**Лабораторная работа № 13**

**Разработка программы, управляемой событиями**

**Цель.** Получить практические навыки разработки программы, управляемой событиями, использования делегатов и событий..

**Теоретические сведения.**

Механизм работы с событиями предусматривает несколько этапов:

1. Объявление делегата-типа, задающего описание методов (еще неизвестных на данном этапе), которые будут вызываться при обработке события.
2. Определение переменной события, имеющей тип делегата-события.
3. Определение генератора события (посылки сообщения) с указанием аргументов, информирующих получателей о состоянии объекта, пославшего сообщение.
4. Определение методов обработки события. Описание каждого метода должно соответствовать типу делегата события.
5. Создание экземпляра делегата, на который настроено событие. Аргумент конструктора – имя метода обработки.
6. Подключение экземпляра делегата к переменной события.

Эти этапы, как правило, относятся к разным классам программ.

Класс\_1 (sender) – генерирует события, реализует этапы 1,2,3.

Класс\_2 (receiver) – обрабатывает события, он должен иметь методы обработки или средства для доступа к методам обработки. Реализует этапы 4,5,6.

Иногда в программе может присутствовать еще и третий класс (супервизор), - он управляет процессом на более высоком уровне. Супервизор выполняет следующие действия:

1. Создает объект для генерации событий (клиента).
2. Создает объект для класса, обрабатывающего события (сервер). Если метод обработки статический, то объект не создается.
3. Подключить объект-делегат к переменной события из объекта-клиента.
4. Передать управление клиенту (какому-либо методу, генерирующему событие).

**Класс sender.**

В C# каждое событие определяется делегатом, описывающим сигнатуру сообщения. Объявление события - это двухэтапный процесс:

* Объявляется делегат - функциональный класс, задающий сигнатуру. Объявление делегата может быть помещено в класс Sender. Но, чаще всего, это объявление находится вне класса в пространстве имен. Поскольку одна и та же сигнатура может быть у разных событий, то для них достаточно иметь одного делегата. Для некоторых событий можно использовать стандартные делегаты, встроенные в каркас. Тогда достаточно знать только их имена.
* Если делегат определен, то в классе Sender, создающем события, достаточно объявить событие как экземпляр соответствующего делегата. Формальный синтаксис объявления таков:

[атрибуты] [модификаторы]event [тип, заданный делегатом] [имя события]

Пример объявления делегата и события, представляющего экземпляр этого делегата:

namespace Events

{

public delegate void EventHandler(object Sender, int a1, int a2);

public class Sender  
{

public event EventHandler evnt;

...  
}//class Sender  
...

}//namespace Events

Здесь делегат EventHandler описывает класс событий, сигнатура которых содержит три аргумента. Событие evnt в классе Sender является экземпляром класса, заданного делегатом.

Поскольку действия по включению могут повторяться, полезно в состав методов класса добавить защищенную процедуру, включающую событие. Даже если событие генерируется только в одной точке, написание такой процедуры считается признаком хорошего стиля. Этой процедуре обычно дается имя, начинающееся со слова On, после которого следует имя события. Будем называть такую процедуру On-процедурой. Она проста и состоит из вызова объявленного события, включенного в тест, который проверяет перед вызовом, а есть ли хоть один обработчик события, способный принять соответствующее сообщение.

protected virtual void OnAction(int a1, int a2)  
{  
 if (evnt!=null)  
 OnAction(this,a1, a2);  
}

Tе, кто принимает сообщение о событии, должны заранее присоединить обработчики событий к объекту EventHandler evnt, задающему событие.

Последний шаг, который необходимо выполнить в классе Sender - это в нужных методах класса вызвать процедуру On. Естественно, что перед вызовом нужно определить значения входных аргументов события. После вызова может быть выполнен анализ выходных аргументов, определенных обработчиками события.

