Оглавление

Лабораторная работа 8

8.1 Общие сведения

В контексте C# событие - это способ, с помощью которого один класс оповещает другой (другие) класс о чем-то произошедшем. Можно сказать, что механизм событий использует идеологию публикация/подписка". Какой-то класс публикует свои события, а другие классы подписываются на те события, которые им интересны.

Эта модель используется, например, при использовании Windows Forms, WPF. Для работы с событием нужно выполнить следующие действия:

- Создать делегат, который будет использоваться для вызова нужного метода при срабатывании события;
- Определить само событие при помощи ключевого слова event.

События являются членами класса и объявляются с помощью ключевого слова **event**. Чаще всего для этой цели используется следующая форма:

```
event делегатСобытия имяСобытия;
```

В качестве примера рассмотрим пример интегрирования из лабораторной работы № 6.

Допустим, что вычисление интегрируемой функции очень ресурсоемко, повторный расчет нежелателен, поэтому требуется: – Сделать пошаговую запись работы интегратора (возможные направления записи: в файл, на график, в базу данных, консоль, в консоль отладки. В том числе возможны комбинации, например файл + график, БД + консоль и т.п.). Кроме того, учет хода вычислений позволит настроить интегратор, например уменьшить или увеличить шаг, т.е. появляется обратная связь; – Уведомить пользователя об окончании расчетов (в случае, если нужно выполнить какие-то дополнительные действия по окончании);

За основу возьмем немного модифицированный код из лабораторной работы № 6:

```
public class Integrator
{
    private readonly Equation equation;
    public Integrator( Equation equation)
    {
        this.equation = equation;
    }
    public double Integrate(double x1,double x2)
    {
        int N = 100;
        double h = (x2 - x1) / N;
        double sum = 0;
        for (int i = 0; i < N; i++) {
            sum = sum + equation.Value(x1 + i * h) * h;
```

```
        }
        return sum;
    }
}
public class Equation
{
    private readonly double a;
    private readonly double b;
    private readonly double c;

    public QuadEquation(double a, double b, double c)
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }
    public double Value(double x)
    {
        return a * x * x + b * x + c;
    }
}

static void Main( string [] args )
{
    Equation e = new QuadEquation(0, 20, 0);
    Integrator i1 = new Integrator(e);
    double integrValue = i1.Integrate(10,30);
}
```

Опишем делегат для события, возникающего на каждом шаге интегрирования. Т.к. мы планируем использовать это событие для пошаговой записи работы интегратора, нам потребуется знать текущую точку X, значение функции F в этой точке и текущее значение интеграла (накопленная на данный момент сумма). Таким образом делегат должен ссылаться на функцию с тремя вещественными параметрами:

```
public delegate void IntStepDelegate  ( double x, double f, double integr );
```

В случае завершения расчетов нам скорее важен сам факт окончания расчетов, чтобы произвести определенные действия, поэтому, определим там только один параметр для передачи значения интеграла по итогам расчетов:

```
public delegate void IntFinishDelegate( double integr );
```

Добавим события в класс интегратора, для этого объявим делегаты данных типов с ключевым словом **event**:

```
class Integrator
{
    //событие, возникающее на каждом шаге интегрирования:
    public event IntStepDelegate    OnStep;
    //событие, возникающее при окончании расчетов:
    public event IntFinishDelegate  OnFinish;
    //остальной код...
}
```

Теперь, чтобы возбудить событие, достаточно написать вызов OnStep и OnFinish с передачей требуемых параметров, например:

```
OnFinish( sum );
```

C# трансформирует приведенный выше код в вызов всех подписанных на событие методов. Однако, следует учитывать, что OnStep и OnFinish - не самостоятельные методы, а лишь ссылки на них, поэтому могут принимать значения null. Это значение они будут иметь, если на данные события не окажется подписчиков, соответственно инициация такого события закончится исключением NullReferenceException. Чтобы избежать этого, необходимо немного добавить проверку на null:

```
if(OnFinish != null )
{
    OnFinish( sum );
}
```

В новых версиях C# этот код может быть сокращен:

```
OnFinish?Invoke( sum );
```

В случае, если событие может возбуждаться в разных местах, его необходимо поместить внутрь метода:

```
void RaiseFinishEvent( double sum )
{
    if(OnFinish != null )
    {
        OnFinish( sum );
    }
}
```

С учетом вышесказанного, класс "интегратор"будет выглядеть следующим образом:

```
public class Integrator
{
    //события:
    public event IntStepDelegate    OnStep;
    public event IntFinishDelegate  OnFinish;

    private readonly Equation equation;
    public Integrator( Equation equation)
    {
        this.equation = equation;
    }
    public double Integrate(double x1,double x2)
    {
        int N = 100;
        double h = (x2 - x1) / N;
        double sum = 0;
        for (int i = 0; i < N; i++) {
            sum = sum + equation.Value(x1 + i * h) * h;
            //генерируем событие о завершенном шаге интегрирования:
            RaiseStepEvent(  x, f,  sum );
```

```
        }
        //генерируем событие о завершении расчетов:
        RaiseFinishEvent( sum );
        return sum;
    }
    void RaiseStepEvent(  double x, double f, double sum )
    {
        if( OnStep != null )
        {
            OnStep( x, f, sum );
        }
    }
    void RaiseFinishEvent( double sum )
    {
        if(OnFinish != null )
        {
            OnFinish( sum );
        }
    }
}
```

