Лабораторная работа №1

«Программы Windows Forms с элементами управления»

1 Форма, кнопка, метка и диалоговое окно

После инсталляции системы программирования Visual Studio, включающей в себя Visual C#, при ее запуске увидим начальный пользовательский интерфейс, похожий на рис. 1.

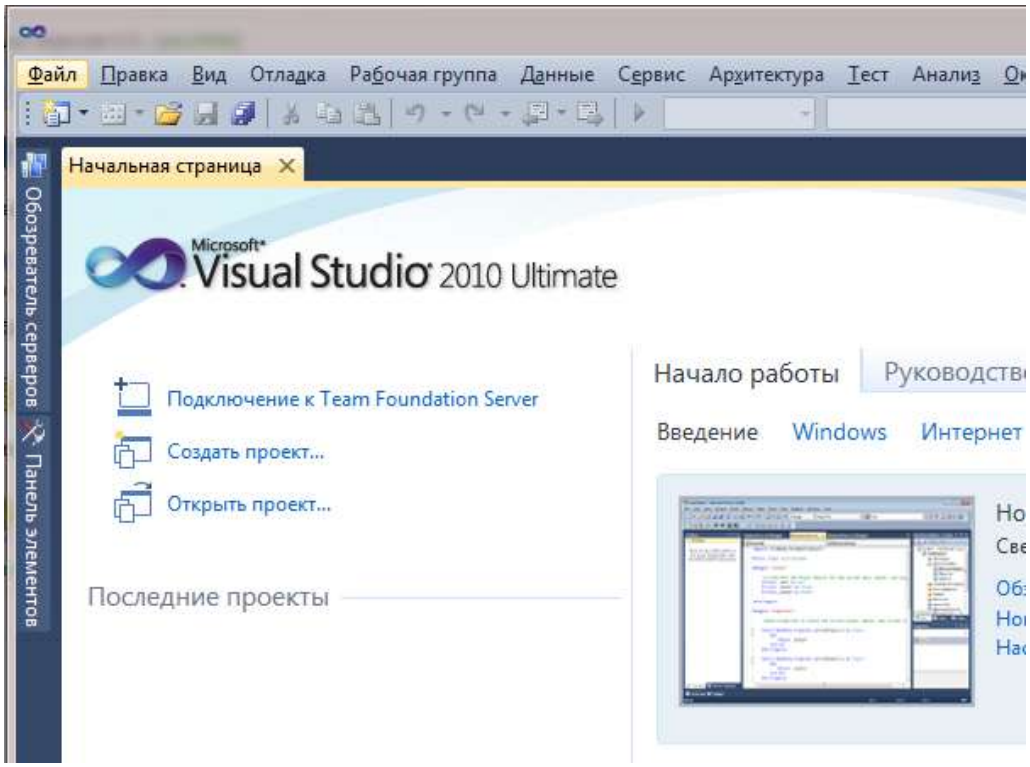


Рисунок 1 – Фрагмент стартовой страницы системы Visual Studio

Чтобы запрограммировать какую-либо задачу, необходимо в пункте меню **Файл** выполнить команды **Создать | Проект.** В появившемся окне **Создать проект** в левой колонке находится список инсталлированных шаблонов **(Установленные шаблоны).** Среди них – шаблоны языков программирования, встроенных в Visual Studio, в том числе Visual Basic, Visual C#, Visual C++, Visual F# и др. Нам нужен язык Visual C#. В средней колонке выберем шаблон **Visual C# | Приложение Windows Forms** и щелкнем на кнопке **ОК**. В результате увидим окно, представленное на рис. 2.

В этом окне изображена *экранная форма* – **Form1,** в которой программисты располагают различные компоненты графического интерфейса пользователя или, как их иначе называют, элементы управления. Это поля для ввода текста **TextBox,** командные кнопки **Button,** строчки текста в форме – метки **Label,** которые не могут быть отредактированы пользователем, и прочие элементы управления. Причем здесь используется самое современное так называемое *визуальное программирование,* предполагающее простое перетаскивание мышью из панели элементов **Панель элементов (ToolBox),** где расположены всевозможные элементы управления, в форму. Таким образом, стараются свести к минимуму непосредственное написание программного кода.

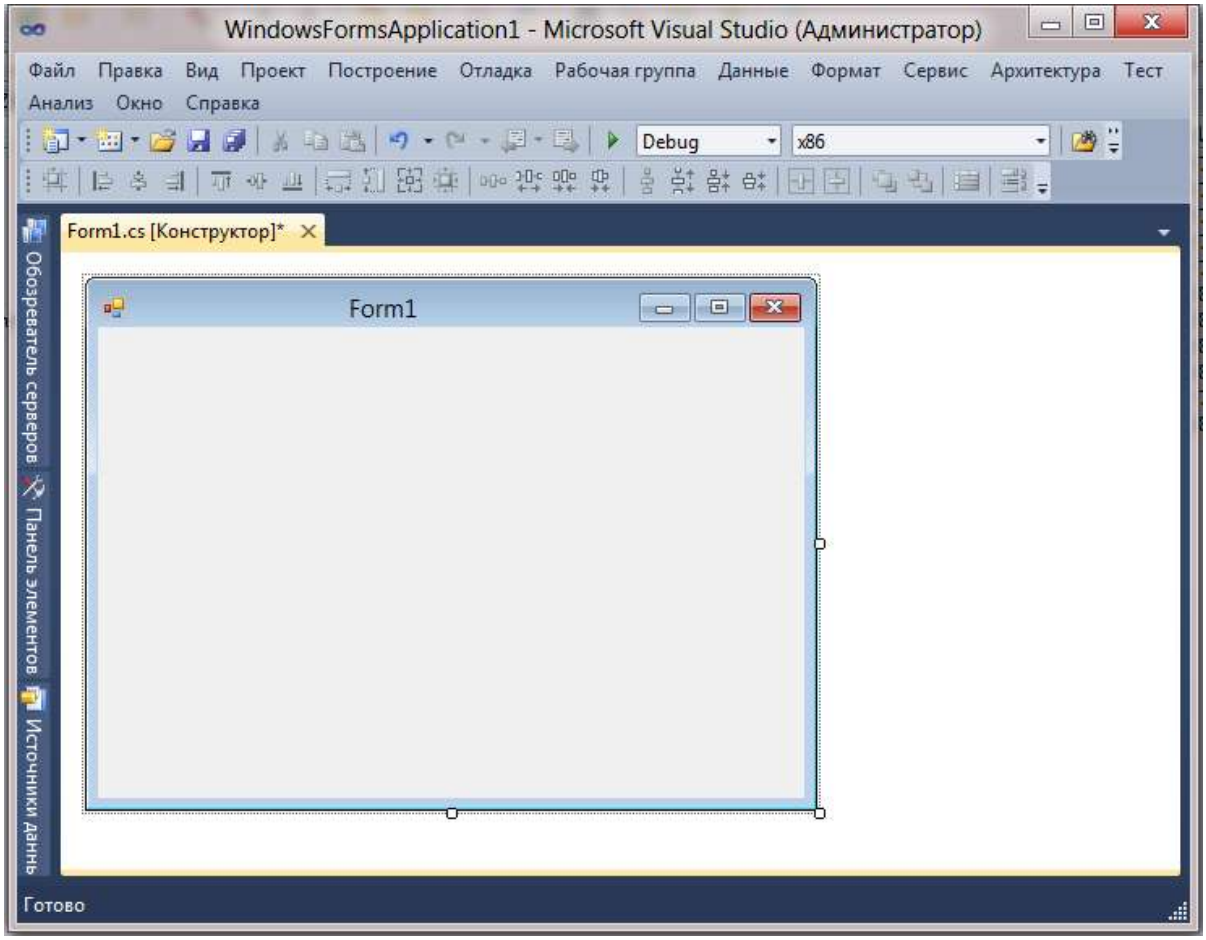


Рисунок 2 – Окно для проектирования пользовательского интерфейса

Ваша первая программа будет отображать такую экранную форму, в которой будет что-либо написано, например "Microsoft Visual C#", также в форме будет расположена командная кнопка с надписью "Нажми меня". При нажатии кнопки появится диалоговое окно с сообщением "Всем привет!"

Написать такую программку просто, но вначале следует объяснить современный объектно-ориентированный подход к программированию. Подход заключается в том, что в программе все, что может быть названо именем существительным, называют *объектом.* Так, в нашей программе мы имеем четыре объекта: форму **Form1**, надпись на форме **Label1**, кнопку **Button1** и диалоговое окно **MessageBox** с текстом “Всем привет!”. Итак, добавьте метку и кнопку на форму примерно так, как показано на рис. 3.

Любой такой объект можно создавать самому, а можно пользоваться готовыми объектами. В данной задаче мы пользуемся готовыми визуальными объектами, которые можно перетаскивать мышью из панели элементов управления Toolbox. В этой задаче нам нужно знать, что каждый объект имеет свойства (properties). Например, свойствами кнопки являются (рис. 4): имя кнопки **(Name) button1,** надпись на кнопке **(Text),** расположение кнопки **(Location)** в системе координат формы x, y, размер кнопки **Size** и т. д. Свойств много, их можно увидеть, если щелкнуть правой кнопкой мыши в пределах формы и выбрать в контекстном меню команду Свойства (Properties), при этом появится панель свойств Свойства.

