Лабораторная работа №2
Создание приложения Windows Form. Стандартные элементы управления и компоненты.

1. Цель

изучить методы создания приложений с использованием Windows Form. Также изучить стандартные компоненты и научиться их применять на практике.

2. Методические указания

1. При изучении языка программирования C# будет использоваться интегрированная среда разработки программного обеспечения Microsoft Visual Studio Express 2013 для Windows Desktop. Будут использованы основные элементы .NET Framework и связь C# с элементами платформы .NET.
2. По окончанию работы сохраните все созданные файлы на своих носителях.
3. Защита лабораторной работы производится только при наличии электронного варианта работающего скрипта.

3. Теоретические сведения

Первое, что необходимо сделать — создать приложение Windows Forms. Для примера создадим пустую форму, и отобразим ее на экране.

```
using System;
using System.Windows.Forms;
namespace NotepadForms
{
    public class MyForm : System.Windows.Forms.Form
    {
        public MyForm()
        {
        }
        [STAThread]
        static void Main()
        {
            Application.Run(new MyForm());
        }
    }
}
```

Когда мы скомпилируем и запустим этот пример, то получим маленькую пустую форму без заголовка. Никаких реальных функций, но это — Windows Forms. В приведенном коде заслуживают внимания две вещи. Первая — тот факт, что при создании класса **MyForm** используется наследование. Следующая строка объявляет **MyForm** как наследника System.Windows.Forms.Form:

public class MyForm : System.Windows.Forms.Form

Класс Form — один из главных классов в пространстве имен System.Windows.Forms. Следующий фрагмент кода стоит рассмотреть более подробно:

```
[STAThread]
static void Main()
{
    Application.Run(new MyForm());
}
```

Main — точка входа по умолчанию в любое клиентское приложение на C#. Как правило, в более крупных приложениях метод Main() не будет находиться в классе формы, а скорее в классе, отвечающем за процесс запуска. В данном случае вы должны установить имя такого запускающего класса в диалоговом окне свойств проекта. Обратите внимание на атрибут [STAThread]. Он устанавливает модель многопоточности COM в STA (однопоточный апартамент). Модель многопоточности STA требуется для взаимодействия с COM и устанавливается по умолчанию в каждом проекте Windows Forms.

Метод Application.Run() отвечает за запуск стандартного цикла сообщений приложения. Application.Run() имеет три перегрузки.

Первая из них не принимает параметров; вторая принимает в качестве параметра объект ApplicationContext. В нашем примере объект MyForm становится главной формой приложения. Это означает, что когда форма закрывается, то приложение завершается. Используя класс ApplicationContext, можно в большей степени контролировать завершение главного цикла сообщений и выход из приложения.

Класс Applicationсодержит в себе очень полезную функциональность. Он предоставляет группу статических методов и свойств для управления процессом запуска
и останова приложения, а также обеспечивает доступ к сообщениям Windows, обрабатываемым приложением. В табл. 2.1 перечислены некоторые из этих наиболее полезных методов и свойств.

Таблица 2.1. Некоторые полезные методы и свойства класса Application

Метод/свойство	Описание
`CommonAppDataPath`	Путь к данным, общий для всех пользователей приложения. Обычно это `БазовыйПуть\Название компании\Название продукта\Версия`, где `БазовыйПуть` — `C:\Documents and Settings\имя пользователя\ApplicationData`. Если путь не существует, он будет создан.
`ExecutablePath`	Путь и имя исполняемого файла, запускающего приложение.
`LocalUserAppDataPath`	Подобно `CommonAppDataPath`, но с тем отличием, что поддерживается роуминг (перемещаемость).
`MessageLoop`	`True` или `false` — в зависимости от того, существует ли цикл сообщений в текущем потоке.
`StartupPath`	Подобно `ExecutablePath`, с тем отличием, что имя файла не возвращается.
`AddMessageFilter`	Используется для предварительной обработки сообщений. Объект, реализующий `IMessageFilter`, позволяет фильтровать сообщения в цикле или организовать специальную обработку, выполняемую перед тем, как сообщение попадет в цикл.
`DoEvents`	Аналогично оператору `DoEvents` языка Visual Basic. Позволяет обработать сообщения в очереди.
`EnableVisualStyles`	Обеспечивает визуальный стиль Windows XP для различных визуальных элементов приложения. Существуют две перегрузки, принимающие информацию манифеста. Одна работает с потоком манифеста, вторая — принимает полное имя и путь файла манифеста.
`Exit` и `ExitThread`	`Exit` завершает текущий работающий цикл сообщений и вызывает выход из приложения. `ExitThread` завершает цикл сообщений и закрывает все окна текущего потока.

А теперь как будет выглядеть это приложение, если его сгенерировать в Visual Studio 2010? Первое, что следует отметить — будет создано два файла. Причина в том, что Visual Studio 2010 использует возможность частичных (partial) классов и выделяет весь код, сгенерированный визуальным дизайнером, в отдельный файл. Если используется имя по умолчанию — Form1, то эти два файла будет называться Form1.cs и Form1.Designer.cs. Если только у вас не включена опция Show All Files (Показать все файлы) в меню Project (Проект), то вы не увидите в проводнике Solution Explorer файла Form1.Designer.cs. Ниже показан код этих двух файлов, сгенерированных Visual Studio.
Сначала — Form1.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

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

Здесь мы видим только операторы using и простой конструктор. А вот код Form1.Designer.cs:

```
namespace MyForm
{
    partial class Form1
    {
        /// <summary>
        /// Требуется переменная конструктора.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Освободить все используемые ресурсы.
        /// </summary>
        /// <param name="disposing">истинно, если управляемый ресурс должен быть
            удален; иначе ложно.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Код, автоматически созданный конструктором форм Windows

        /// <summary>
        /// Обязательный метод для поддержки конструктора - не изменяйте
        /// содержимое данного метода при помощи редактора кода.
```

```
        /// </summary>
        private void InitializeComponent()
        {
            this.SuspendLayout();
            //
            // Form1
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(383, 335);
            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);

        }

        #endregion
    }
}
```