Объекты класса Sender создают события и уведомляют о них объекты, возможно, разных классов, названных нами классами Receiver. Класс Receiver должен:

* иметь обработчик события - процедуру, согласованную по сигнатуре с функциональным типом делегата, который задает событие;
* иметь ссылку на объект, создающий событие, чтобы получить доступ к этому событию event-объекту;
* уметь присоединить обработчик события к event-объекту. Это можно реализовать по-разному, но технологично это делать непосредственно в конструкторе класса, так что когда создается объект, получающий сообщение, он изначально готов принимать и обрабатывать сообщения о событиях.

public class Receiver  
{

// ссылка на объект, генерирующий событие  
private Sender MySender;

public Receiver (Sender S) // Присоединяем //обработчик события к event-объекту   
{  
 this.MySender=s;

MySender.evnt += new EventHandler (MyHandler);  
}

private void MyHandler(object Sender, int a1, int a2)  
{  
 Console.WriteLine(“a1={0}, a2={1} “, a1, a2);  
}  
public void GoOut()  
{  
 MySender.evnt - = new EventHandler(MyHandler);  
}  
}//Receiver

В классе Receiver есть ссылка на объект класса Sender, создающий события. Сам объект передается в конструкторе класса. Здесь же происходит присоединение обработчика события к event-объекту. Обработчик события MyEventHandler выводит сообщение на консоль.

public class MyEventArgs : EventArgs

{

private object item;//задает входной аргумент события, передаваемый обработчику события

private bool permit;//задает выходной аргумент события, получающий в обработчике значение True, если обработчик события разрешает изменение элемента.

public object Item

{

get { return (item); }

set { item = value; }

}

public bool Permit

{

get { return (permit); }

set { permit = value; }

}

}//class MyEventArgs

// Объявление делегата

public delegate void MyEventHandler(object sender, MyEventArgs args);

// Класс, создающий событие. Потомок класса ArrayList.

public class MyList : ArrayList

{

//Свойства класса: событие и его аргументы

public event MyEventHandler Changed; //Событие, включается при всех изменениях элементов списка.

//Аргументы события

private MyEventArgs evargs = new MyEventArgs();// аргументы события, передаваемые обработчикам.

//Методы класса: процедура On и переопределяемые методы.

//Процедура On, включающая событие

protected virtual void OnChanged(MyEventArgs args)

{

if (Changed != null)

Changed(this, args);

}

//Переопределяемые методы, вызывающие событие Changed

public override int Add(object value) //Добавление нового элемента при получении разрешения у обработчиков события

{

int i = 0;

evargs.Item = value;

OnChanged(evargs);

if (evargs.Permit)

i = base.Add(value);

else

Console.WriteLine("Добавление элемента запрещено. Значение = {0}", value);

return i;

}

public override void Remove(object value) //Удаление элемента при получении разрешения у обработчиков события

{

evargs.Item = value;

OnChanged(evargs);

if (evargs.Permit)

base.Remove(value);

else

Console.WriteLine("Удаление элемента запрещено. Значение = {0}", value);

}

public override void Clear()

{

evargs.Item = 0;

OnChanged(evargs);

base.Clear();

}

public override object this[int index]//индексатор

{

set

{

evargs.Item = value;

OnChanged(evargs);

if (evargs.Permit)

base[index] = value;

else

Console.WriteLine("Замена элемента запрещена. Значение = {0}", value);

}

get { return (base[index]); }

}

}

class Receiver1//получает и обрабатывает событие Changed

{

private MyList List;

public Receiver1(MyList list)

{

List = list;

// Присоединяет обработчик к событию.

OnConnect();

}

//Обработчик события - выдает сообщение.