Указывая мышью на все другие элементы управления в форме, можно просмотреть их свойства: формы **Form1** и надписи в форме – метки **label1.**

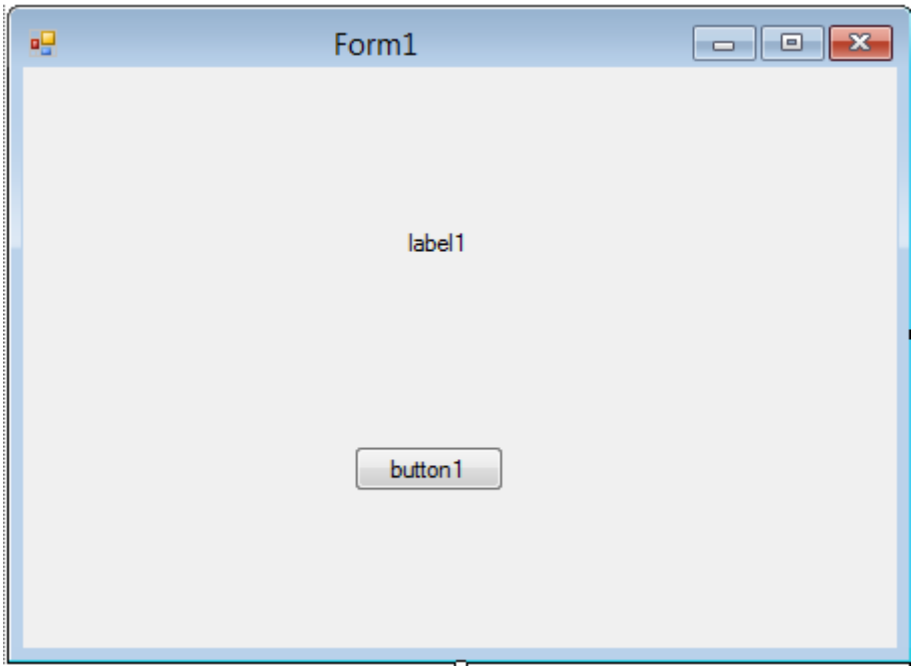


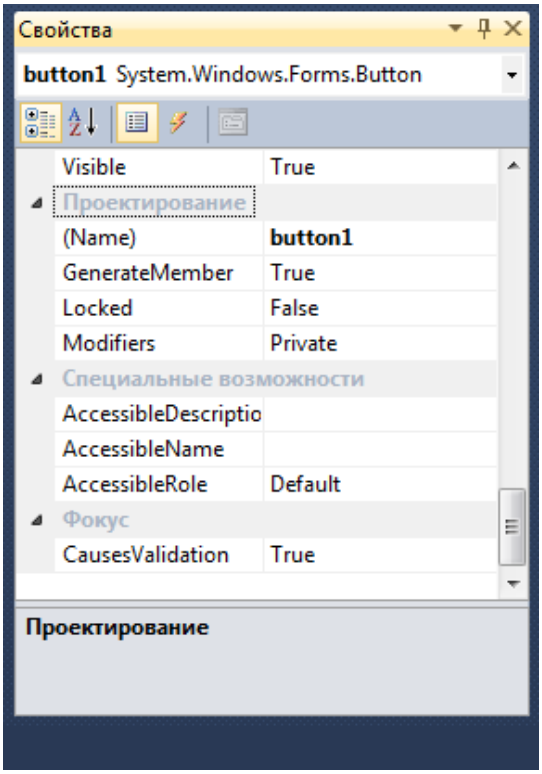
Рисунок 3 – Форма первого проекта

Вернемся к нашей задаче. Для объекта **labeii** выберем свойство **Text** и напишем напротив этого поля "Microsoft Visual C#" (вместо текста **label1).** Для объекта **button1** также в свойстве **Text** напишем "Нажми меня".

Кроме того, что объекты имеют свойства, следует знать, что объекты обрабатываются событиями. Событием, например, является щелчок на кнопке, щелчок в пределах формы, загрузка (**Load**) формы в оперативную память при старте программы и проч.

Управляют тем или иным событием посредством написания процедуры обработки события в программном коде. Для этого вначале нужно получить *"пустой" обработчик события.* В нашей задаче единственным событием, которым мы управляем, является щелчок на командной кнопке. Для получения пустого обработчика этого события следует в свойствах кнопки button1 (рис. 4) щелкнуть на значке молнии **События** (**Events**) и в списке всех возможных событий кнопки button1 выбрать двойным щелчком событие **MouseClick**. После этого мы попадем на вкладку программного кода **Forml.cs** (рис. 5).

При этом управляющая среда Visual C# сгенерировала пустой обработчик события



button1_MouseClick:

```
private void button1_MouseClick(object sender,
MouseEventArgs e) { }
```

Рисунок 4 – Форма первого проекта

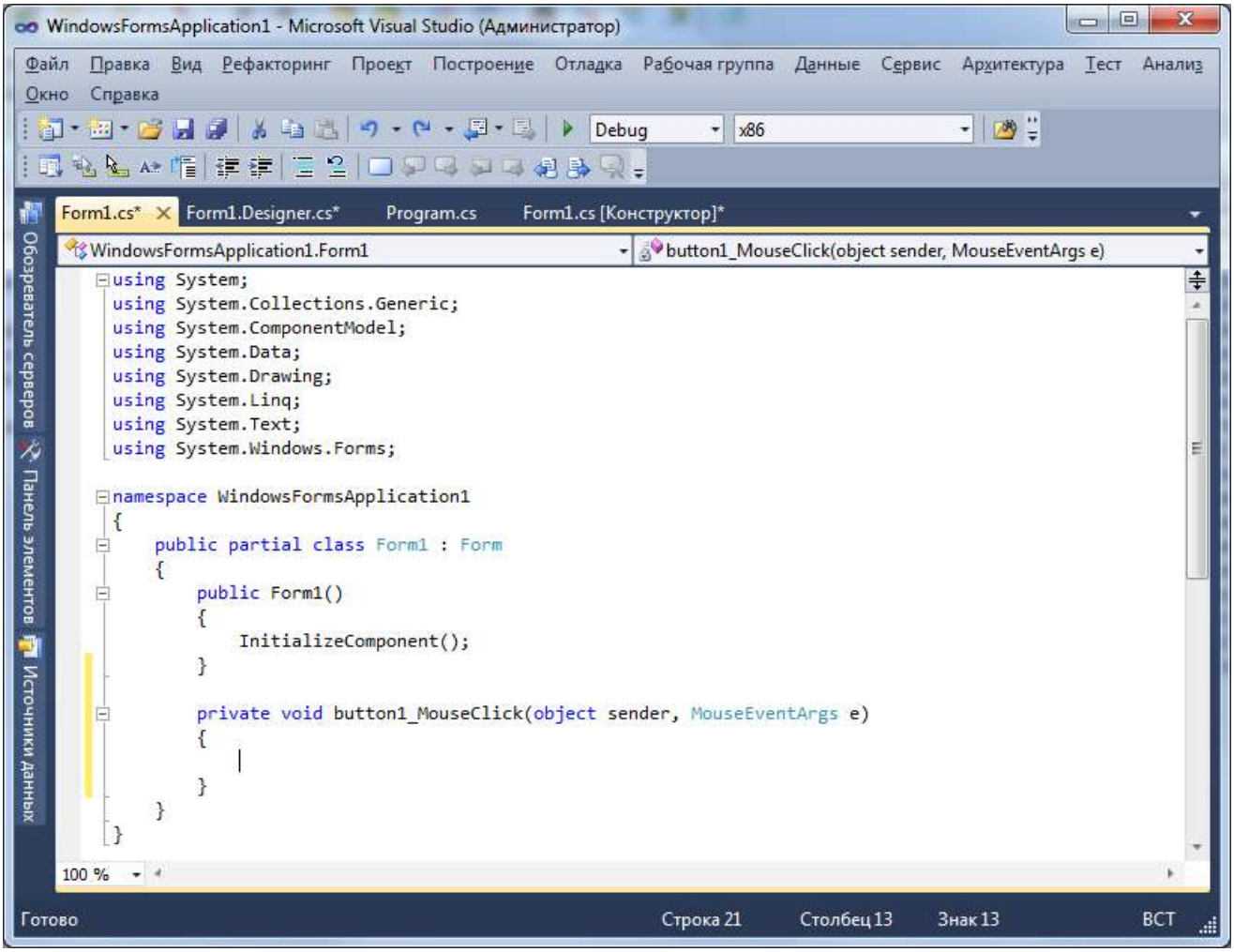


Рисунок 5 – Вкладка программного кода

В фигурных скобках этого обработчика события мы можем написать команды, подлежащие выполнению после щелчка на кнопке. Вы видите, что у нас теперь две вкладки: **Forml.cs** и **Forml.cs [Конструктор]**, т. е. вкладка программного кода и вкладка визуального проекта программы (другое название этой вкладки – *конструктор формы*). Переключаться между ними можно мышью или нажатием комбинации клавиш <Ctrl>+<Tab>, как это принято обычно между вкладками в Windows, а также функциональной клавишей <F7>.

Напомню, что после щелчка на кнопке должно появиться диалоговое окно, в котором написано: Всем привет! Поэтому в фигурных скобках обработчика события напишем:

```
MessageBox.Show( "Всем привет!");
```

Здесь вызывается метод (программа) Show объекта MessageBox с текстом “Всем привет!”. Таким образом, объекты кроме свойств имеют также и методы, т. е. программы, которые обрабатывают объекты.

Таким образом, мы написали *процедуру обработки события* щелчка (MouseClick) на кнопке button1. Теперь нажмем клавишу <F5> и проверим работоспособность программы (рис.6).