Файл, сгенерированный дизайнером форм, редко подвергается ручному редактированию. Единственным исключением может быть случай, когда необходима специальная обработка в методе Dispose(). Метод InitializeComponent мы обсудим позднее в этой главе. Если взглянуть на этот код примера приложения в целом, то мы увидим, что он намного длиннее, чем простой пример командной строки. Здесь перед началом класса присутствует несколько операторов using, и большинство из них в данном примере не нужны. Однако их присутствие ничем не мешает. Класс Form1 наследуется от System.Windows.Forms.Form, как и в предыдущем, введенном примере, но в этой точке начинаются расхождения. Во-первых, в файле Form1.Designer появляется строка:

private System.ComponentModel.IContainer components = null;

В данном примере эта строка кода ничего не делает. Но, добавляя компонент в форму, вы можете также добавить его в объект components, который представляет собой контейнер. Причина добавления этого контейнера — в необходимости правильной обработки уничтожения формы. Класс формы поддерживает интерфейс IDisposable, потому что он реализован в классе Component. Когда компонент добавляется в контейнер, то этот контейнер должен позаботиться о корректном уничтожении своего содержимого при закрытии формы. Это можно увидеть в методе Dispose нашего примера:

```
    protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }
```

Здесь мы видим, что когда вызывается метод Dispose формы, то также вызывается метод Dispose объекта components, поскольку он содержит в себе другие компоненты, которые также должны быть корректно удалены.

Конструктор класса Form1, находящийся в файле Form1.cs, выглядит так:

```
        public Form1()
        {
            InitializeComponent();
        }
```

Обратите внимание на вызов InitializeComponent().

Метод InitializeComponent() находится в файле Form1.Designer.cs и делает то, что следует из его наименования — инициализирует все элементы управления, которые могут быть добавлены к форме. Он также инициализирует свойства формы. В нашем примере метод InitializeComponent() выглядит следующим образом:

```
private void InitializeComponent()
{
    this.SuspendLayout();
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(383, 335);
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
}
```

Как видите, здесь присутствует лишь базовый код инициализации. Этот метод связан с визуальным дизайнером Visual Studio. Когда в форму вносятся изменения в дизайнере, они отражаются на InitializeComponent(). Если вы вносите любые изменения в InitializeComponent(), то следующий раз после того, как что-то будет изменено в дизайнере, эти ручные изменения будут утеряны. InitializeComponent() повторно генерируется после каждого изменения дизайна формы. Если возникает необходимость добавить некоторый дополнительный код для формы или элементов управления и компонентов формы, это должно быть сделано после вызова InitializeComponent(). Этот метод также отвечает за создание экземпляров элементов управления, поэтому любой вызов, ссылающийся на них, выполненный до InitializeComponent(), завершится возбуждением исключения нулевой ссылки.

Чтобы добавить элемент управления или компонент в форму, нажмите комбинацию клавиш <Ctrl+Alt+X> или выберите пункт меню View - Toolbox (Вид - Панель инструментов) в среде Visual Studio .NET. Щелкните правой кнопкой мыши на Form1.cs в проводнике Solution Explorer и в появившемся контекстном меню выберите пункт View Designer (Показать дизайнер). Выберите элемент управления Button и перетащите на поверхность формы в визуальном дизайнере. Можно также дважды щелкнуть на выбранном элементе управления, и он будет добавлен в форму. То же самое следует проделать с элементом TextBox рис. 1.1. Теперь, после добавления этих двух элементов управления на форму, метод InitializeComponent() расширяется и содержит такой код:

```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    //
    // button1
    //
    this.button1.Location = new System.Drawing.Point(166, 44);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 0;
    this.button1.Text = "button1";
    this.button1.UseVisualStyleBackColor = true;
    //
    // textBox1
    //
    this.textBox1.Location = new System.Drawing.Point(43, 44);
    this.textBox1.Name = "textBox1";
```

```
        this.textBox1.Size = new System.Drawing.Size(100, 20);
        this.textBox1.TabIndex = 1;
        //
        // Form1
        //
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(383, 335);
        this.Controls.Add(this.textBox1);
        this.Controls.Add(this.button1);
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);
        this.PerformLayout();

}
```

