№ 6 Создание приложений на основе Windows Forms

Задание

1) Изучите методические указания, создайте приложение по варианту. Используйте ЭУ – кнопки, тестовые поля, метки, флаги и т.п. Начните с разработки класса Калькулятор. Используйте делегаты и подписки на события. Не забывайте проверять тип введенных значений на корректность (механизм исключений)

Вариант	Задания
1, 6, 11	Приложение Калькулятор для целых Сложение, вычитание, деление, умножение, двух целых чисел, возведение в степень, хранение значения в памяти.
2, 7, 12	Приложение Калькулятор для вещественных Сложение, вычитание, деление, умножение двух вещественных чисел, извлечение корня, sin, cos, хранение значения в памяти.
3, 8, 13	Приложение Калькулятор для текста. Объединение, замена подстроки на подстроку, удаление подстрок, получение символа по индексу, длина строки, удвоение каждого гласного.
4, 9, 14	Приложение Бинарный калькулятор. И, ИЛИ, Исключающее или, НЕ для двух целых, представление результатов в разных системах счисления
5, 10, 15	Приложение Сравнивающий калькулятор для строк >, <, ==, != и операции сдвига <<, >>.

* ЭУ – элементы управления

2) Создать Windows приложение на основе лабораторной №4. Форма должна содержать кнопку генерации коллекции объектов, заданного размера, окно для вывода коллекции, две кнопки для сортировки (по убыванию и возрастанию) , кнопки для выполнения запросов и окна вывода результатов.

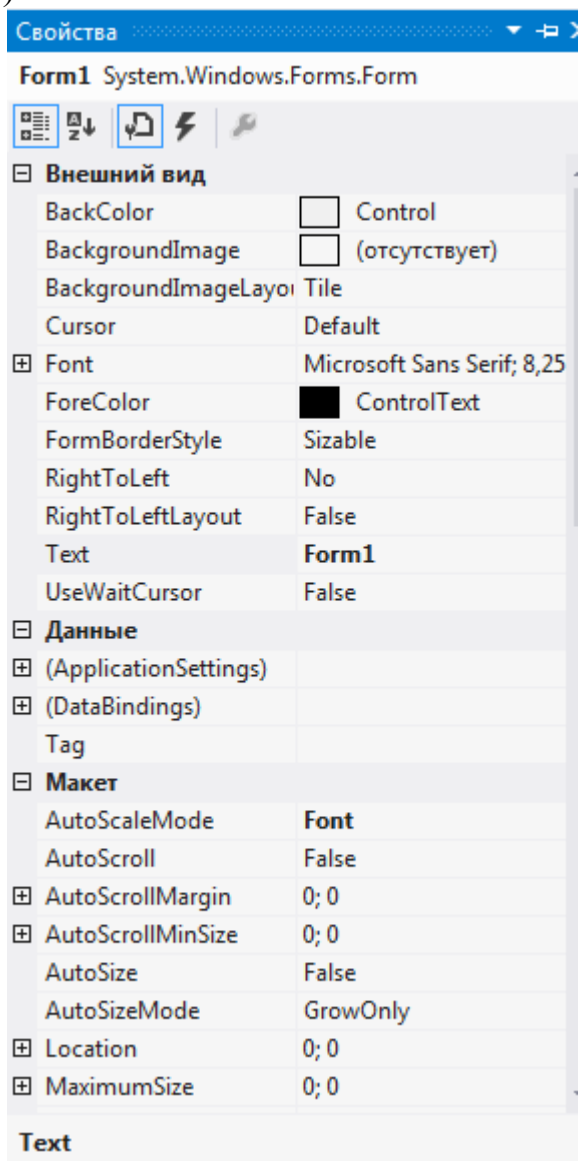
Для сортировки должен быть один метод и делегат Comparator, который определяет порядок сортировки.

Методические указания

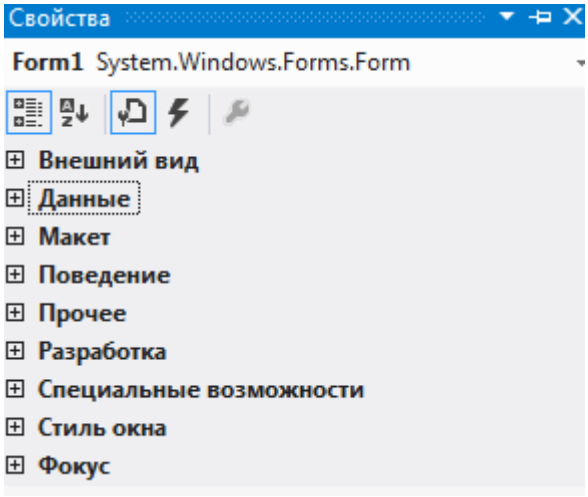
Любое окно приложения для Windows , представляет собой форму, порожденную от класса System.Windows.Forms.Form.