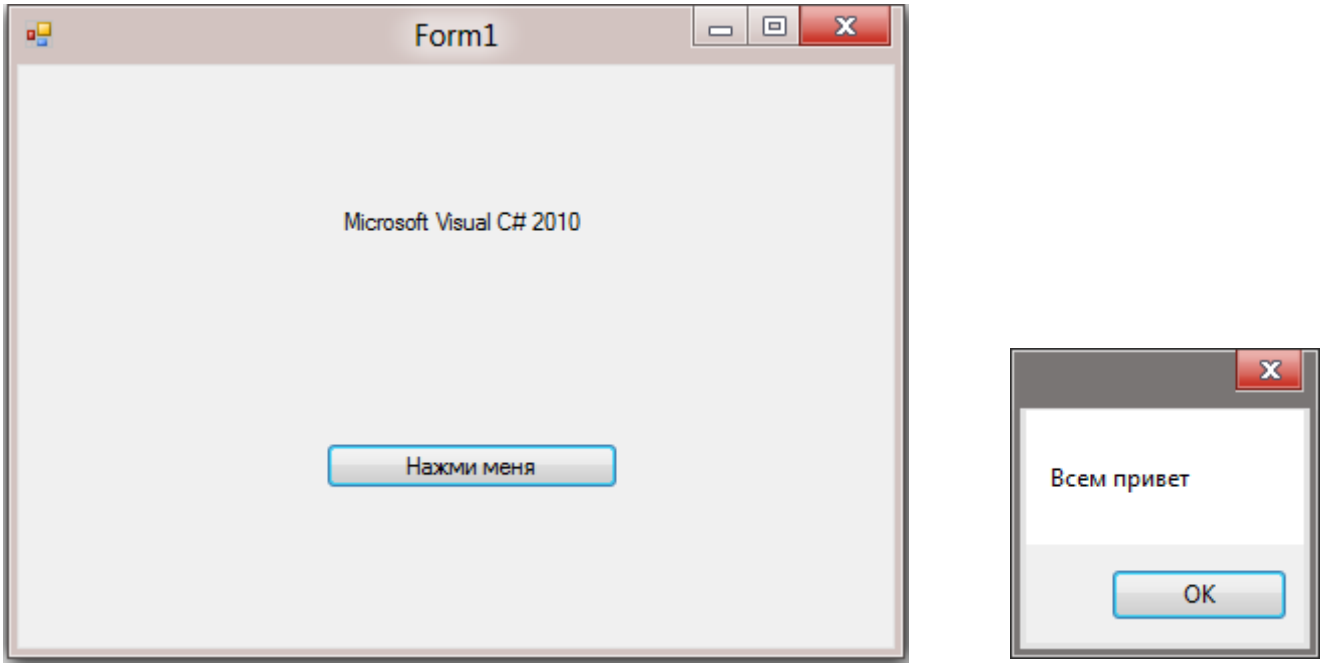


Рисунок 6 – Работающая программа

5.2 Событие MouseHover

Немного усложним предыдущую задачу. Добавим еще одну обработку события MouseHover мыши для объекта label1**.** Событие MouseHover наступает тогда, когда пользователь указателем мыши "зависает" над каким-либо объектом, причем именно "зависает", а не просто перемещает мышь над объектом (от англ. *hover* – реять, парить). Есть еще событие MouseEnter (Войти), когда указатель мыши *входит в пределы* области элемента управления (в данном случае метки label1**).**

Чтобы добавить обработку события MouseHover мыши для объекта label1**,** следует так же, как это мы делали при обработке события "щелчок на кнопке", в панели **Свойства (Properties)** щелкнуть на значке молнии **(События (Events))** и двойным щелчком выбрать для объекта label1 событие MouseHover**.** При этом осуществится переход на вкладку программного кода **Forml.cs** и среда Visual Studio сгенерирует простой обработчик события:

```
private  void  label1_MouseHover(object sender, EventArgs e) {        }
```

Между фигурными скобками вставим вызов диалогового окна:

```
MessageBox.Show("Событие Hover!");
```

Теперь проверим возможности программы: нажимаем клавишу <F5>, "зависаем" указателем мыши над label1**,** щелкаем на кнопке button1**.** Все работает!

А теперь мы будем немного противоречить самим себе. Мы говорили ранее про визуальную технику программирования, направленную на минимизацию написания программного кода. А сейчас надо сказать про *наглядность, оперативность, технологичность* работы программиста. Посмотрите на свойства каждого объекта в панели **Свойства** (**Properties**) Вы видите, как много строчек. Если вы меняете какое-либо свойство, то оно будет выделено жирным шрифтом. Удобно! Но все-таки еще более удобно свойства объектов *назначать* (устанавливать) *в программном коде.* Почему?

Каждый программист имеет в своем арсенале множество уже отлаженных фрагментов, которые он использует в своей очередной новой программе. Программисту стоит лишь вспомнить, где он программировал ту или иную ситуацию. Программа, которую написал программист, имеет свойство быстро забываться. Если вы посмотрите на строчки кода, которые писали три месяца назад, то будете ощущать, что многое забыли; если прошел год, то вы

смотрите на написанную вами программу, как на чужую. Поэтому при написании программ на первое место выходят *понятность, ясность, очевидность* написанного программного кода. Для этого каждая система программирования имеет какие-либо средства. Кроме того, сам программист придерживается некоторых правил, помогающих ему работать *про*изводительно и эффективно.

Назначать свойства объектов в программном коде удобно или сразу после инициализации компонентов формы (после процедуры InitializeComponent), или при обработке события Form1_Load, т. е. события загрузки формы в оперативную память при старте программы. Получим простой обработчик этого события. Для этого, как и в предыдущих случаях, можно выбрать нужное событие в панели свойств объекта, а можно еще проще: дважды щелкнуть в пределах проектируемой формы на вкладкё **Form1.cs [Конструктор].** В любом случае получаем пустой обработчик события на вкладке программного кода. Заметим, что для формы таким умалчиваемым событием, для которого можно получить пустой обработчик двойным щелчком, является событие загрузки формы Form1_Load, для командной кнопки **Button** и метки **Label** таким событием является одиночный щелчок мышью на этих элементах управления. То есть если дважды щелкнуть в дизайнере формы по кнопке, то получим пустой обработчик button1_ciick в программном коде, аналогично для метки **Label.**

Итак, вернемся к событию загрузки формы, для него управляющая среда сгенерировала пустой обработчик:

```
private  void  Form1_Load(object sender, EventArgs e) {  }
```

Здесь в фигурных скобках обычно вставляют свойства различных объектов и даже часто пишут много строчек программного кода. Здесь мы назначим свойству Text объекта label1 значение "Microsoft Visual C#":

```
label1.Text = "Microsoft Visual C#";
```

Аналогично для объекта button1:

```
button1.Text = "Нажми меня!";
```

Совершенно необязательно писать каждую букву приведенных команд. Например, для первой строчки достаточно написать "1a", уже это вызовет появление раскрывающегося меню, где вы сможете выбрать нужные для данного контекста ключевые слова. Это очень мощное и полезное современное средство, называемое IntellSense, для редактирования программного кода.

Вы написали название объекта label1, поставили точку. Теперь вы видите раскрывающееся меню, где можете выбрать либо нужное свойство объекта, либо метод (т. е. подпрограмму, функцию). В данном случае выберите свойство Text.

Как видите, не следует пугаться слишком длинных ключевых слов, длинных названий объектов, свойств, методов, имен переменных. Система подсказок современных систем программирования значительно облегчает всю нетворческую работу. Вот почему в современных программах можно наблюдать такие длинные имена ключевых слов, имен переменных и проч. Рекомендуется также использовать в своих программах для названия переменных, объектов наиболее ясные, полные имена, причем можно на вашем родном русском языке. Потому что на первое место выходят ясность, прозрачность программирования, а громоздкость названий с лихвой компенсируется системой подсказок.

Далее хотелось бы, чтобы слева вверху формы на синем фоне (в так называемой строке заголовка) была не надпись "Form1", а что-либо осмысленное. Например, слово "Приветствие". Для этого ниже присваиваем эту строку свойству Text формы. Поскольку мы изменяем свойство объекта Form1 внутри подпрограммы обработки события, связанного с формой, следует к форме обращаться через ссылку this: this.Text = "Приветствие" или base.Text = "Приветствие".

После написания последней строчки кода мы должны увидеть на экране программный код, представленный в листинге 1.

Листинг 1. Программирование событий

```
using System;
using System.Collections.Generic;
```

```
using System.ComponentModel;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }
   // Обработка события загрузки формы:
      private void Form1_Load(object sender, EventArgs e)
      {
         this.Text = "Приветствие";
         label1.Text = "Microsoft Visual C#";
         button1.Text = "Нажми меня";
      }
   // Обработка события щелчок на кнопке:
      private void button1_Click(object sender, EventArgs e)
      {
         MessageBox.Show("Всем привет!");
      }
   // Обработка события, когда указатель мыши "завис" над меткой:
      private void label1_MouseHover(object sender, EventArgs e)
      {
         MessageBox.Show("Событие Hover!");
      }
   }
}
```

Комментарии, поясняющие работу программы, в окне редактора кода будут выделены зеленым цветом, чтобы в тексте выразительно отделять его от прочих элементов программы. Даже если вам кажется весьма очевидным то, что вы пишете в программном коде, *напишите комментарий.* Как показывает опыт, даже весьма очевидный замысел программиста забывается удивительно быстро. Человеческая память отметает все, что по оценкам организма считается ненужным. Кроме того, даже если текст программы вполне ясен, в начале программы должны быть описаны ее назначение и способ использования, т. е. как бы "преамбула" программы. Далее в примерах мы будем следовать этому правилу.

Обычно в редакторах программного кода используется моноширинный шрифт, поскольку все символы такого шрифта имеют одинаковую ширину, в том числе точка и прописная русская буква "Ш". По умолчанию в редакторе программного кода задан шрифт Consolas. Однако если пользователь привык к шрифту Courier New, то его настройку можно изменить, выбрав меню **Сервис | Параметры | Среда | Шрифты и цвета** (**Tools | Options | Environment | Fonts and Colors**).