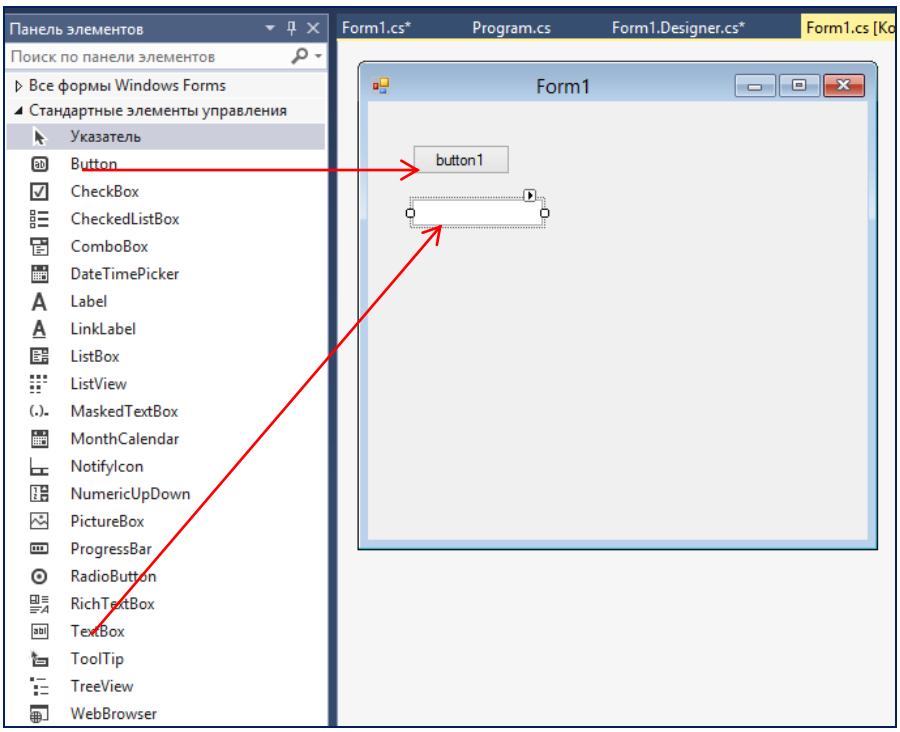


Рис. 1.1. Расположение элементов на форме

Если посмотреть на первые три строки кода этого метода, мы увидим в них создание экземпляров элементов управления Button и TextBox. Обратите внимание на имена, присвоенные им — textBox1и button1. По умолчанию дизайнер в качестве имен использует имя класса элемента управления, дополненное целым числом. Когда вы добавляете следующую кнопку, дизайнер называет ее button2 и т.д. Следующая строка — часть пары SuspendLayout/ResumeLayout. Метод SuspendLayout() временно приостанавливает события размещения, которые имеют место при первоначальной инициализации элемента управления. В конце метода вызывается ResumeLayout(), чтобы вернуть все в норму. В сложной форме с множеством элементов управления метод InitializeComponent() может стать достаточно большим.

Чтобы изменить значения свойств элемента управления, нужно либо нажать <F4>, либо выбрать пункт меню View-Properties Window (Вид - Окно свойств). Это окно позволяет модифицировать большинство свойств элемента управления или компонента. При внесении изменений в окне свойств метод InitializeComponent() автоматически переписывается с тем, чтобы отразить новые значения свойств. Например, изменив свойство Text на My Button в окне свойств, получим следующий код в InitializeComponent():

```
//
// button1
//
this.button1.Location = new System.Drawing.Point(34, 33);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "My Button ";
this.button1.UseVisualStyleBackColor = true;
```

Иерархия классов

Важность понимания иерархии становится очевидной в процессе проектирования и конструирования пользовательских элементов управления. Если такой элемент управления унаследован от конкретного библиотечного элемента управления, например, когда создается текстовое поле с некоторыми дополнительными методами и свойствами, то имеет смысл унаследовать его от обычного текстового поля и затем переопределить и добавить необходимые методы. Однако если приходится создавать элемент управления, который не соответствует ни одному из существующих в .NET Framework, то его придется унаследовать от одного из базовых классов: Control или ScrollableControl, если нужны возможности прокрутки, либо ContainerControl, если он должен служить контейнером для других элементов управления.

Класс Control

Пространство имен System.Windows.Forms включает один класс, который является базовым почти для всех элементов управления и форм — System.Windows.Forms.Control. Он реализует основную функциональность для создания экранов, которые видит пользователь. Класс Control унаследован от System.ComponentModel.Component. Класс Component обеспечивает классу Control инфраструктуру, необходимую для того, чтобы его можно было перетаскивать и помещать на поле дизайнера, а также, чтобы он мог включать в себя другие элементы управления. Класс Control предлагает огромный объем функциональности классам, наследуемым от него. Этот список слишком большой, чтобы приводить его здесь, поэтому в данном разделе мы рассмотрим только наиболее важные возможности, предоставляемые классом Control.

Размер и местоположение

Размер и местоположение элементов управления определяются свойствами Height, Width, Top, Bottom, Left и Right, вместе с дополняющими их Size и Location. Отличие состоит в том, что Height, Width, Top, Bottom, Leftи Right принимают одно целое значение. Size принимает значение структуры Size, а Location — значение структуры Point. Структуры Size и Point включают в себя координаты X, Y. Point обычно описывает местоположение, а Size— высоту и ширину объекта. Size и Point определены в пространстве имен System.Drawing. Обе структуры очень похожи в том, что представляют пары координат X, Y, но, кроме того — переопределенные операции, упрощающие сравнения и преобразования. Вы можете, например, складывать вместе две структуры Size. В случае структуры Point операция сложения переопределена таким образом, что можно прибавить к Point структуру Size и получить в результате Point. Это дает эффект прибавления расстояния к местоположению, чтобы получить новое местоположение, что очень удобно для динамического создания форм и элементов управления.

Свойство Bounds возвращает объект Rectangle, представляющий экранную область, занятую элементом управления. Эта область включает полосы прокрутки и заголовка. Rectangle также относится к пространству имен System.Drawing. Свойство ClientSize — структура Size, представляющая клиентскую область элемента управления за вычетом полос прокрутки и заголовка.

Методы PointToClient и PointToScreen— удобные методы преобразования, которые принимают Point и возвращают Point. Метод PointToClient принимает структуру Point, представляющую экранные координаты, и транслирует их в координаты текущего клиентского объекта. Это удобно для операций перетаскивания. Метод PointToScreen выполняет обратную операцию — принимает координаты в клиентском объекте и транслирует их в экранные координаты.

Методы RectangleToScreen и ScreenToRectangle выполняют те же операции, но со структурами Rectangle вместо Point.

Свойство Dock определяет, к какой грани родительского элемента управления должен пристыковываться данный элемент. Перечисление DockStyle задает возможные значения этого свойства. Они могут быть такими: Top, Bottom, Right, Left, Fill и None. Значение Fill устанавливает размер данного элемента управления равным размеру родительского.

Свойство Anchor(якорь) прикрепляет грань данного элемента управления к грани родительского элемента управления. Это отличается от стыковки (docking) тем, что не устанавливает грань дочернего элемента управления точно на грань родительского, а просто выдерживает постоянное расстояние между ними. Например, если якорь правой грани элемента управления установлен на правую грань родительского элемента, и если родитель изменяет размер, то правая грань данного элемента сохраняет постоянную дистанцию от правой грани родителя — т.е. он изменяет размер вместе с родителем. Свойство Anchor принимает значения из перечисления AnchorStyle, а именно: Top, Bottom, Left, Right и None. Устанавливая эти значения, можно заставить элемент управления изменять свой размер динамически вместе с родителем. Таким образом, кнопки и текстовые поля не будут усечены или скрыты при изменении размеров формы пользователем.

Свойства Dock и Anchor применяются в сочетании с компоновками элементов управления Flow и Table они позволяют создавать очень сложные пользовательские окна. Изменение размеров окна может оказаться достаточно непростой задачей для сложных форм с множеством элементов управления. Эти инструменты существенно облегчают задачу.