Откроем новый проект – проект Windows Application. Проект создается по аналогии с Console Appliation. Перед нами появится пустая поверхность окна приложения – форма. Размер, местоположение, цвет фона, имя

заголовка и другие свойства формы можно изменять с помощью окна свойств – Properties(отображение свойств выбранного элемента и событий, связанных с ним).



Панель Properties содержит следующие разделы свойств:



Accessibility – достижимость
Appearance – вид
Behavior – поведение
Configurations – конфигурации
Data – данные
Design – проект
Focus – центр
Layout – размещение
Window style – стиль окна

Используя раздел свойств Accessibility можно изменять:

- цвет поверхности – BackColor (по умолчанию такого же цвета будут все элементы управления);
- фоновую картинку − BackgroundImage,
- вид курсора − Cursor,
- цвет шрифта − ForeColor,
- стиль окна − FormBorderStyle,
- текст заголовка формы − Text ,
- местоположение текста заголовка формы − RightToLeft

Используя раздел свойств Layout можно изменять:

- размер окна − Size
- параметр, определяющий начальную позицию − StartPosition,
- координаты левого верхнего угла формы − Location (актуален, если StartPosition = Manual),
- вид начального отображения формы (минимизированное, нормальное или максимизированное) − WindowState,
- размер формы в максимизированном состоянии − MaxsimumSize,
- размер формы в минимизированном состоянии − MinimumSize,

Используя раздел свойств Window style можно изменять:

- вид иконки − Icon
- доступность кнопки максимизации −MaximizeBox
- доступность кнопок минимизации − MinimizeBox

и т.д.

Легко заметить, что мы можем изменять размеры формы не только с помощью панели свойства (Properties), но и с помощью мыши, при этом размеры (Size) будут меняться автоматически.

Поэкспериментируйте с этими и другими свойствами, посмотрите изменения на форме, связанные с изменением свойств.

Так же можно увидеть панели:

- Server Explorer (подключение к проекту БД),
- Solution Explorer (отображение подключенных проектов, файлов, пространств имен и т.п.),
- Dynamic Help (отображение справочной информации),
- Toolbox (на поверхность формы можно добавлять различные элементы управления)

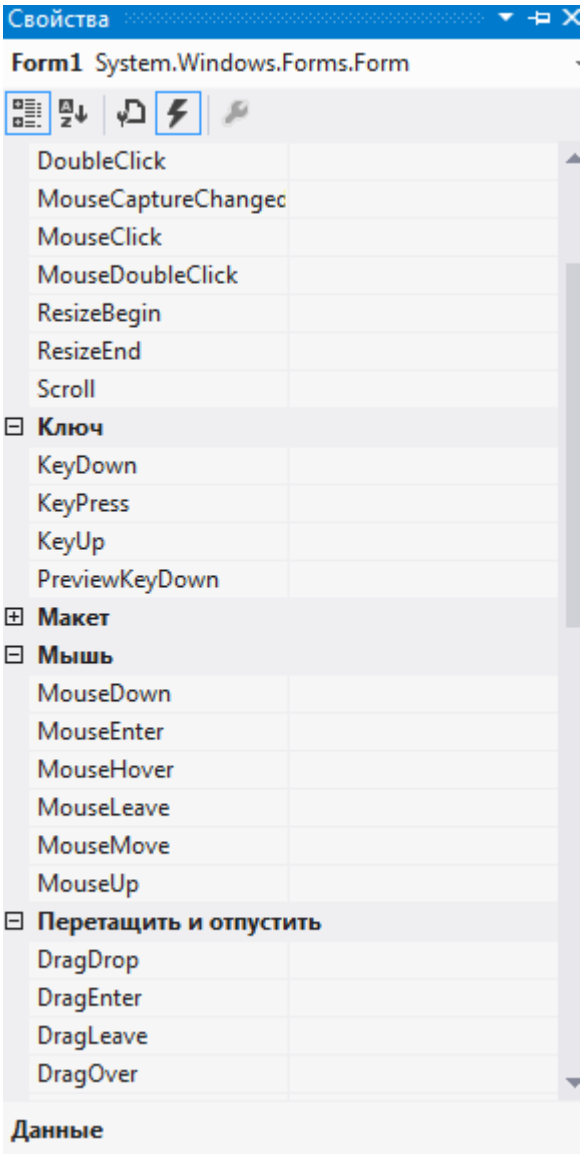
и другие.