Теперь закроем проект **Файл | Закрыть решение (File | Close Project).** Система предложит нам сохранить проект, сохраним проект под именем Hover. Теперь программный код этой программы можно посмотреть, открыв решение Hover.sln в папке Hover.

3 Ввод данных через текстовое поле TextBox с проверкой типа методом TryParse

При работе с формой очень часто ввод данных организуют через элемент управления "текстовое поле" **TextBox.** Напишем типичную программу, которая вводит через текстовое поле число, при нажатии командной кнопки извлекает из него квадратный корень и выводит результат на метку **Label.** В случае ввода не числа сообщает пользователю об этом.

Решая сформулированную задачу, запускаем Visual Studio, выбираем пункт меню Файл | Создать | Проект (**File | New | Project),** затем – шаблон **Visual C# | Приложение Windows Forms**

(Visual C# | Windows Forms Application) и щелкаем на кнопке **ОК.** Далее из панели элементов управления **Панель элемен**тов (**Toolbox)** (если в данный момент вы не видите панель элементов, то ее можно добавить, например, с помощью комбинации клавиш <Ctrl>+<Alt>+<X> или меню **Вид** | **Панель элементов (View | Toolbox))** в форму указателем мыши перетаскиваем текстовое поле **TextBox,** метку **Label** и командную кнопку **Button.** Таким образом, в форме будут находиться три элемента управления.

Теперь перейдем на вкладку программного кода, для этого правой кнопкой мыши вызовем контекстное меню и выберем в нем пункт Перейти к коду (**View Code).** Здесь сразу после инициализации компонентов формы, т. е. после вызова процедуры **InitializeComponent** задаем свойствам формы (к форме обращаемся посредством ссылки this или base**),** кнопкам button1 и текстового поля textBox1**,** метке label1 следующие значения:

```
this.Text = "Извлечение квадратного корня";
button1.Text = "Извлечь корень";
textBox1.Clear();   // Очистка текстового поля
labeii.Text = null; // или = string.Empty;
```

Нажмем клавишу <F5> для выявления возможных опечаток, т. е. синтаксических ошибок и предварительного просмотра дизайна будущей программы.

Далее программируем событие button1_Click – щелчок мышью на кнопке **Извлечь корень.** Создать пустой обработчик этого события удобно, дважды щелкнув мышью на этой кнопке. Между двумя появившимися строчками программируем диагностику правильности вводимых данных, конвертирование строковой переменной в переменную типа Single и непосредственное извлечение корня (листинг 2).

Листинг 2. Извлечение корня с проверкой типа методом TryParse

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
// Программа вводит через текстовое поле число, при щелчке на командной кнопке
// извлекает из него квадратный корень и выводит результат на метку label1. В случае
// ввода не числа сообщает пользователю об этом красным цветом также на метку label1
namespace Sqroot
{
   public partial class Form1 : Form
   {
      public Form1()
      { // Инициализация компонентов формы
         InitializeComponent();
         button1.Text = "Извлечь корень";
         label1.Text = null; // или = ""
         this.Text = "Извлечение квадратного корня";
         textBox1.Clear(); // Очистка текстового поля
         textBox1.TabIndex = 0; // Установка фокуса в текстовом поле
      }
      private void button1_Click(object sender, EventArgs e)
      { // Обработка события щелчок на кнопке Извлечь корень
         Single X; // - из этого числа будем извлекать корень
         // Преобразование из строковой переменной в Single:
         bool Число_ли = Single.TryParse(textBox1.Text,
            System.Globalization.NumberStyles.Number,
            System.Globalization.NumberFormatInfo.CurrentInfo, out X);
         // второй параметр - это разрешенный стиль числа (Integer,
         // шестнадцатиричное число, экспоненциальный вид числа и прочее)
         // третий параметр - форматирует значения на основе текущего языка
         // и региональных параметров из Панели управления - Язык и региональные
         // стандарты число допустимого формата,
```

```
            // метод возвращает значение в переменную X
            if (Число_ли == false)
            { // Если пользователь ввел не число:
               label1.Text = "Следует вводить числа";
               label1.ForeColor = Color.Red; // - красный цвет текста на метке
               return; // - выход из процедуры
            }
            Single Y = (Single)Math.Sqrt(X); // - извлечение корня
            label1.ForeColor = Color.Black; // - черный цвет текста на метке
            label1.Text = string.Format("Корень из {0} равен {1:F5}", X, Y);
        }
    }
}
```

Здесь при обработке события "щелчок мышью" на кнопке **Извлечь корень** проводится проверка, введено ли число в текстовом поле. Проверка осуществляется с помощью функции TryParse**.** Первым параметром метода TryParse является анализируемое поле textBox1.Text**.** Второй параметр – это разрешаемый для преобразования стиль числа, он может быть целого типа **(**Integer**),** шестнадцатеричным **(**HexNumber**),** представленным в экспоненциальном виде и проч. Третий параметр указывает, на какой основе формируется допустимый формат, в нашем случае мы использовали CurrentInfo**,** т. е. на основе текущего языка и региональных параметров. По умолчанию при инсталляции русской версии Windows разделителем целой и дробной части числа является запятая. Однако эту установку можно изменить, если в **Панели управления** выбрать значок **Язык и региональные стандарты**, затем на вкладке **Региональные параметры** щелкнуть на кнопке **Настройка** и на появившейся новой вкладке указать в качестве разделителя целой и дробной частей точку вместо запятой. В обоих случаях (и для запятой, и для точки) метод TryParse будет работать так, как указано на вкладке **Региональные параметры**.

Четвертый параметр метода TryParse возвращает результат преобразования. Кроме того, функция TryParse возвращает булеву переменную true или false**,** которая сообщает, успешно ли выполнено преобразование. Как видно из текста программы, если пользователь ввел не число (например, введены буквы), то на метку label1 выводится красным цветом текст "Следует вводить числа". Далее, поскольку ввод неправильный, организован выход из программы обработки события button1_Click с помощью оператора return**.** На рис. 8 показан фрагмент работы программы.

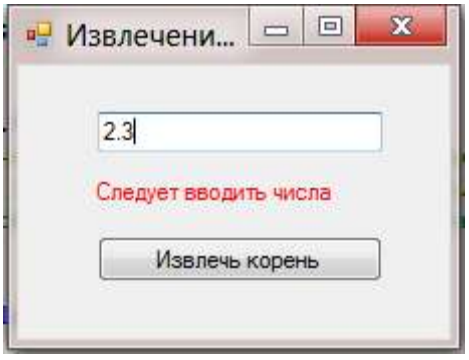


Рисунок 8 – Фрагмент работы программы

Как видно из рисунка, функция TryParse не восприняла введенные символы "2.3" как число, так как для демонстрации данного примера на вкладке **Региональные параметры** запятая выбрана в качестве разделителя целой и дробной частей.

Если пользователь ввел все-таки число, то будет выполняться следующий оператор извлечения квадратного корня Math.Sqrt (x). Математические функции Visual Studio 2010 являются методами класса Math**.** Их можно увидеть, набрав Math и поставив точку (.). В выпадающем списке вы увидите множество математических функций: Abs**,** Sin**,** Cos**,** Min и т. д. и два свойства – две константы E = 2.71... (основание натуральных логарифмов) и PI = 3,14... (число диаметров, уложенных вдоль окружности). Функция Math.Sqrt (x) возвращает значение

типа double (двойной точности с плавающей запятой), которое *приводим* с помощью неявного преобразования (Single) к переменной одинарной точности.

Последней строчкой обработки события button1_Click является формирование строки label1.Text с использованием метода string.Format. Использованный формат "Корень из {0} равен {1:F5>" означает: взять нулевой выводимый элемент, т. е. переменную X, и записать эту переменную вместо фигурных скобок; после чего взять первый выводимый элемент, т. е. Y, и записать его вместо вторых фигурных скобок в формате с фиксированной точкой и пятью десятичными знаками после запятой.

Нажав клавишу <F5>, проверяем, как работает программа.

Результат работающей программы представлен на рис. 9.

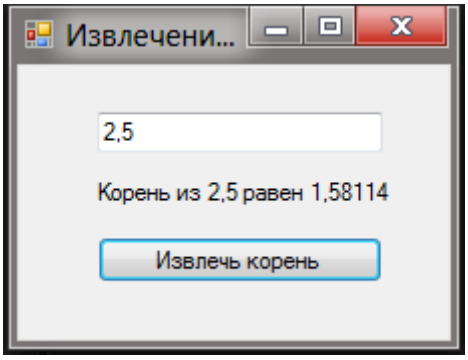


Рисунок 9 – Извлечение квадратного корня

Если появились ошибки, то работу программы следует проверить отладчиком – клавиши <F8> или <F11>. В этом случае управление останавливается на каждом операторе, и вы можете проверить значение каждой переменной, наводя указатель мыши на переменные. Можно выполнить программу до определенной программистом точки *(точки останова),* используя, например, клавишу <F9> или оператор Stop, и в этой точке проверить значения необходимых переменных.

Убедиться в работоспособности этой программы можно, открыв соответствующее решение в папке SQroot.

4 Ввод пароля в текстовое поле и изменение шрифта

Это очень маленькая программа для ввода пароля в текстовое поле, причем при вводе вместо вводимых символов некто, "находящийся за спиной пользователя", увидит только звездочки. Программа состоит из формы, текстового поля TextBox, метки **Label**, куда для демонстрации возможностей мы будем копировать пароль (т. е. секретные слова) и командной кнопки **Button – Введите пароль.**

Перемещаем в форму все названные элементы управления. Текст программы приведен в листинге 3.

Листинг 3. Ввод пароля