Внешний вид

Свойства, имеющие отношение к внешнему виду элемента управления — это BackColor и ForeColor, которые принимают объект System.Drawing.Color в качестве значения. Свойство BackGroundImage принимает объект графического образа как значение. Класс System.Drawing.Image — абстрактный класс, служащий в качестве базового для классов Bitmap и Metafile. Свойство BackgroundImageLayout использует перечисление ImageLayout для определения способа отображения графического образа в элементе управления. Допустимые значения таковы: Center, Tile, Stretch, Zoom или None.

Свойства Font и Text работают с надписями. Чтобы изменить Font, необходимо создать объект Font. При создании этого объекта указывается имя, стиль и размер шрифта.

Взаимодействие с пользователем

Взаимодействие с пользователем лучше всего описывается серией событий, которые генерирует элемент управления и на которые он реагирует. Некоторые из наиболее часто используемых событий: Click, DoubleClick, KeyDown, KeyPress, Validatingи Paint. События, связанные с мышью — Click, DoubleClick, MouseDown, MouseUp, MouseEnter, MouseLeave и MouseHover— описывают взаимодействие мыши и экранного элемента управления. Если вы обрабатываете оба события — Click и DoubleClick— то всякий раз, когда перехватывается событие DoubleClick, также возбуждается и событие Click. Это может привести к нежелательным последствиям при неправильной обработке. К тому же и Click, и DoubleClick принимают в качестве аргумента EventArgs, в то время как события MouseDown и MouseUp принимают MouseEventArgs. Структура MouseEventArgs содержит несколько частей полезной информации — например, о кнопке, на которой был выполнен щелчок, количестве щелчков на

кнопке, количестве щелчков колесика мыши, текущих координатах X и Y указателя мыши. Если нужен доступ к любой подобной информации, то вместо событий Click или DoubleClick потребуется обрабатывать события MouseDown и MouseUp. События клавиатуры работают подобным образом. Объем необходимой информации определяет выбор обрабатываемых событий. Для простейших случаев событие KeyPress принимает KeyPressEventArgs. Эта структура включает KeyChar, представляющий символ нажатой клавиши. Свойство Handled используется для определения того, было ли событие обработано. Установив значение Handled в true, можно добиться того, что событие не будет передано операционной системе для совершения стандартной обработки. Если необходима дополнительная информация о нажатой клавише, то больше подойдут события KeyDown или KeyUp. Оба принимают структуру KeyEventArgs. Свойства KeyEventArgs включают признак одновременного состояния клавиш <Ctrl>, <Alt> или <Shift>. Свойство KeyCode возвращает значение типа перечисления Keys, идентифицирующее нажатую клавишу. В отличие от свойства KeyPressEventArgs. KeyChar, свойство KeyCode сообщает о каждой клавише клавиатуры, а не только о буквенно-цифровых клавишах. Свойство KeyData возвращает значение типа Keys, а также устанавливает модификатор. Значение модификатора сопровождает значение клавиши, объединяясь с ним двоичной логической операцией “ИЛИ”. Таким образом, можно получить информацию о том, была ли одновременно нажата клавиша <Shift>или <Ctrl>. Свойство KeyValue— целое значение из перечисления Keys. Свойство Modifiers содержит значение типа Keys, представляющее нажатые модифицирующие клавиши. Если было нажато более одной такой клавиши, их значения объединяются операцией “ИЛИ”. События клавиш поступают в следующем порядке:

1. KeyDown
2. KeyPress
3. KeyUp

События Validating, Validated, Enter, Leave, GotFocusи LostFocus имеют отношение к получению фокуса элементами управления (т.е. когда становятся активными) и утере его. Это случается, когда пользователь нажатием клавиши <Tab> переходит к данному элементу управления либо выбирает его мышью. Может показаться, что события Enter, Leave, GotFocusи LostFocus очень похожи в том, что они делают. События GotFocus и LostFocus относятся к низкоуровневым, и связаны с событиями Windows WM_SETFOCUS и WM_KILLFOCUS. Обычно когда возможно, лучше использовать события Enter и Leave. События Validatin gи Validated возбуждаются при проверке данных в элементе управления. Эти события принимают аргумент CancelEventArgs. С его помощью можно отменить последующие события, установив свойство Cancel в true. Если вы разрабатываете собственный проверочный код, и проверка завершается неудачно, то в этом случае можно установить Cancel в true — тогда элемент управления не утратит фокус. Validating происходит во время проверки, а Validated — после нее. Порядок возникновения событий следующий:

1. Enter
2. GotFocus
3. Leave
4. Validating
5. Validated
6. LostFocus

Понимание последовательности этих событий важно, чтобы избежать рекурсивных ситуаций. Например, попытка установить фокус элемента управления внутри обработчика события LostFocus создает ситуацию взаимоблокировки в цикле событий, и приложение перестает реагировать на внешние воздействия.

Стандартные элементы управления и компоненты

Рассмотрим различные стандартные элементы управления, поставляемые в составе .NET Framework, и объясним, какую дополнительную функциональность они предлагают.

Форма frmControls, содержит многие элементы управления с базовой функциональностью. На рис. 2.2 показан ее внешний вид.

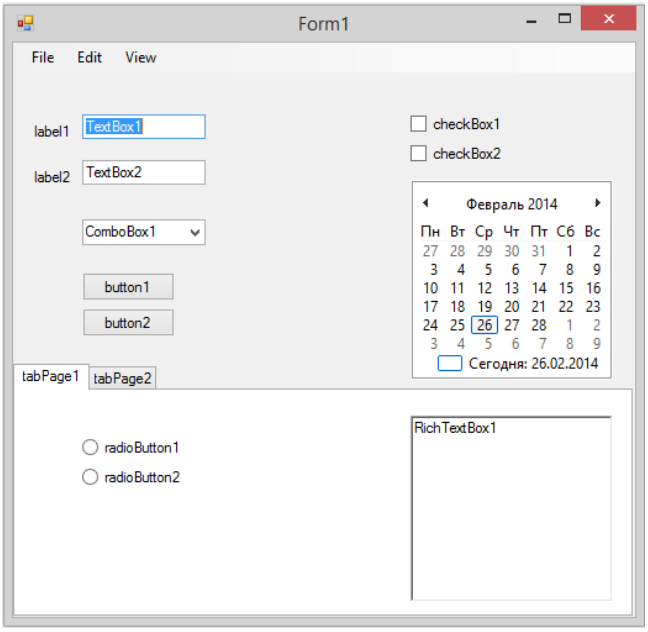


Рис. 2.2 Форма с основными элементами

Button

Класс Button представляет простую командную кнопку и наследуется от ButtonBase. Чаще всего требует написания кода обработки события Click. Следующий фрагмент кода реализует обработчик события Click. Когда выполняется щелчок на кнопке, появляется окно сообщения, отображающее имя кнопки.