Добавить дополнительные панели можно через пункт меню «View».

Теперь можно рассмотреть события. На той же панели свойств есть кнопка с пиктограммой «молния». Щелкнем на ней. При этом отобразятся различные события, связанные:

- с мышью (движение, движение в определенных направлениях, вход или выход из определенной области, нажатиями клавиши мыши),
- с нажатием клавиши клавиатуры,
- с рисованием,
- с изменением стиля,
- с размером окна,

и многие другие (подробнее с теми или иными событиями можно ознакомиться самостоятельно, используя источники с более глубокой детализацией материала).

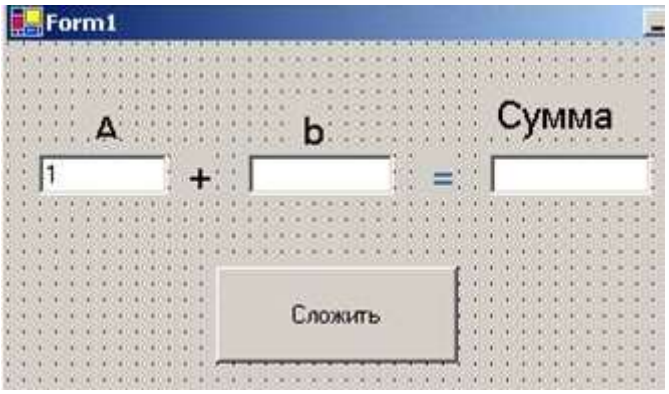


Чтобы задействовать то или иное событие достаточно лишь дважды щелкнуть на его названии и компилятор автоматически создаст имя и тело метода, который будет обрабатывать выбранной событие. Так же можно

ввести самостоятельно имя метода и сметить фокус от выбранного события – тело метода создастся автоматически с введенным именем! Существует еще и третий способ: сначала создается метод обрабатывающий событие, а затем выбирается в выпадающем списке! В тело метода необходимо поместить все, что необходимо для обработки события.

Создадим простенькое приложение − калькулятор для сложения двух чисел.

Для этого, во-первых, удаляем все «набросанные» на поверхность формы элементы управления, поверхность − должна быть пустой.



Во-вторых, «выкладываем» (перетаскиваем) из Toolbox на форму два Textbox, пять Label и одну Button. Изменив имена Label на – «A», «B», «+»,«=» и «Сумма» добиваемся, чтобы это выглядело примерно как на рисунке.

Затем, для удобства изменим имена (не текст!) Textbox и Button. Первому Textbox присваиваем имя «A», второму – «B»,третьему – «C», а Button – «Calculate». Все это делается в панели свойств (не забывайте при этом выделить нужный элемент!). Теперь нам нужно добавить обработку лишь одного события – нажатие кнопки. Это можно сделать всеми перечисленными выше способами или просто двойным щелчком по кнопке. И сразу же автоматически появляется код приложения с телом метода, обрабатывающего событие нажатия кнопки.

Пояснение. При двойном щелчке по любому элементу управления автоматически создается метод по обработке того или иного события, но(!!!) создается всегда, то событие обработка, которого более характерна для данного элемента управления. Для кнопки – нажатие на кнопку, для формы – загрузка и т.п.

В теле метода нужно дописать код, который будет выглядеть следующим образом:

```
     private void Calculate_Click(object sender, System.EventArgs e)
     {
          double a, b;
          try/*Обработка исключений, здесь выделяется блок кода, в
котором могут возникнуть исключения (ошибки)*/
          {
               a=Convert.ToDouble(A.Text);/*Считываем текст (A.Text),
                    затем конвертируем его в формат double*/
               b=Convert.ToDouble(B.Text);/*Считываем текст (B.Text),
                    затем конвертируем его в формат double*/
               C.Text=Convert.ToString(a+b);/*Присваиваем
           тексту    третьего редактора (C.Text) конвертированную в
```

```
            строковый формат сумму чисел     (a+b)полученных из двух
            редакторов*/
           }
           catch//Обработка исключений – произошла ошибка
           {
           MessageBox.Show("Проверьте правильность ввода чисел!");
           }
      }
```

Делегаты

Делегаты – это новый тип данных в C#. Они предназначены для передачи в качестве параметров одних методов другим методам. Другими словами, если требуется передать метод в качестве параметра, сведения об методе необходимо поместить (обернуть) в особый тип объекта – делегат.