//Разрешает добавление элементов, меньших 10.

private void ListChanged(object sender,MyEventArgs args)

{

Console.WriteLine("Receiver1: Сообщаю об изменениях: Item ={0}", args.Item);

args.Permit = ((int)args.Item < 10);

}

//Разрешает удаление элементов, меньших 5.

private void ListRemoved(object sender, MyEventArgs args)

{

Console.WriteLine("Receiver1: Сообщаю об изменениях: Item ={0}", args.Item);

args.Permit = ((int)args.Item < 5);

}

public void OnConnect()

{

//Присоединяет обработчик к событию

List.Changed += new MyEventHandler(ListChanged);

List.Changed += new MyEventHandler(ListRemoved);

}

public void OffConnect()

{

//Отсоединяет обработчик от события и удаляет список

List.Changed -= new MyEventHandler(ListChanged);

List.Changed -= new MyEventHandler(ListRemoved);

List = null;

}

}//class Receiver1

class Receiver2

{

private MyList List;

public Receiver2(MyList list)

{

List = list;

// Присоединяет обработчик к событию.

OnConnect();

}

// Обработчик события - выдает сообщение.

//Разрешает добавление элементов, меньших 20.

private void ListChanged(object sender,MyEventArgs args)

{

Console.WriteLine("Receiver2: Сообщаю об изменениях: " + "Item ={0}", args.Item);

args.Permit = ((int)args.Item < 20);

}

public void OnConnect()

{

//Присоединяет обработчик к событию

List.Changed += new MyEventHandler(ListChanged);

//Заметьте, допустимо только присоединение (+=),

//но не замена (=)

//List.Changed = new ChangedEventHandler(ListChanged);

}

public void OffConnect()

{

//Отсоединяет обработчик от события и удаляет список

List.Changed -= new MyEventHandler(ListChanged);

List = null;

}

}//class Receiver2

class Program

{

static void Main(string[] args)

{

//Создаются два объекта, вырабатывающие события

MyList list = new MyList();

MyList list1 = new MyList();

//Создаются три объекта двух классов EventReceiver1 и Receiver2, способные обрабатывать события класса ListWithChangedEvent

Receiver1 Receiver1 = new Receiver1(list);

Receiver2 Receiver21 = new Receiver2(list);

Receiver2 Receiver22 = new Receiver2(list1);

Random rnd = new Random();

//Работа с объектами, приводящая к появлению событий

list.Add(rnd.Next(20)); list.Add(rnd.Next(20));

list[1] = 17;

int val = (int)list[0] + (int)list[1]; list.Add(val);

list.Clear();

list1.Add(10); list1[0] = 25; list1.Clear();

list.Add(21); list.Add(3);

list.Remove(3); list.Remove(21);

//Отсоединение обработчика событий

Receiver1.OffConnect();

list.Add(21);

list.Clear();

}

}

}

**Постановка задачи**

**Часть 1.**

1. Создать иерархию классов (см. лаб. 9). Для каждого класса реализовать конструктор без параметров, с параметрами, свойства для доступа к полям объектов, метод для автоматического формирования объектов. Перегрузить метод ToString() для формирования строки со значениями всех полей класса.
2. Создать класс MyCollection как производный класс от класса Сollection<MyClass>. Реализовать в классе методы для заполнения коллекции (элементы коллекции формируются автоматически), добавления элементов коллекции, удаления элементов коллекции, сортировки элементов коллекции по заданному полю, очистки коллекции. Реализовать итератор для доступа к элементам коллекции. Реализовать свойство Length (только для чтения), содержащее текущее количество элементов коллекции.
3. Написать демонстрационную программу иллюстрирующую работу всех методов класса MyCollection. Использовать исключительные ситуации для обработки ошибочных ситуаций.

**Часть 2.**

1. Определить класс MyNewCollection производный от класса MyCollection, который с помощью событий извещает об изменениях в коллекции.

* Коллекция состоит из объектов ссылочных типов. Коллекция изменяется при удалении/добавлении элементов или при изменении одной из входящих в коллекцию ссылок, например, когда одной из ссылок присваивается новое значение. В этом случае в соответствующих методах или свойствах класса бросаются события.
* При изменении данных объектов, ссылки на которые входят в коллекцию, значения самих ссылок не изменяются. Этот тип изменений не порождает событий.
* Для событий, извещающих об изменениях в коллекции, определяется свой делегат. События регистрируются в специальных классах-слушателях.