```
// Программа ввода пароля в текстовое поле, причем при вводе вместо
// вводимых символов некто, "находящийся за спиной пользователя",
// увидит только звездочки

using System;
using System.Drawing;
using System.Windows.Forms;
// Другие директивы using удалены, поскольку они не используются
// в данной программе

namespace Passport
{
```

```
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
         base.Text = "Введи пароль";
         textBox1.Text = null;
         textBox1.TabIndex = 0;
         textBox1.PasswordChar = '*';
         textBox1.Font = new Font("Courier New", 9.0F);
         // или textBox1.Font = new Font(FontFamily.GenericMonospace, 9.0F);
         label1.Text = "";
         label1.Font = new Font("Courier New", 9.0F);
         button1.Text = "Введите пароль";
      }
      private void button1_Click(object sender, EventArgs e)
      {
         // Обработка события "щелчок на кнопке"
         label1.Text = textBox1.Text;
      }
   }
}
```

Как видно из текста программы, сразу после инициализации компонентов формы, т. е. после вызова процедуры `InitializeComponent`, очищаем текстовое поле и делаем его "защищенным от посторонних глаз" с помощью свойства `textBox1.PasswordChar`, каждый введенный пользователем символ маскируется символом звездочки (*). Далее мы хотели бы для большей выразительности и читабельности программы, чтобы вводимые звездочки и результирующий текст имели одинаковую длину: Все символы шрифта Courier New имеют одинаковую ширину, поэтому его называют моноширинным шрифтом. Кстати, используя именно этот шрифт, удобно программировать таблицу благодаря одинаковой ширине букв этого шрифта. Еще одним широко используемым моноширинным шрифтом является шрифт Consolas. Задаем шрифт, используя свойство `Font` обоих объектов: `textBox1` и `label1`. Число `9.0` означает размер шрифта. Свойство текстового поля `TabIndex = 0` обеспечивает передачу фокуса при старте программы именно в текстовое поле.

Осталось обработать событие `button1_Click` – щелчок на кнопке. Здесь – банальное присваивание текста из поля тексту метки. Программа написана, нажимаем клавишу <F5>. На рис. 10 приведен вариант работы данной программы.

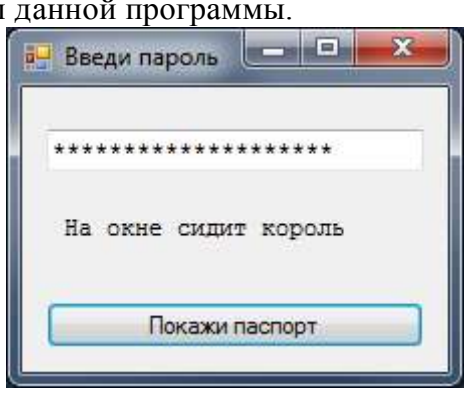


Рисунок 10 – Вариант работы программы

При необходимости используйте отладчик (клавиша <F11> или <F10>) для *пошагового выполнения программы* и выяснения всех промежуточных значений переменных путем "зависания" указателя мыши над переменными.

5 Управление стилем шрифта с помощью элемента управления CheckBox

Кнопка **CheckBox** (Флажок) также находится на панели элементов управления Панель элементов (**Toolbox**). Флажок может быть либо установлен (содержит "галочку"), либо сброшен (пустой). Напишем программу, которая управляет стилем шрифта текста, выведенного на метку **Label.** Управлять стилем будем посредством флажка **CheckBox.**

Используя панель элементов **Toolbox,** в форму поместим метку label1 и флажок checkBox1. В листинге 4 приведен текст программы управления этими объектами.

Листинг4. Управление стилем шрифта

```
// Программа управляет стилем шрифта текста, выведенного на метку
// Label, посредством флажка CheckBox
using System;
using System.Drawing;
using System.Windows.Forms;
// Другие директивы using удалены, поскольку они не используются в данной программе

namespace CheckBox
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
         this.Text = "Флажок CheckBox";
         checkBox1.Text = "Полужирный";
         checkBox1.Focus();
         label1.Text = "Выбери стиль шрифта";
         label1.TextAlign = ContentAlignment.MiddleCenter;
         label1.Font = new System.Drawing.Font("Courier New", 14.0F);
      }

      private void checkBox1_CheckedChanged(object sender, EventArgs e)
      { // Изменение состояния флажка на противоположное
         if (checkBox1.Checked == true) label1.Font =
                     new Font("Courier New", 14.0F, FontStyle.Bold);
         if (checkBox1.Checked == false) label1.Font =
                     new Font("Courier New", 14.0F, FontStyle.Regular);
      }
   }
}
```

Сразу после вызова процедуры InitializeComponent задаем начальные значения некоторых свойств объектов Form1 (посредством ссылки this**),** label1 и checkBox1**.** Так, тексту флажка, выводимого с правой стороны, присваиваем значение "Полужирный". Кроме того, при старте программы фокус должен находиться на флажке **(**checkBox1.Focus();), в этом случае пользователь может изменять установку флажка даже клавишей <Пробел>.

Текст метки – "Выбери стиль шрифта", выравнивание метки TextAlign задаем посередине и по центру **(**MiddleCenter**)** относительно всего того места, что предназначено для метки. Задаем шрифт метки Courier New (в этом шрифте все буквы имеют одинаковую ширину) размером 14 пунктов.

Изменение состояния флажка соответствует событию CheckedChanged**.** Чтобы получить пустой обработчик события CheckedChanged**,** следует дважды щелкнуть на элементе checkBox1 вкладки **Forml.cs [Конструктор].** Между соответствующими строчками следует записать (см. текст программы): если флажок установлен (т. е. содержит "галочку") Checked = true**,** то для метки label1 устанавливается тот же шрифт Courier New, 14 пунктов, но Bold**,** т. е. полужирный.

Далее – следующая строчка кода: если флажок не установлен, т. е. checkBox1.Checked = false, то шрифт устанавливается Regular**,** т. е. обычный. Очень часто эту ситуацию программируют, используя ключевое слово else (иначе), однако это выражение будет выглядеть более выразительно и понятно так, как написали мы.

Программа написана, нажмите клавишу <F5>. Проверьте работоспособность программы. В рабочем состоянии она должна работать примерно так, как показано на рис.11.

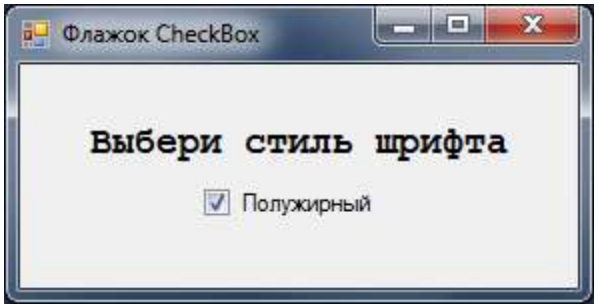


Рисунок 11 – Фрагмент работы программы управления стилем шрифта

6 Оператор "исключающее ИЛИ"

Несколько изменим предыдущую программу в части обработки события CheckedChanged (Изменение состояния флажка). Вместо двух условий if() напишем один оператор:

```
label1.Font = new Font("Courier New", 14.0F, label1.Font.Style ^ FontStyle.Bold);
```

Здесь каждый раз при изменении состояния флажка значение параметра label1.Font.Style сравнивается с одним и тем же значением FontStyle.Bold**.** Поскольку между ними стоит побитовый оператор ^ (исключающее ИЛИ), он будет назначать Bold**,** если текущее состояние label1.Font.Style "не Bold". А если label1.Font.style пребывает в состоянии "Bold", то оператор ^ будет назначать состояние "не Bold". Этот оператор еще называют логическим исключающим ИЛИ (XOR)**.**

Как видно, применение побитового оператора привело к существенному уменьшению количества программного кода. Использование побитовых операторов может значительно упростить написание программ со сложной логикой. Посмотрите, как работает программа, нажав клавишу <F5>.

Теперь добавим в форму еще один элемент управления **CheckBox**. Мы собираемся управлять стилем шрифта FontStyle двумя флажками. Один, как и прежде, задает полужирный стиль Bold или обычный Regular**,** а второй задает наклонный Italic или возвращает в Regular**.** Текст новой программы приведен в листинге 5.

Листинг 5. Усовершенствованный программный код

```
// Побитовый оператор "^" - исключающее ИЛИ
using System;
using System.Drawing;
using System.Windows.Forms;
// Другие директивы using удалены, поскольку они не используются в данной программе

namespace CheckBox2
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
         this.Text = "Флажок CheckBox";
         checkBox1.Text = "Полужирный"; checkBox2.Text = "Наклонный";
         label1.Text = "Выбери стиль шрифта";
         label1.TextAlign = ContentAlignment.MiddleCenter;
         label1.Font = new Font("Courier New", 14.0F);
      }
      private void checkBox1_CheckedChanged(object sender, EventArgs e)
      {
         label1.Font = new System.Drawing.Font(
            "Courier New", 14.0F, label1.Font.Style ^ FontStyle.Bold);
      }
```