Подписка на события будет выглядеть следующим образом:

```
//функция, запускающая расчеты:
void BeginCalculation()
{
    //создаем объект "уравнение"
    Equation e = new QuadEquation(0, 20, 0);
    //создаем интегратор:
    Integrator integr = new Integrator(e);
    //подписываемся на события
    //с явным созданием объекта делегата:
    integr.OnStep += new IntStepDelegate (WriteToFile);
    //с передачей только имени функции:
    integr.OnStep += WriteToDataBase;
    //с использованием лямбда-выражения:
    integr.OnStep += (x, f, integr) =>
    {
        chart1.Series[0].Add(x,f);
        chart2.Series[0].Add(x, integr);
    };
    //подписываемся на событие о завершении расчетов:
    integr.OnFinish += (sum) => {
        MessageBox.Show("Интеграл = {0}", sum );};

    double integrValue = i1.Integrate(10,30);
}

//функция записи в файл:
void WriteToFile( double x, double f, double s)
{
    //каким то образом, записываем в файл
}

static void WriteToDataBase( double x, double f, double s)
{
    //каким то образом, записываем в БД
}
```

8.2 Использование EventHandler'ов

В C# разрешается формировать какие угодно разновидности событий. Однако, для совместимости программных компонентов со средой .NET Framework следует придерживаться следующих требований: у обработчиков событий должны быть два параметра.

Первый из них — ссылка на объект, формирующий событие, второй — параметр типа EventArgs, содержащий любую дополнительную информацию о событии, которая требуется обработчику.

Таким образом, .NET-совместимые обработчики событий должны иметь следующий вид:

```
void обработчик   (object отправитель, EventArgs e)
{

}

//как пример, обработчик нажатия кнопок в windows forms:
void button1_Click ( object sender, EventArgs e )
{

}
```

В среде .NET Framework предоставляется встроенный делегат *EventHandler<TEventArgs>*, реализующий такую модель. Тип *TEventArgs* обозначает тип второго аргумента события, например:

```
class IntegratorEventArgs
{
    public double X {get;set;}
    public double F { get; set; }
    public double Integr{ get;set;}
}

class Integrator
{
    public EventHandler< IntegratorEventArgs > OnStep;
    public double Integrate(double x1,double x2)
    {
        int N = 100;
        double h = (x2 - x1) / N;
        double sum = 0;
        for (int i = 0; i < N; i++) {
            sum = sum + equation.Value(x1 + i * h) * h;
            RaiseStepEvent(  x, f,  sum );
        }
        RaiseFinishEvent( sum );
        return sum;
    }
    void RaiseStepEvent(  double x, double f, double sum )
    {
        if( OnStep != null )
        {
            //"упкаковываем" параметры в объект типа IntegratorEventArgs
            IntegratorEventArgs  args = new IntegratorEventArgs()
            {
                X = x,
                F = f,
                Integr = sum
```

```
            };
            //и генерируем событие:
            OnStep( this, args );
        }
    }
}
```

8.3 Задание

Для лабораторной работы № 6 добавьте поддержку событий. Так же, как и описано в данной работе, должно быть 2 типа событий: на каждый шаг интегрирования, на завершение расчетов, при необходимости можете добавить еще один, запускаемый при начале расчетов (в случае, если необходима какая то дополнительная инициализация перед приемом данных).

В качестве возможных обработчиков создайте по 2 из предложенных ниже. Вариант задания согласуется с преподавателем.

Протестируйте работу приложения с одним, двумя и совсем без обработчиков событий.

- Для каждого шага интегрирования:
    1. Запись данных в текстовых файл;
    2. Запись данных в бинарный файл;
    3. Вывод данных в ListBox;
    4. Вывод данных на график, должно быть построено две функции: F(x) и Integr(X) - зависимость интеграла от точки X;
    5. Запись данных в строку, с выводом содержимого этой строки по окончании расчетов.
- На событие окончания расчетов:
    1. Вывод текстового сообщения с указанием значения интеграла;
    2. Вывод сообщения в заголовок формы;
    3. Блокировка кнопки запуска при начале расчетов и ее разблокировка при окончании;
    4. Смена цвета формы;
    5. Запись данных в строку, с выводом содержимого этой строки по окончании расчетов.

8.4 Дополнительное задание

1. Добавьте обработчик, сообщающий пользователю общее время расчетов. Для моделирования сложных вычислений можете на каждой итерации добавить *Thread.Sleep(100);* для задержки потока выполнения на 100мс.
2. Решите задание с использованием EventHandler'ов.