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Выполнен щелчок на " + ((Button)sender).Name + ".");
}
```

С помощью метода PerformClick можно эмулировать событие Click кнопки без необходимости действительного выполнения щелчка пользователем. Метод NotifyDefault принимает в параметре значение булевского типа и сообщает кнопке, чтобы она отобразила себя как кнопку по умолчанию. Обычно кнопка по умолчанию на форме имеет слегка утолщенную рамку. Чтобы идентифицировать кнопку как кнопку по умолчанию, потребуется установить свойство AcceptButton формы равным ссылке на эту кнопку. После этого, если пользователь нажмет клавишу <Enter>, сгенерируется событие Click этой кнопки по умолчанию. На рис. 2.3 Кнопка с меткой Default (По умолчанию) является кнопкой по умолчанию (обратите внимание на темную рамку). Кнопки могут содержать на своей поверхности как текст, так и графическое изображение. Изображения доступны для кнопок через объект ImageList или свойство Image.

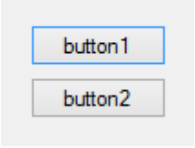


Рис. 2.3 Кнопка с меткой Default

Объект ImageList представляет собой именно то, что можно предположить по его названию: список изображений, который управляется компонентом, помещенным на форму. Как Text, так и Image имеют свойство Align, предназначенное для выравнивания текста или изображения на поверхности кнопки. Свойство Align принимает значения типа перечисления ContentAlignment. Текст или изображение могут быть выровнены в комбинации по левой или правой границе кнопки либо по верхней или нижней границе.

CheckBox

Элемент управления CheckBox (флажок) также унаследован от ButtonBase и применяется для принятия команды пользователя с двумя или тремя состояниями. Если свойство ThreeState установлено в true, то свойство CheckState элемента CheckBox может принимать одно из следующих трех перечислимых значений:

- Checked — Элемент CheckBox отмечен.
- Unchecked — Элемент CheckBox не отмечен.
- Indeterminate — В этом состоянии элемент CheckBox не доступен.

Состояние Indeterminate может быть установлено только программно, а не пользователем. Это удобно, если вы хотите сообщить пользователю, что опция не была установлена. Для получения текущего состояния в виде булевского значения можно обратиться к свойству Checked.

События CheckedChanged и CheckStateChanged возникают, когда изменяется свойство CheckState или Checked. Перехват этих событий может пригодиться для установки других значений на основе нового состояния CheckBox. В классе формы frmControls событие CheckedChanged для нескольких элементов CheckBox обрабатывается следующим методом:

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
    CheckBox checkBox = (CheckBox)sender;
    MessageBox.Show("Новое значение " + checkBox.Name + " равно " +
    checkBox.Checked.ToString());
}
```

При изменении состояния каждого из этих элементов отображается окно сообщения с именем элемента CheckBox и его новым состоянием.

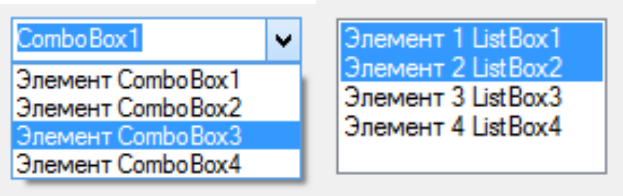
RadioButton

Последний элемент управления, унаследованный от ButtonBase — это RadioButton (переключатель). Переключатели обычно используются в составе групп. Иногда называемые кнопками выбора (option buttons), переключатели дают возможность пользователю выбирать одну из нескольких опций. Когда вы используете множество элементов управления RadioButton в одном контейнере, выбранным может быть только один из них. Поэтому если у вас есть три опции, например, Red, Green и Blue, и если выбрана опция Red, а пользователь щелкает на Blue, то Red автоматически отключается. Свойство Appearance принимает значение из перечисления Appearance. Оно может быть либо Button, либо Normal. Когда выбирается Normal, то переключатель выглядит как маленький кружок с меткой рядом с ним. Выбор его заполняет кружок, выбор другого переключателя из той же группы отменяет выбор текущего выбранного переключателя и делает его кружок пустым. При установке значения Appearance равным Button переключатель выглядит подобно стандартной кнопке, но работает подобно переключателю — выбранная кнопка нажата, не выбранная — отпущена. Свойство CheckedAlign определяет, где находится кружок по отношению к тексту метки. Он может быть над текстом, под ним, справа или слева. Событие CheckedChanged возникает всякий раз, когда значение свойства Checked

изменяется. Подобным образом можно выполнить другие действия на основе нового значения элемента управления.

ComboBox, ListBox и CheckedListBox

ComboBox, ListBoxи CheckedListBox — все унаследованы от класса ListControl. Этот класс определяет некоторую базовую функциональность управления списками. Самое главное в использовании списочных элементов управления — это добавление и выбор элементов списка. То, какой список нужно применять, в основном определяется тем, как его предполагается использовать, и типом данных, которые в нем должны содержаться. Если необходимо иметь возможность множественного выбора, или пользователю нужно видеть в любой момент несколько позиций списка, то лучше всего подойдут ListBox или CheckListBox. Если же в списке может быть выбран только один элемент за раз, то больше подойдет ComboBox.