```
        private void checkBox2_CheckedChanged(object sender, EventArgs e)
        {
            label1.Font = new System.Drawing.Font(
                "Courier New", 14.0F, label1.Font.Style ^ FontStyle.Italic);
        }
    }
}
```

Как видно, здесь принципиально ничего нового нет, только лишь добавлена обработка события изменения состояния флажка `CheckedChanged` для `CheckBox2`**.** Фрагмент работы программы можно увидеть на рис. 12.

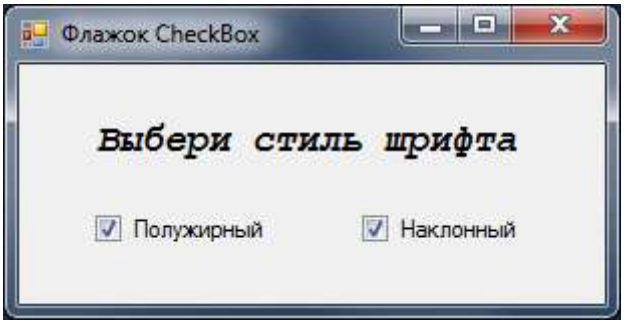


Рисунок 12 – Фрагмент работы усовершенствованной программы

7 Вкладки TabControl и переключатели RadioButton

Вкладки используются для организации управления и оптимального расположения экранного пространства. То есть если требуется отобразить большое количество управляемой информации, то весьма уместно использовать вкладки **TabControl**.

Поставим задачу написать программу, позволяющую выбрать текст из двух вариантов, задать цвет и размер шрифта этого текста на трех вкладках **TabControl** с использованием переключателей **RadioButton.**

Фрагмент работы программы приведен на рис. 13.

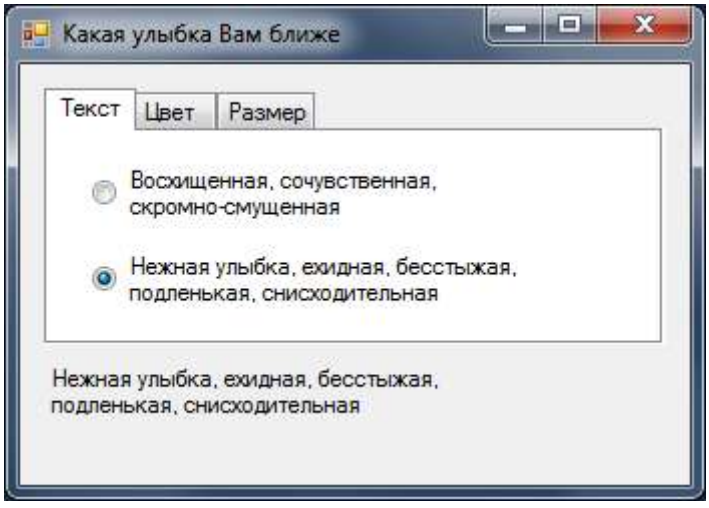


Рисунок 13 – Программа с переключателями и вкладками

Программируя поставленную задачу, создадим новый проект **Visual C# | Приложение Windows Forms,** назовем его, например, `ВкладкиTabControl`, получим стандартную форму. Затем, используя панель элементов Панель элементов (**Toolbox**), в форму перетащим мышью элемент управления **TabControl.** Как видно, по умолчанию имеем две вкладки, а по условию задачи, как показано на рис. 13, три вкладки. Добавить третью вкладку можно в конструкторе формы, а можно программно.

Вначале покажем, как добавить третью вкладку в конструкторе. Для этого в свойствах (окно Свойства (**Properties**)**)** элемента управления `TabControl1` выбираем свойство `TabPages`**,** в

результате попадаем в диалоговое окно **Редактор коллекции TabPage** (**TabPage Collection Edit),** где добавляем (кнопка **Добавить** (**Add**)) третью вкладку (первые две присутствуют по умолчанию). Эти вкладки нумеруются от нуля, т.е. третья вкладка будет распознаваться как TabPages(2). Название каждой вкладки будем указывать в программном коде.

Покажем, как добавить третью вкладку не в конструкторе, а в программном коде сразу после вызова процедуры InitiaiizeComponent (листинг 6**).** Однако, прежде чем перейти на вкладку программного кода, для каждой вкладки выбираем из панели **Панель элементов** (**Toolbox)** по два переключателя **RadioButton,** а в форму перетаскиваем метку **Label.** Теперь через щелчок правой кнопкой мыши в пределах формы переключаемся на редактирование программного кода.

Листинг 6. Использование вкладок и переключателей

```
// Программа, позволяющая выбрать текст из двух вариантов, задать цвет и размер
// шрифта для этого текста на трех вкладках TabControl с использованием
// переключателей RadioButton

using System;
using System.Drawing;
using System.Windows.Forms;
// Другие директивы using удалены, поскольку они не используются в данной программе

namespace PageTabControl
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
         // Создание третьей вкладки "программно":
         System.Windows.Forms.TabPage tabPage3 =
                              new System.Windows.Forms.TabPage();
         tabPage3.UseVisualStyleBackColor = true;
         // Добавление третьей вкладки в существующий набор вкладок tabControl1:
         this.tabControl1.Controls.Add(tabPage3);
         //this.Visible = false;

         // Добавление переключателей 5 и 6 на третью вкладку:
         tabPage3.Controls.Add(this.radioButton5);
         tabPage3.Controls.Add(this.radioButton6);

         // Расположение переключателей 5 и 6:
         this.radioButton5.Location = new System.Drawing.Point(20, 15);
         this.radioButton6.Location = new System.Drawing.Point(20, 58);

         this.Text = "Какая улыбка Вам ближе";
         // Задаем названия вкладок:
         tabControl1.TabPages[0].Text = "Текст";
         tabControl1.TabPages[1].Text = "Цвет";
         tabControl1.TabPages[2].Text = "Размер";

         // Эта пара переключателей изменяет текст:
         radioButton1.Text =
                  "Восхищенная, сочувственная,\nскромно-смущенная";
         radioButton2.Text = "Хитрая улыбка, ехидная, " +
                        "\nподленькая, снисходительная";

         // Эта пара переключателей изменяет цвет текста:
         radioButton3.Text = "Красный";
         radioButton4.Text = "Синий";

         // Эта пара переключателей изменяет размет шрифта:
         radioButton5.Text = "11 пунктов";
```

```
            radioButton6.Text = "13 пунктов";

            label1.Text = radioButton1.Text;
        }

        // Ниже обработки событий изменения состояния шести переключателей
        private void radioButton1_CheckedChanged(object sender, EventArgs e)
            { label1.Text = radioButton1.Text; }

        private void radioButton2_CheckedChanged(object sender, EventArgs e)
            { label1.Text = radioButton2.Text; }

        private void radioButton3_CheckedChanged(object sender, EventArgs e)
            { label1.ForeColor = Color.Red; }

        private void radioButton4_CheckedChanged(object sender, EventArgs e)
            { label1.ForeColor = Color.Blue; }

        private void radioButton5_CheckedChanged(object sender, EventArgs e)
            { label1.Font = new Font(label1.Font.Name, 11); }

        private void radioButton6_CheckedChanged(object sender, EventArgs e)
            { label1.Font = new Font(label1.Font.Name, 13); }
    }
}
```

Как видно из текста программы, сразу после вызова процедуры InitializeComponent (этот программный код можно было бы задать при обработке события загрузки формы Form1_Load**)** создаем "программно" третью вкладку и добавляем ее в набор вкладок tabControl1**,** созданный в конструкторе. Далее "привязываем" пятый и шестой переключатели к третьей вкладке.

Дальнейшие установки очевидны. Заметим, что каждая пара переключателей, расположенных на каком-либо элементе управления (в данном случае на различных вкладках), "отрицают" друг друга, т. е. если пользователь выбрал один, то другой переходит в противоположное состояние. Отслеживать изменения состояния переключателей удобно с помощью обработки событий переключателей CheckChanged (листинг 6**).** Чтобы получить пустой обработчик этого события в конструкторе формы, следует дважды щелкнуть на соответствующем переключателе и таким образом запрограммировать изменения состояния переключателей.

8 Свойство Visible и всплывающая подсказка ToolTip в стиле Balloon

Продемонстрируем возможности свойства **Visible** (Видимый). Программа пишет в метку **Label** некоторый текст, а пользователь с помощью командной кнопки делает этот текст невидимым, а затем опять видимым и т.д. При зависании мыши над кнопкой появляется подсказка "Нажми меня".

Запускаем Visual Studio, далее выбираем пункты меню **Файл | Создать | Проект | Visual C# | Приложение Windows Forms** (**File | New | Project | Windows Forms Application C#)** и нажимаем кнопку **ОК.** Затем из панели элементов управления **Панель элементов** (**Toolbox)** в форму перетаскиваем метку **Label,** кнопку **Button** и всплывающую подсказку **ToolTip.** Только в этом случае каждый элемент управления в форме (включая форму) получает свойство ToolTip on Tip**.** Убедитесь в этом, посмотрев свойства (окно Свойства (**Properties**)) элементов управления.

Для кнопки button1 напротив свойства ToolTip on Tip мы могли бы написать "Нажми меня". Однако можно написать это непосредственно в программном коде. В этом случае программист не будет долго искать соответствующее свойство, когда станет применять данный фрагмент в своей новой программе.

Перейдем на вкладку программного кода – щелчок правой кнопкой мыши в пределах формы и выбор команды Перейти к коду (**View Code).** Окончательный текст программы представлен в листинге 7.

Листинг 7. Свойство Visible и всплывающая подсказка ToolTip