1. Для событий определить делегат CollectionHandler с сигнатурой:

void CollectionHandler (object source, CollectionHandlerEventArgs args);

1. Определить класс CollectionHandlerEventArgs, производный от класса System.EventArgs, который содержит

* открытое автореализуемое свойство типа string с названием коллекции, в которой произошло событие;
* открытое автореализуемое свойство типа string с информацией о типе изменений в коллекции;
* открытое автореализуемое свойство для ссылки на объект, с которым связаны изменения;
* конструкторы для инициализации класса;
* перегруженную версию метода string ToString() для формирования строки с информацией обо всех полях класса.

1. В новую версию класса MyNewCollection добавить

* открытое автореализуемое свойство типа string с названием коллекции;
* метод bool Remove (int j) для удаления элемента с номером j ; если в списке нет элемента с номером j, метод возвращает значение false;
* индексатор (с методами get и set) с целочисленным индексом для доступа к элементу с заданным номером.

1. В класс MyNewCollection добавить два события типа CollectionHandler.

* CollectionCountChanged, которое происходит при добавлении нового элемента в коллекцию или при удалении элемента из коллекции; через объект CollectionHandlerEventArgs cобытие передает имя коллекции, строку с информацией о том, что в коллекцию был добавлен новый элемент или из нее был удален элемент, ссылку на добавленный или удаленный элемент;
* CollectionReferenceChanged, которое происходит, когда одной из ссылок, входящих в коллекцию, присваивается новое значение; через объект CollectionHandlerEventArgs событие передает имя коллекции, строку с информацией о том, что был заменен элемент в коллекции, и ссылку на новый элемент.

1. Событие CollectionCountChanged бросают следующие методы класса MyNewCollection

* AddDefaults();
* Add (object[] ) ;
* Remove (int index).

1. Событие CollectionReferenceChanged бросает метод set индексатора, определенного в классе MyNewCollection.
2. Определить класс Journal, который можно использовать для накопления информации об изменениях в коллекциях типа MyNewCollection. Класс Journal хранит информацию в списке объектов типа JournalEntry. Каждый элемент списка содержит информацию об отдельном изменении, которое произошло в коллекции. Класс JournalEntry содержит

* открытое автореализуемое свойство типа string с названием коллекции, в которой произошло событие;
* открытое автореализуемое свойство типа string с информацией о типе изменений в коллекции;
* открытое автореализуемое свойство типа string c данными объекта, с которым связаны изменения в коллекции;
* конструктор для инициализации полей класса;
* перегруженную версию метода string ToString().

1. Класс Journal содержит

* коллекцию элементов типа JournalEntry (закрытое поле);
* обработчики событий CollectionCountChanged и CollectionReferenceChanged, которые добавляют новый элемент JournalEntry в коллекцию; для инициализации JournalEntry используется информация из объекта CollectionHandlerEventArgs, который передается вместе с событием;
* перегруженную версию метода string ToString() для формирования строки с информацией обо всех элементах массива.

1. Написать демонстрационную программу, в которой:

* создать две коллекции MyNewCollection.
* Создать два объекта типа Journal, один объект Journal подписать на события CollectionCountChanged и CollectionReferenceChanged из первой коллекции, другой объект Journal подписать на события CollectionReferenceChanged из обеих коллекций.

1. Внести изменения в коллекции MyNewCollection

* добавить элементы в коллекции;
* удалить некоторые элементы из коллекций;
* присвоить некоторым элементам коллекций новые значения.

1. Вывести данные обоих объектов Journal.

**При работе в среде VisualStudio необходимо уметь**

* средствами Solution Explorer добавить в проект новый класс;
* с помощью диаграммы классов(Class Diagram) добавить в класс методы, поля и свойства;
* с помощью диаграммы классов добавить в класс перегруженную (override) версию виртуального метода.