Прежде чем списком можно будет пользоваться, к нему нужно добавить данные. Это делается добавлением объектов в ListBox.ObjectCollection. Эта коллекция представлена свойством списка Items. Поскольку коллекция сохраняет объекты, любой корректный тип .NET может быть добавлен в список. Для того чтобы идентифицировать элементы, необходимо установить два важных свойства. Первое из них — DisplayMember. Эта установка сообщает ListControl, какое свойство вашего объекта должно быть отображено в списке. Второе — ValueMember, оно указывает свойство вашего объекта, которое нужно возвратить в качестве его значения. Если в список добавляются строки, то по умолчанию они и используются для обоих этих свойств. Форма frmListsв примере приложения показывает, как и объекты, и строки (которые, разумеется, тоже представляют собой объекты) могут быть загружены в окно списка. В примере в качестве данных списка применяются объекты Vendor. После того, как данные загружены в список, для их получения можно использовать свойства SelectedItem и SelectedIndex. Свойство SelectedItem возвращает текущий выбранный объект. Если список разрешает множественный выбор, нет гарантии того, какой именно элемент будет возвращен. В этом случае должна использоваться коллекция SelectObject. Она содержит список всех текущих выбранных элементов списка. Если нужно получить элемент по определенному индексу, то свойство Items может быть использовано для доступа к ListBox.ObjectCollection.

ErrorProvider

ErrorProvider — на самом деле не элемент управления, а компонент. Когда вы перетаскиваете компонент в дизайнер форм, он отображается в лотке компонентов под дизайнером. Назначение ErrorProvider заключается в том, чтобы высвечивать пиктограмму рядом с элементом управления, когда возникает ошибочная ситуация или не проходит проверка. Предположим, что у вас есть поле TextBox, предназначенное для ввода возраста. Ваше бизнес-правило гласит, что значение возраста не должно превышать 65. Если пользователь попытается ввести большее значение, его нужно будет информировать, что введен возраст, превышающий допустимый, и это следует исправить. Проверка правильности введенного значения выполняется в обработчике события Validated этого текстового поля. Если проверка не прошла, можно вызвать метод SetError, передав ссылку на тот элемент управления, который вызвал ошибку, и когда пользователь наведет курсор мыши на пиктограмму, будет отображен текст сообщения об ошибке. На рис. 2.4 показана пиктограмма, которая появляется в случае ввода в текстовое поле недопустимого значения. Вы можете создать ErrorProvider для каждого элемента управления на форме, который может быть причиной ошибки, но если у вас очень много элементов управления, это может оказаться

слишком громоздко. Другой вариант — использовать один поставщик ошибок, и в событии проверки вызывать метод IconLocation с тем элементом управления, который вызвал проверку, и одним из значений перечисления ErrorIconAlignment. Это значение устанавливает выравнивание пиктограммы по элементу управления. Затем следует вызвать метод SetError. Если нет никаких ошибочных условий, можно очистить ErrorProvider, вызвав SetErrorc пустой строкой ошибки. В следующем примере показано, как это работает.

```
private void textBox1_Validated(object sender, EventArgs e)
  {
      if (textBox1.TextLength > 0 && Convert.ToInt32(textBox1.Text) > 65)
      {
          errorProvider1.SetError(this.textBox1, String.Empty);
      }
      else
      {
          errorProvider1.SetError(this.textBox1, "Name is required.");
      }

  }
```

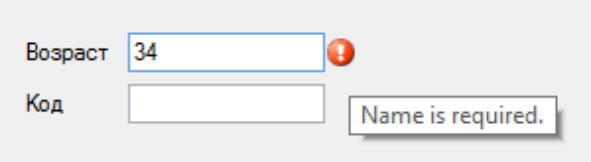


Рис. 2.4. Пиктограмма, которая появляется в случае ввода в текстовом поле недопустимого значения

Если проверка не проходит (например, в **Возраст** введено число меньше 65), вызывается метод SetIcon поставщика ошибок errMain. Он устанавливает пиктограмму рядом с элементом управления, не прошедшим проверку. Тут же устанавливается текст ошибки, так что когда пользователь наведет курсор мыши на эту пиктограмму, то увидит сообщение, информирующее его о том, что является причиной неудачной проверки.

PictureBox

Элемент управления PictureBox применяется для отображения графических изображений. Изображение может быть в формате BMP, JPEG, GIF, PNG, метафайла или пиктограммы. Свойство SizeMode использует перечисление PictureBoxSizeMode для определения того, как изображение размещается в элементе управления. SizeMode может быть равно AutoSize, CenterImage, Normal или StretchImage. Размер отображения PictureBo хможно изменять, устанавливая свойство ClientSize. При создании PictureBox сначала создается объект, базирующийся на Image.

ProgressBar

Элемент управления ProgressBar (индикатор хода работ) используется для визуального представления состояния длительного действия. Он уведомляет пользователя, что нечто происходит, поэтому следует подождать. Для элемента управления ProgressBar устанавливаются значения свойств Minimum и Maximum. Эти свойства соответствуют положению индикатора хода работ в крайнем левом (Minimum) и крайнем правом (Maximum) положениях. Свойство Step устанавливает число, на которое увеличивается значение при каждом вызове метода PerformStep. Можно также использовать метод Increment и увеличивать значение на переданную ему величину. Свойство Value возвращает текущее значение ProgressBar. С помощью свойства Text можно информировать пользователя о процентной доле выполнения работы или же о количестве оставшихся до ее завершения

позиций. Имеется также свойство BackgroundImage, предназначенное для настройки внешнего вида индикатора выполнения.

Label

Метки Label применяются для представления пользователю описательного текста. Текст может иметь отношение к другому элементу управления либо к текущему состоянию системы. Обычно метки помещаются рядом с текстовыми полями. Метка предлагает пользователю описание типа данных для ввода в текстовое поле. Элемент управления Label всегда доступен только для чтения — пользователь не может изменить значение строки в его свойстве Text. Однако вы можете изменять значение свойства Text программно. Свойство UseMnemonic позволяет включить функциональность клавиши доступа. Когда букве в свойстве Text предшествует символ амперсанда (&), эта буква высвечивается с подчеркиванием. Нажатие клавиши <Alt> в сочетании с клавишей этой буквы устанавливает фокус на следующий (в порядке обхода) после метки элемент управления. Если свойство Text уже содержит в тексте амперсанд, то добавление второго не вызовет подчеркивания буквы. Например, если текстом метки должно быть Nuts & Bolts, то свойство должно иметь значение Nuts && Bolts. Поскольку элемент управления Label доступен только для чтения, он не может получать фокус — вот почему фокус передается следующему доступному элементу управления. По этой причине важно помнить, что если вы используете мнемонику (т.е. клавишу быстрого доступа), нужно правильно устанавливать в форме порядок обхода с помощью клавиши табуляции. Свойство AutoSize содержит булевское значение, указывающее на то, что Label может автоматически изменять свой размер в соответствии со значением текста метки. Это может быть удобно для многоязычных приложений, где длина свойства Text изменяется в зависимости от текущего языка.