```
// Программа пишет в метку Label некоторый текст, а пользователь
// с помощью командной кнопки делает этот текст либо видимым, либо
// невидимым. Здесь использовано свойство Visible. При зависании мыши
// над кнопкой появляется подсказка "Нажми меня" (свойство ToolTip).
using System;
using System.Drawing;
using System.Windows.Forms;
namespace Visible
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
         base.Text = "Житейская мудрость";
         label1.Text = "Сколько ребенка не учи хорошим манерам,\n" +
                   "он будет поступать так, как папа с мамой";
         label1.TextAlign = ContentAlignment.MiddleCenter;
         button1.Text = "Кнопка";
         toolTip1.SetToolTip(button1, "Кнопка\r\nсчастья");
         // Должна ли всплывающая подсказка использовать всплывающее окно:
         toolTip1.IsBalloon = true;
         // Если IsBalloon = false, то используется стандартное прямоугольное окно
      }
      private void button1_Click(object sender, EventArgs e)
      {
         // Обработка события щелчок на кнопке:
         // Можно программировать так:
         // if (label1.Visible) label1.Visible = false;
         // else label1.Visible = true;
         // или так:
         // label1.Visible = label1.Visible ^ true;
         // здесь "^" - логическое исключающее ИЛИ,
         // или еще проще:
         label1.Visible = !label1.Visible;
      }
   }
}
```

Сразу после вызова процедуры InitializeComponent свойству Text метки присваиваем некоторый текст, "склеивая" его с помощью знака "плюс" (+) из отдельных фрагментов. Использование в текстовой строке символов "\n" (или "\r\n") означает перенос текста на новую строку (это так называемый *перевод каретки).* Можно переводить каретку с помощью строки Environment.NewLine. В перечислении Environment можно выбрать и другие управляющие символы. Свойство метки TextAlign располагает текст метки по центру и посередине (MiddleCenter). Выражение, содержащее ToolTip1, устанавливает (Set) текст всплывающей подсказки для кнопки button1 при "зависании" над ней указателя мыши (рис. 14). По умолчанию свойство isBalloon пребывает в состоянии false, и при этом во всплывающей подсказке используется стандартное прямоугольное окно, установка isBalloon = true переводит подсказку в стиль Balloon (см. рис. 14).

Чтобы в программном коде получить пустой обработчик события "щелчок мышью на кнопке", следует в дизайнере (конструкторе) формы (т. е. на вкладке **Forml.cs[Конструктор]**) дважды щелкнуть на кнопке button1. При обработке этого события, как видно, закомментированы пять строчек, в которых записана логика включения видимости метки или ее выключение. Логика абсолютно понятна: если свойство видимости (Visible) *включено* (true), то его следует *выключить* (false); иначе (else) – включить.

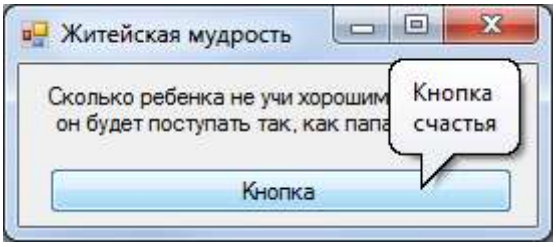


Рисунок 14 – Фрагмент работы программы

Так все работает. Проверьте. Кнопку можно нажимать мышью, клавишей <Enter> и клавишей <Пробел>.

Однако можно пойти другим путем. Именно поэтому пять строчек этой сложной логики переведены в комментарий. Мы уже встречались с побитовым оператором л (исключающее ИЛИ). Напоминаю, что этот оператор, говоря кратко, выбирает "да" **(**true**),** сравнивая "нет" и "да", и выбирает "нет" **(**false**),** сравнивая "да" и "да". Однако можно еще более упростить написание программного кода: label1.Visible = ! label1.Visible;

То есть при очередной передаче управления на эту строчку свойство label1.visible будет принимать противоположное значение. Вы убедились, что можно по-разному программировать подобные ситуации.

Как видно, здесь использовали много закомментированных строчек программного кода. Очень удобно комментировать строки программного кода в редакторе Visual Studio, вначале выделяя их, а затем использовать комбинацию клавиш <Ctrl>+<K> | <C> (Comment). Чтобы убрать с нескольких строк знак комментария, можно аналогично вначале отметить их, а затем пользоваться уже дугой комбинацией клавиш <Ctrl>+<K> | <U> (Uncomment).

9 Калькулятор на основе комбинированного списка ComboBox

Элемент управления **ComboBox** служит для отображения вариантов выбора в выпадающем списке. Продемонстрируем работу этого элемента управления на примере программы, реализующей функции калькулятора. Здесь для отображения вариантов выбора арифметических операций используется комбинированный список **ComboBox.**

После запуска Visual Studio и выбора шаблона **Приложение Windows Forms** (**Windows Forms Application C#)** из панели **Панель элементов** (**Toolbox)** перетащим в форму два текстовых поля **TextBox,** метку **Label** и комбинированный список **ComboBox.**

Текст программы представлен в листинге 8.

Листинг 8. Суперкалькулятор

```
// Программа, реализующая функции калькулятора. Здесь для отображения вариантов
// выбора арифметических действий используется комбинированный список ComboBox

using System;
using System.Windows.Forms;

// Другие директивы using удалены, поскольку они не используются в данной программе

namespace ComboBox_Calc
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
         comboBox1.Text = "Выбери операцию";
         comboBox1.Items.AddRange(new string[] {"Прибавить",
                  "Отнять", "Умножить", "Разделить", "Очистить"});
         comboBox1.TabIndex = 2;
         textBox1.Clear(); textBox2.Clear();
         textBox1.TabIndex = 0; textBox2.TabIndex = 1;
         this.Text = "Суперкалькулятор";
```

