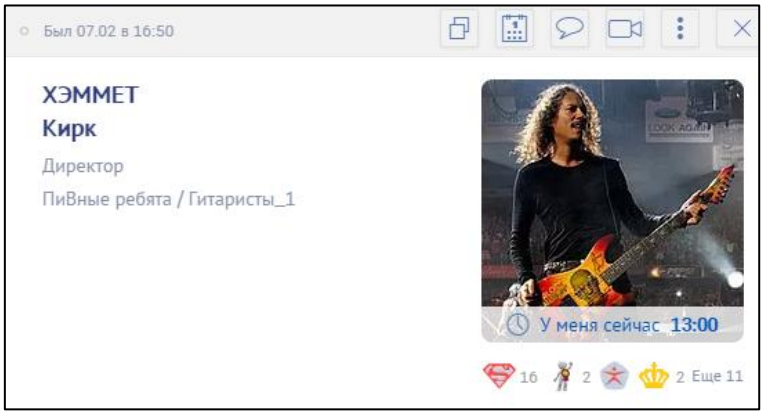
**Задание:**

Разработать демонстрационное приложение с пользовательскими элементами управления:

1. Галерея (компонент для просмотра изображений) в Windows Forms (на основе класса `HoverButton`).
   a) галерея с hover-эффектом. При наведении курсора на изображение появляется её подпись и другие декоративные элементы. В качестве примера можно использовать один из представленных эффектов по ссылке (анимацией можно пренебречь). По клику на изображение загружается следующая картинка.
   b) аналогичная галерея круглой формы (должна быть наследником класса из пункта a).

   Галерея должна быть универсальной и совместимой с любым набором изображений.

2. Карточка сотрудника на WPF. Карточка сотрудника отображает основные сведения о сотруднике (имя, отдел, должность, фотография, статус и другие). При наведении на кнопки должна появляться всплывающая подсказка (ToolTip). Один из примеров реализации:




**Контрольные вопросы:**

1) Опишите принципы создания пользовательского интерфейса в проектах WPF и Windows Forms, назовите основные отличия.
2) Приведите пример наследование и полиморфизма при создании пользовательских компонентов в Windows Forms.
3) Дайте определение модификаторам `virtual` и `override` и приведите пример их использования.
4) Как осуществляется добавление картинки в проект?

**Список литературы:**


1) Герберт Шилдт "C# 4.0: полное руководство"
2) Эндрю Троелсен "Язык программирования C# 5.0 и платформа .NET 4.5"
3) Руководство по программированию на C# https://docs.microsoft.com/ruru/dotnet/csharp/programming-guide/index
4) Полное руководство по языку программирования C# 7.0 и платформе .NET 4.7 https://metanit.com/sharp/tutorial/
5) Руководство по WPF https://metanit.com/sharp/wpf/
6) C# 5.0 и платформа .NET 4.5 http://professorweb.ru/my/csharp/charp_theory/level1/infocsharp.php
7) https://github.com/Microsoft/WPF-Samples