TextBox, RichTextBox и MaskedTextBox

Элемент управления TextBox— один из наиболее часто используемых. TextBox, RichTextBox и MaskedTextBoxy наследованы от TextBoxBase. Класс TextBoxBase представляет такие свойства, как MultiLine и Lines. Свойство MultiLine — булевское значение, позволяющее элементу управления TextBox отображать текст в более чем одной строке. При этом каждая строка в текстовом окне является частью массива строк. Этот массив доступен через свойство Lines. Свойство Text возвращает полное содержимое текстового окна в виде одной строки. TextLength— общая длина текста. Свойство MaxLength ограничивает длину текста определенной величиной. SelectedText, SelectionLengthи SelectionStart имеют дело с текущим выделенным текстом в текстовом окне. Выделенный текст подсвечивается, когда элемент управления получает фокус. TextBox добавляет множество интересных свойств. AcceptsReturn— булевское значение, позволяющее TextBox воспринимать клавишу <Enter> как символ новой строки либо активизировать кнопку по умолчанию на форме. Когда это свойство имеет значение true, то нажатие <Enter> создает новую строку в TextBox. Свойство CharacterCasingo пределяет регистр текста в текстовом окне. Перечисление CharacterCasing содержит три значения: Lower, Normalи Upper. Значение Lower переводит в нижний регистр весь текст, независимо от того, как он был введен, Upper переводит весь текст в верхний регистр, а Normal отображает текст так, как он был введен. Свойство PasswordChar позволяет указать символ, который будет отображаться при вводе пользователем всех символов в текстовом окне. Это применяется при вводе паролей и PIN-кодов. Свойство text вернет действительный введенный текст; свойство PasswordChar касается только отображения символов.

RichTextBox — элемент управления, служащий для редактирования текста с расширенными возможностями форматирования. Как следует из его названия, RichTextBox использует Rich Text Format (RTF) для обработки специального форматирования. Изменения формата обеспечивается свойствами SelectionFont, SelectionColor и SelectionBullet, а форматирование параграфов — свойствами SelectionIndent, SelectionRightIndent и SelectionHangingIndent. Все

свойства их группы Selection работают одинаково. Если выделена часть текста, то изменение свойства касается этого выделенного фрагмента. Если же выделенного фрагмента нет, то изменения затрагивают любой текст, вставляемый справа от текущей позиции вставки. Текст данного элемента управления может быть извлечен из свойства Text либо Rtf. Свойство Text возвращает простой текст элемента управления, в то время как Rtf — форматированный текст.

Panel

Panel— простой элемент управления, содержащий в себе другие элементы управления. За счет группирования вместе элементов управления и помещения их в панель существенно упрощается управление ими. Например, можно сделать недоступными все элементы управления в панели, просто сделав недоступной всю панель. Поскольку Panel наследуется от ScrollableControl, также можно воспользоваться преимуществами AutoScroll. Если в пределах доступной области нужно отобразить слишком много элементов управления, поместите их в панель и установите значение true свойству AutoScroll— после этого их можно будет прокручивать в пределах этой области. Панели по умолчанию не отображают рамки, но, присвоив значение свойству BorderStyle, можно визуально группировать взаимосвязанные элементы управления посредством рамок. Это делает пользовательский интерфейс более дружественным.

TabControl u TabPages

TabControl позволяет группировать связанные элементы управления в серии страниц-вкладок. TabControl управляет коллекцией элементов типа TabPages. Несколько свойств управляют внешним видом TabControl. Свойство Appearance использует перечисление TabAppearance для определения внешнего вида вкладок. Допустимыми значениями являются FlatButtons, Buttonsи Normal. Свойство Multiline булевского типа указывает на то, что может отображаться более одной строки вкладок. Если свойство Multiline установлено в false, а количество вкладок превышает такое, что не может уместиться на экране, появляется пара кнопок, позволяющая прокручивать вкладки и видеть те, что не уместились. Свойство Text элемента TabPage— это то, что отображается на отдельной вкладке. Свойство Text устанавливается через параметр конструктора. Создав элемент управления TabPage, вы получаете контейнер, куда можно помещать другие элементы управления. Средствами дизайнера Visual Studio .NET легко добавить элемент TabPage к элементу управления TabControl, используя редактор коллекций. При добавлении каждой такой страницы можно установить множество ее свойств, затем перетащить на нее другие дочерние элементы управления. Получить текущую вкладку можно из свойства SelectedTab. Событие SelectedIndex возникает при каждом переключении вкладки. Прослушивая свойство SelectedIndex и затем подтверждая текущий выбор страницы через SelectedTab, вы можете организовать специальную обработку для каждой вкладки. Вы могли бы, к примеру, управлять данными, отображаемыми для каждой вкладки.

ContextMenuStrip

Класс ContextMenuStrip применяется для показа контекстного меню, или меню, отображаемого по нажатию правой кнопки мыши. Подобно MenuStrip, ContextMenuStrip является контейнером объектов ToolStripMenuItem. Однако он унаследован от ToolStripDropDownMenu. Элемент ContextMenuStrip создается так же, как MenuStrip. К нему добавляются элементы ToolStripMenuItemи определяются события Click каждого элемента для выполнения специфического действия. Контекстное меню назначается конкретному элементу управления. Это делается установкой свойства ContextMenuStrip элемента управления. Когда пользователь щелкает правой кнопкой мыши в поле элемента управления, отображается упомянутое меню.