```
            label1.Text = "Равно: ";
        }
        private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
            // Обработка события изменения индекса выбранного элемента
            label1.Text = "Равно: ";
            // Преобразование из строковой переменной в Single:
            Single X, Y, Z = 0;
            bool Число_ли1 = Single.TryParse(textBox1.Text,
                System.Globalization.NumberStyles.Number,
                System.Globalization.NumberFormatInfo.CurrentInfo, out X);
            bool Число_ли2 = Single.TryParse(textBox2.Text,
                System.Globalization.NumberStyles.Number,
                System.Globalization.NumberFormatInfo.CurrentInfo, out Y);
            if (Число_ли1 == false || Число_ли2 == false)
            {
                MessageBox.Show("Следует вводить числа!", "Ошибка",
                MessageBoxButtons.OK, MessageBoxIcon.Error);
                return;
            }
            // Оператор множественного выбора:
            switch (comboBox1.SelectedIndex) // Выбор арифмет. операции:
            {
                case 0:  // Выбрали "Прибавить":
                    Z = X + Y; break;
                case 1:  // Выбрали "Отнять":
                    Z = X - Y; break;
                case 2:  // Выбрали "Умножить":
                    Z = X * Y; break;
                case 3:  // Выбрали "Разделить":
                    Z = X / Y; break;
                case 4:  // Выбрали "Очистить":
                    textBox1.Clear(); textBox2.Clear();
                    label1.Text = "Равно: "; return;
            }
            label1.Text = string.Format("Равно {0:F5}", Z);
        }
    }
}
```

В данной программе сразу после вызова процедуры InitializeComponent присваиваем начальные значения некоторым свойствам, в том числе задаем коллекцию элементов комбинированного списка: "Прибавить", "Отнять" и т. д. Здесь также задаем табличные индексы TabIndex для текстовых полей и комбинированного списка. Табличный индекс определяет порядок обхода элементов. Так, при старте программы фокус будет находиться в первом текстовом поле, поскольку мы назначили textBoxi.Tabindex = 0. Далее при нажатии пользователем клавиши <Tab> будет происходить переход от элемента к элементу соответственно табличным индексам (рис. 15).

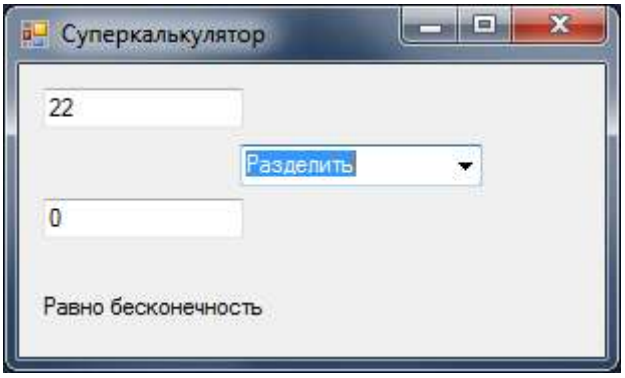


Рисунок 15 – Переход от одного текстового поля к другому

При обработке события "изменение индекса выбранного элемента" comboBoxl_SelectedIndexChanged с помощью функции TryParse проверяем, можно ли текстовые поля преобразовать в число. Первым параметром метода TryParse является анализируемое поле. Второй параметр – это разрешаемый для преобразования стиль числа, в данном случае типа Number**,** т. е. десятичное число, которое имеет целую и дробную части. Третий параметр указывает, на какой основе формируется допустимый формат, в нашем случае мы использовали CurrentInfo**,** т. е. на основе текущего языка и региональных параметров. По умолчанию при инсталляции руссифицированной версии Windows разделителем целой и дробной частей числа является запятая. Однако эту установку можно изменить, если в Панели управления выбрать значок **Язык и региональные стандарты**, а затем на вкладке **Региональные параметры** щелкнуть на кнопке **Настройк**а и на появившейся новой вкладке указать в качестве разделителя целой и дробной частей точку вместо запятой. В обоих случаях (и для запятой, и для точки) метод TryParse будет работать так, как указано на вкладке Региональные параметры.

Четвертый параметр метода TryParse возвращает результат преобразования. Кроме того, функция TryParse возвращает булеву переменную true или false**,** которая сообщает, успешно ли выполнено преобразование. Как видно из текста программы, если хотя бы одно поле невозможно преобразовать в число, то программируем сообщение "Следует вводить числа!" и выход из процедуры обработки события с помощью оператора return**.**

Далее оператор switch осуществляет множественный выбор арифметической операции в зависимости от индекса выбранного элемента списка **selectedIndex.** Оператор switch передает управление той или иной "метке case**".** Причем, как говорят программисты, оператор множественного выбора в Си-подобных языках "проваливается", т. е. управление переходит на следующую метку case**,** поэтому приходится использовать оператор break для выхода из switch**.**

Последний оператор в процедуре осуществляет формирование строки с помощью метода string.Format для вывода ее на метку label1**.** Формат "{0:F5}" означает, что значение переменной z, следует выводить по фиксированному формату с *пятью знаками* после запятой (или точки). Заметьте, в этом примере не пришлось программировать событие деления на ноль.

10 Вывод греческих букв, математических символов. Кодовая таблица Unicode

Хранение текстовых данных в памяти компьютера *предполагает кодирование символов по какому-либо принципу.* Таких кодировок несколько. Каждой кодировке соответствует своя таблица символов. В этой таблице каждой ячейке соответствуют номер в таблице и символ. Мы упомянем такие кодовые таблицы: ASCII, ANSI Cyrillic (другое название – Windows 1251), а также Unicode.

Первые две таблицы являются однобайтовыми, т. е. каждому символу соответствует 1 байт данных. Поскольку в 1 байте – 8 битов, байт может принимать 28 = 256 различных состояний, этим состояниям можно поставить в соответствие 256 разных символов. Так, в таблице ASCII от 0 до 127: базовая таблица – есть английские буквы, цифры, знаки препинания, управляющие символы. От 128 до 255: это расширенная таблица, в ней находятся русские буквы и символы псевдографики. Некоторые из этих символов соответствуют клавишам IBM-совместимых компьютеров. Еще эту таблицу называют "ДОСовской" по имени операционной системы MS-DOS, где она применяется. Эта кодировка встречается также в Интернете.

В операционной системе Windows используется преимущественно ANSI (Windows 1251). Базовые символы с кодами от 0 до 127 в этих таблицах совпадают, а расширенные – нет. То есть русские буквы в этих таблицах находятся в разных местах таблицы. Из-за этого бывают недоразумения. В ANSI нет символов псевдографики. Другое название кодовой таблицы Windows 1251.

В интернет существует также двухбайтовый стандарт Unicode. Здесь один символ кодируется двумя байтами. Размер такой таблицы кодирования – 216 = 65 536 ячеек. Кодовая таблица Unicode включает в себя практически все современные письменности. Когда в текстовом редакторе Word мы выполняем команду **Вставка | Символ**, то вставляем символ из таблицы Unicode. Также в **Блокноте** можно сохранять файлы в кодировке Unicode: **Сохранить как |**

Кодировка Юникод. В этом случае в **Блокноте** будут, например, греческие буквы, математические операторы греческие буквы, математические операторы Π, Δ, Σ и проч. Кстати, греческая буква Σ и математический оператор Σ занимают разные ячейки в Unicode. Размер файла при сохранении в Блокноте будет в два раза большим.

Напишем программу, которая приглашает пользователя ввести радиус *R,* чтобы вычислить длину окружности. При программировании этой задачи длину окружности в метке **Label** назовем греческой буквой β, приведем формулу для вычислений с греческой буквой π = 3,14. Результат вычислений выведем в диалоговое окно **MessageBox** также с греческой буквой.

После традиционного запуска Visual Studio и выбора шаблона **Приложение Windows Forms** (**Windows Forms Application C#)** перетащим в форму две метки **Label**, текстовое поле **TextBox** и командную кнопку **Button**. Посмотрите на рис. 16, так должна выглядеть форма после программирования этой задачи.

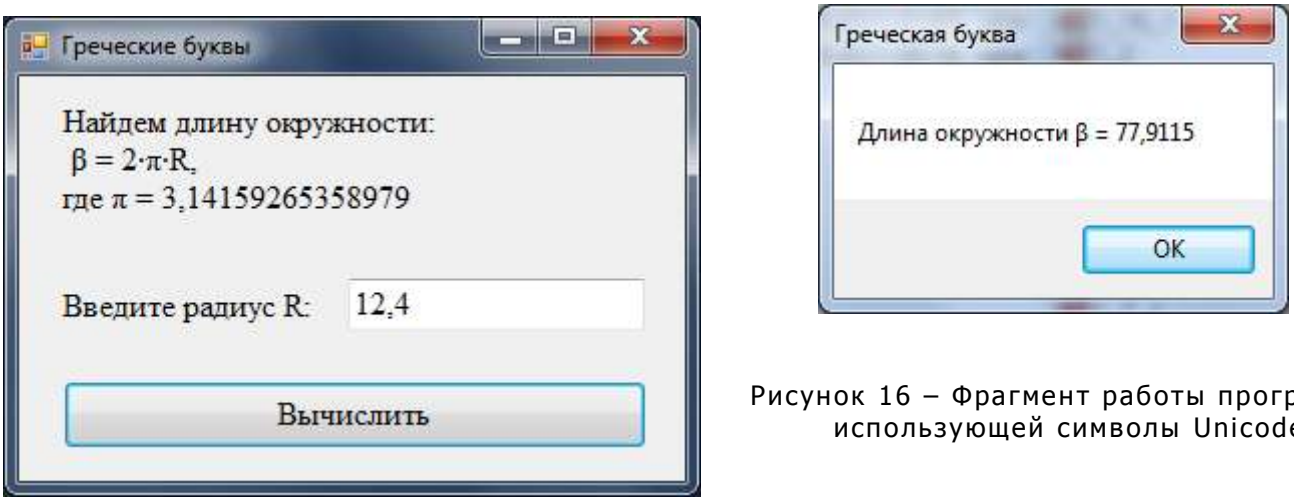


Рисунок 16 – Фрагмент работы программы, использующей символы Unicode

Вывод греческих букв на метку `label1` и в диалоговое окно `MessageBox` можно осуществить, например, таким путем. В текст программы через буфер обмена вставляем греческие буквы из текстового редактора MS Word. Поскольку по умолчанию Visual Studio 2010 сохраняет **cs**-файлы в формате Unicode, в принципе таким образом можно программировать вывод греческих букв и других символов на форму и на другие элементы управления.

Более технологично пойти другим путем, а именно будем вставлять подобные символы с помощью функции `Convert.ToChar`**,** а на вход этой функции будем подавать номер символа в таблице Unicode. В этом случае, даже если **cs**-файл будет сохранен в традиционной для Блокнота кодировке ANSI, программа будет работать корректно. Номер символа в таблице Unicode легко выяснить, выбрав в редакторе MS Word пункты меню **Вставка | Символ**. Здесь в таблице следует найти этот символ и соответствующий ему код знака в шестнадцатеричном представлении. Чтобы перевести шестнадцатеричное представление в десятичное, следует перед шестнадцатеричным числом поставить `0x`. Например, после выполнения оператора `n = 0x3B2` в переменной `n` будет записано десятичное число 946. На этом месте в таблице Unicode расположена греческая буква β.

Листинг 9. Использование символов Unicode

```
// Программа демонстрирует возможность вывода в текстовую метку, а также в
// диалоговое окно MessageBox греческих букв. Программа приглашает пользователя
// ввести радиус R, чтобы вычислить длину окружности

using System;
using System.Windows.Forms;
// Другие директивы using удалены, поскольку они не используются в данной программе

namespace Unicodes
{
   public partial class Form1 : Form
   {
```

```
        public Form1()
        { // Инициализация нового экземпляра класса System.Windows.Forms.Form
           InitializeComponent();
           base.Font = new System.Drawing.Font("Times New Roman", 12.0F);
           base.Text = "Греческие буквы";
           button1.Text = "Вычислить";
           // бета = 2 x PI x R
           label1.Text = string.Format(
            "Найдем длину окружности:\n {0} = 2{1}{2}{1}R,\nгде {2} = {3}",
             Convert.ToChar(0x3B2), Convert.ToChar(0x2219),
           //          0=бета             1-точка
            Convert.ToChar(0x3C0), Math.PI);
           //           2-Пи   3-число Пи
           label2.Text = "Введите радиус R:";
           textBox1.Clear();
        }
        private void button1_Click(object sender, EventArgs e)
        {  // Проверка - число ли введено:
           Single R; // - радиус
           bool Число_ли = Single.TryParse(textBox1.Text,
                    System.Globalization.NumberStyles.Number,
                    System.Globalization.NumberFormatInfo.CurrentInfo, out R);
           if (Число_ли == false)
           {
              MessageBox.Show("Следует вводить числа!", "Ошибка",
              MessageBoxButtons.OK, MessageBoxIcon.Error); return;
           }
           Single beta = 2 * (Single)Math.PI * R;
           // 0x3B2 - греческая буква бета
           MessageBox.Show(String.Format("Длина окружности {0} = {1:F4}",
                      Convert.ToChar(0x3B2), beta), "Греческая буква");
        }
     }
}
```

Как видно из программного кода, сразу после вызова процедуры InitializeComponent мы задали шрифт Times New Roman, 12 пунктов для формы, этот шрифт будет распространяться на все элементы управления на форме, т. е. на текстовое поле, метку и командную кнопку. Далее, используя метод String.Format**,** инициализировали свойство **Text** метки label1**.** Различные шестнадцатеричные номера соответствуют греческим буквам и арифметической операции "умножить", в инициализации строки участвует также константа $\pi = 3{,}14$. Ее *более* точное значение получаем из Math.PI**.** Escape-последовательность "\n" используем для переноса текста на новую строку. Так называемый перевод каретки можно осуществить также с помощью строки NewLine из перечисления Environment**.**

Обрабатывая событие button1_Click (щелчок на кнопке), мы проверяем с помощью метода TryParse**,** число ли введено в текстовое поле. Если пользователь ввел число **(**true**),** то метод TryParse возвращает значение радиуса R**.** При вычислении длины окружности beta приводим значение константы Math.PI из типа Double к типу Single посредством неявного преобразования.

После вычисления длины окружности beta выводим ее значение вместе с греческой буквой β – Convert.ToChar(0x3B2) в диалоговое окно MessageBox**.** Здесь используем метод String.Format**.** Выражение "{0:F4}" означает, что значение переменной beta следует выводить по фиксированному формату с четырьмя знаками после запятой.

Данная программа будет корректно отображать греческие буквы.