По своей структуре делегат - это объект, который ссылается на метод, то есть делегат указывает на адрес области памяти, являющейся точкой входа в метод. С помощью делегата можно вызвать метод, на который он указывает.

В процессе выполнения программы делегату можно присвоить ссылку на другой метод. Это дает возможность определять во время выполнения программы, какой из методов должен быть вызван. Существенным моментом, является то, что в отличие от указателей, делегаты безопасны по типу.

Делегаты реализуются как экземпляры классов, производных от библиотечного класса System.Delegate. Для создания делегата необходимо выполнит два шага.

На первом шаге необходимо объявить делегат. При этом сигнатура делегата должна полностью соответствовать сигнатуре метода, который он представляет.

Например, делегат должен ссылаться на метод класса CA:

static **int min(int x,int y)**,

тогда объявление делегата может выглядеть, следующим образом:

delegate int LpFunc(int a,int b);

На втором шаге мы должны создать экземпляр делегата для хранения сведения о представляемом им методе:

LpFunc pfnk = new LpFunk(CA.min);

Экземпляр делегата может ссылаться на любой статический метод или метод объекта любого класса, при условии, что сигнатура метода полностью соответствует сигнатуре делегата.

Использование делегата демонстрируется в коде примера.

Пример:

```
using System;

namespace ConsoleApplication14
{
      /// <summary>
      /// Summary description for Class1.
      /// </summary>
      ///
```

```
class MathOprt
{
      public static double Mul2(double val)
      {
            return val*2;
      }
      public static double Sqr(double val)
      {
            return val*val;
      }
}
delegate double DblOp(double x);//объявление делегата

class Class1
{
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]

       static void Main(string[] args)
      {
            //
            // TODO: Add code to start application here
            DblOp [] operation = // создание экземпляров делегата
            {
                  new DblOp(MathOprt.Mul2),
                  new DblOp(MathOprt.Sqr)
            };

            for(int j=0;j<operation.Length;j++)
            {
                  Console.WriteLine("Резльтаты операции[{0}]:",j);
                  Prc(operation[j], 4.0);
                  Prc(operation[j], 9.94);
                  Prc(operation[j], 3.143);
            }


            //
      }

      static void Prc(DblOp act, double val)
      {
      double rslt = act(val);
      Console.WriteLine("Исходное значение {0}, результат  {1}",
                  val,rslt);
      }
}
}
```

Делегаты могут хранить несколько адресов областей памяти. То есть делегат может указывать на несколько различных методов. Это позволяет, последовательно инициализируя адреса вызывать метод за методом. Эта способность делегатов называется *многоадресностью делегатов*.

Для создания цепочки вызовов методов необходимо сначала создать экземпляр делегата для одного метода, а затем с помощью операции “ + = ” добавить остальные методы. В процессе выполнения кода можно не только добавлять новые методы, но и удалять не нужные с помощью операции ” - = ”.

Многоадресные делегаты должны иметь тип возвращаемого значения **void**. Это означает, что и методы, представляемые многоадресными делегатами также должны возвращать значение **void**.

Перепишем код из примера с применением многоадресного делегата.

Пример:

```
using System;

namespace ConsoleApplication14
{
      /// <summary>
      /// Summary description for Class1.
      /// </summary>
      ///
      class MathOprt
      {
            public static void Mul2(double val)
            {
                  double rslt= val*2;
            Console.WriteLine("Mul2
bсходное значение {0},результат  {1}",
                        val,rslt);
            }
            public static void Sqr(double val)
            {
                  double rslt = val*val;
            Console.WriteLine("Sqr исходное значение {0}, результат  {1}
",
                        val,rslt);

            }
      }
      delegate void DblOp(double x);//объявление делегата

      class Class1
      {
            /// <summary>
            /// The main entry point for the application.
            /// </summary>
            [STAThread]
            static void Main(string[] args)
            {
                  //
                  // TODO: Add code to start application here
                  DblOp  operations = new DblOp(MathOprt.Mul2);
                        operations += new DblOp(MathOprt.Sqr);

                        Prc(operations, 4.0);
                        Prc(operations, 9.94);
                        Prc(operations, 3.143);
```

```
            }


                  //

            static void Prc(DblOp act, double val)
            {

                  Console.WriteLine("\n*********\n");
                  act(val);
            }
      }
}
```