4. Содержание отчёта

Отчёт должен содержать название и цель. Выполненные задания из ПРИЛОЖЕНИЕ А. Выводы.

5. Контрольные вопросы

1. Особенности построения приложений на C#?
2. Для чего используется компонент Label;
3. Для чего используется компонент Button;
4. Для чего используется компонент CheckBox;
5. Для чего используется компонент ErrorProvider;
6. Для чего используется компонент ListBox;
7. Для чего используется компонент TabControl;
8. Для чего используется компонент Panel;
9. Для чего используется компонент RadioButton;
10. Какие компоненты используются для создания меню на форме;

ПРИЛОЖЕНИЕ А

1. Создать калькулятор, который полностью повторяет обычный калькулятор Windiws. Интерфейс должен быть аналогичен рис. 2.5. Необходимо, чтоб программа отображала ошибки пользователя.

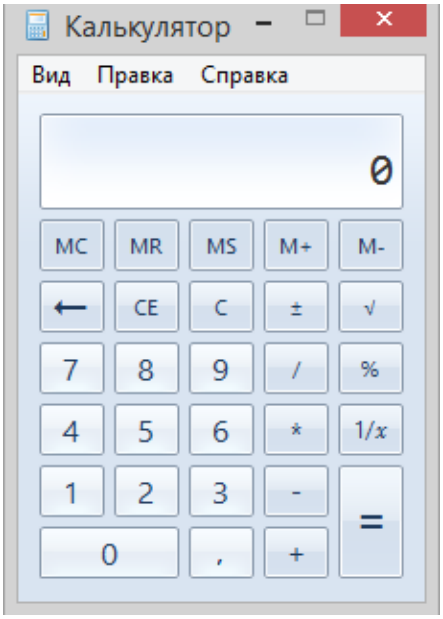


Рис. 2.5. Обычный калькулятор Windows

2. Добавить в калькулятор функции согласно варианту и вид калькулятора должен изменяться по событию, на вид рис. 2.6. Используйте для решения поставленной задачи два дополнительных компонента из библиотеки, которые не изображены на рис. 2.6. Необходимо, чтобы программа отображала ошибки пользователя с использованием ErrorProvider.

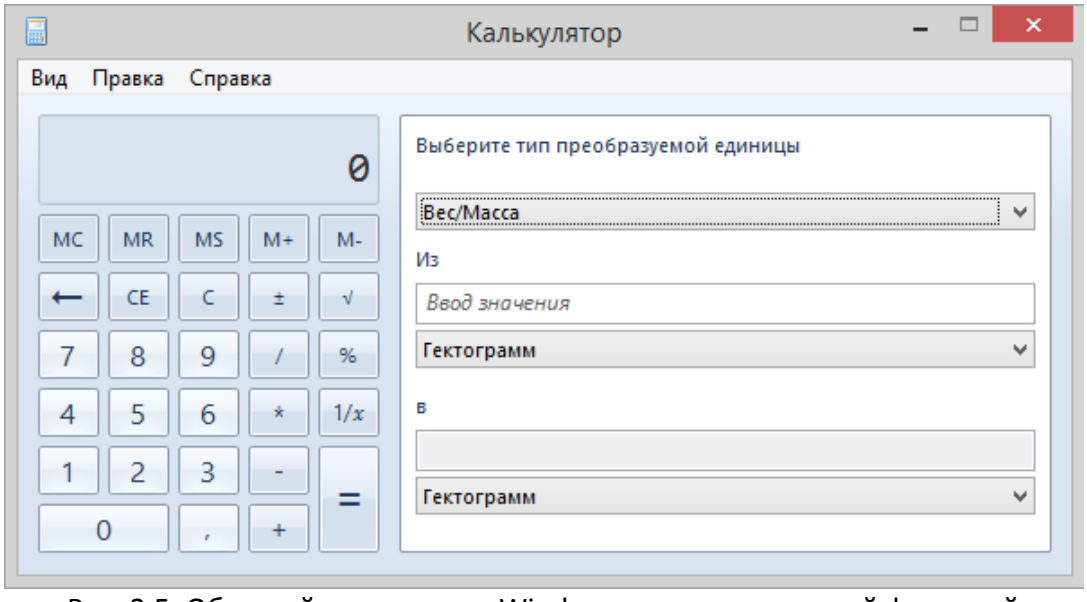


Рис. 2.5. Обычный калькулятор Windows с дополнительной функцией

№ ВАРИАНТА	ФУНКЦИЯ
1	Добавить две кнопки для расчета X^2, %
2	Добавить две кнопки для расчета 1/x, cos(x)

3	Добавить две кнопки для расчета X^3, 10^x
4	Добавить две кнопки для расчета X^y, Pi
5	Добавить две кнопки для расчета sin(x), n!
6	Добавить функцию преобразования величин Вес (Граммы в Караты) и обратно.
7	Добавить функцию преобразования величин Время (Год в Минуты) и обратно.
8	Добавить функцию преобразования величин Давление (Атмосферы в Бары) и обратно.
9	Добавить функцию преобразования величин Длина (Сантиметры в Дюймы) и обратно.
10	Добавить функцию преобразования величин Мощность (Ватт в Киловатт) и обратно.
11	Добавить функцию преобразования величин Объем (Литр в Британскую пинту) и обратно.
12	Добавить функцию преобразования величин Площадь (Гектары в Квадратные метры) и обратно.
13	Добавить функцию преобразования величин Скорость (Узлы в Километр час) и обратно.
14	Добавить функцию преобразования величин Температура (Градусы Цельсия в Градусы Фаренгейта) и обратно.
15	Добавить функцию преобразования величин Угол (Градусы в Радианы) и обратно.
16	Добавить функцию преобразования величин Энергия (Калория в Джоуль) и обратно.
17	Добавить функционал расчета ипотеки.
18	Написать логическое выражение для расчета суммы выплаты по депозиту на вклад: до 5000 грн. начисляется 20% годовых, от 5000 грн. до 10000 грн. - 22% годовых. Все данные должны вводится с элементов форм.
19	Добавить в калькулятор расчет уравнения $ax^2+bx+c=0$.
20	Добавить в калькулятор три основных тригонометрических функций на выбор.