События

Работа с событиями осуществляется в C# согласно модели «издатель-подписчик». Класс, ответственный за инициализацию (выработку) событий публикует событие, и любые классы могут подписаться на это событие. При возникновении события исполняющая среда уведомляет всех подписчиков о произошедшем событии, при этом вызываются соответствующие методы-обработчики событий подписчиков. Какой обработчик события будет вызван – определяется делегатом.

Платформа .NET требует для всех обработчиков событий следующей сигнатуры кода:

```
void OnRecChange(object source, ChangeEventArgs e)
{
      // Код для обработки события
}
```

Обработчики событий обязательно имеют тип возвращаемого значения void. Обработчики событий принимают два параметра. Первый параметр _ это ссылка на объект, сгенерировавший событие. Эта ссылка передается обработчику самим генератором событий. Второй параметр – это ссылка на объект класса EventArgsили класса производного от него. В производном классе может содержаться дополнительная информация о событии.

Обработчик события определяется делегатом. Согласно сигнатуре обработчика события делегат должен принимать два параметра и выглядеть следующим образом:

```
public delegate void ChangeEventHandle(object source,ChangeEventArgs
e);
```

Для того, чтобы иметь возможность подписаться на событие класс генератора событий должен содержать член типа указанного делегата с ключевым словом event и метод, который будет вызываться при возникновении события, например:

```
public event ChangeEventHandler OnChangeHandler;
```

Этот член является специализированной формой многообъектного делегата. Использую операцию “+=”, клиенты могут подписаться на это сообщение:

```
gnEvent.OnChangeHandler +=
          new GenEvent.ChangeEventHandler (OnRecChange);
```

где `gnEvent` имя класса генератора событий.

Нижеследующий пример демонстрирует работу с событиями.

Пример:

```
using System;
namespace sobit
{
      /// <summary>
      /// Summary description for Class1.
      /// </summary>

      class ChangeEventArgs : EventArgs
      {
            string str;
            public string Str
            {
                  get
                  {
                        return str;
                  }
            }
            int change;
            public int Change
            {
                  get
                  {
                        return change;
                  }
            }
            public ChangeEventArgs(string str,int change)
            {
                  this.str = str;
                  this.change = change;
            }
      }
            class GenEvent  // Генератор событий - издатель
            {
                  public delegate void ChangeEventHandler
                        (object source,ChangeEventArgs e);

                  public event ChangeEventHandler OnChangeHandler;

                  public void UpdateEvent(string str,int change)
                  {
```

```
                    if(change==0)
                        return;
                    ChangeEventArgs e =
                    new ChangeEventArgs(str,change);

                    if (OnChangeHandler != null)
                        OnChangeHandler(this,e);
                }

            }

        //Подписчик
        class RecEvent
        {
        //Обработчик события
            void OnRecChange(object source,ChangeEventArgs e)
            {
                int change = e.Change;
                Console.WriteLine("Вес груза '{0}' был {1} на {2} тонн
ы",
                    e.Str,change > 0 ? "увеличен" : "уменьшен",
                    Math.Abs(e.Change));
            }

            GenEvent gnEvent;
        // в конструкторе класса осуществляется подписка
            public RecEvent(GenEvent gnEvent)
            {
            this.gnEvent = gnEvent;
            gnEvent.OnChangeHandler +=     //здесь осуществляется
подписка
                    new GenEvent.ChangeEventHandler(OnRecChange);
            }
        }

        class Class1
        {
            /// <summary>
            /// The main entry point for the application.
            /// </summary>
            [STAThread]
            static void Main(string[] args)
            {
                //
                // TODO: Add code to start application here

                GenEvent gnEvent = new GenEvent();
                RecEvent inventoryWatch =      new RecEvent(gnEvent);
                gnEvent.UpdateEvent("грузовика", -2);
                gnEvent.UpdateEvent("автопоезда", 4);


                //
            }
        }
}
```