

Лабораторная работа №6

Цель работы:

Познакомиться с механизмом событий(event) в с#.

Необходимые теоретические сведения**События**


События сигнализируют системе о том, что произошло определенное действие. И если нам надо отследить эти действия, то как раз мы можем применять события.

Например, возьмем следующий класс, который описывает банковский счет:

```
class Account
{
    public Account(int sum)
    {
        Sum = sum;
    }
    // сумма на счете
    public int Sum { get; private set; }
    // добавление средств на счет
    public void Put(int sum)
    {
        Sum += sum;
    }
    // списание средств со счета
    public void Take(int sum)
    {
        if (Sum >= sum)
        {
            Sum -= sum;
        }
    }
}
```

В конструкторе устанавливаем начальную сумму, которая хранится в свойстве Sum. С помощью метода Put мы можем добавить средства на счет, а с помощью метода Take, наоборот, снять деньги со счета. Попробуем использовать класс в программе - создать счет, положить и снять с него деньги:

```
static void Main(string[] args)
{
    Account acc = new Account(100);
    acc.Put(20);    // добавляем на счет 20
    Console.WriteLine($"Сумма на счете: {acc.Sum}");
    acc.Take(70);  // пытаемся снять со счета 70
    Console.WriteLine($"Сумма на счете: {acc.Sum}");
    acc.Take(180); // пытаемся снять со счета 180
    Console.WriteLine($"Сумма на счете: {acc.Sum}");
    Console.Read();
}
```

Консольный вывод:

```
Сумма на счете: 120
Сумма на счете: 50
Сумма на счете: 50
```

Все операции работают как и положено. Но что если мы хотим уведомлять пользователя о результатах его операций. Мы могли бы, например, для этого изменить метод Put следующим образом:

```
public void Put(int sum)
{
    Sum += sum;
    Console.WriteLine($"На счет поступило: {sum}");
}
```

Казалось, теперь мы будем извещены об операции, увидев соответствующее сообщение на консоли. Но тут есть ряд замечаний. На момент определения класса мы можем точно не знать, какое действие мы хотим произвести в методе Put в ответ на добавление денег. Это может вывод на консоль, а может быть мы захотим уведомить пользователя по email или sms. Более того мы можем создать отдельную библиотеку классов, которая будет содержать этот класс, и добавлять ее в другие проекты. И уже из этих проектов решать, какое действие должно выполняться. Возможно, мы захотим использовать класс Account в графическом приложении и выводить при добавлении на счет в графическом сообщении, а не консоль. Или нашу библиотеку классов будет использовать другой разработчик, у которого свое мнение, что именно делать при добавлении на счет. И все эти вопросы мы можем решить, используя события.

Определение и вызов событий

События объявляются в классе с помощью ключевого слова event, после которого указывается тип делегата, который представляет событие:

```
delegate void AccountHandler(string message);
event AccountHandler Notify;
```

В данном случае вначале определяется делегат AccountHandler, который принимает один параметр типа string. Затем с помощью ключевого слова event определяется событие с именем Notify, которое представляет делегат AccountHandler. Название для события может быть произвольным, но в любом случае оно должно представлять некоторый делегат.

Определив событие, мы можем его вызвать в программе как метод, используя имя события:

```
Notify("Произошло действие");
```

Поскольку событие Notify представляет делегат AccountHandler, который принимает один параметр типа string - строку, то при вызове события нам надо передать в него строку.

Однако при вызове событий мы можем столкнуться с тем, что событие равно null в случае, если для него не определен обработчик. Поэтому при вызове события лучше его всегда проверять на null. Например, так:

```
if(Notify != null) Notify("Произошло действие");
```

Или так:

```
Notify?.Invoke("Произошло действие");
```

В этом случае поскольку событие представляет делегат, то мы можем его вызвать с помощью метода Invoke(), передав в него необходимые значения для параметров.

Объединим все вместе и создадим и вызовем событие:

```
class Account
{
    public delegate void AccountHandler(string message);
    public event AccountHandler Notify;           // 1.Определение события
    public Account(int sum)
    {
        Sum = sum;
    }
    public int Sum { get; private set; }
    public void Put(int sum)
    {
        Sum += sum;
        Notify?.Invoke($"На счет поступило: {sum}"); // 2.Вызов события
    }
    public void Take(int sum)
    {
        if (Sum >= sum)
        {
            Sum -= sum;
            Notify?.Invoke($"Со счета снято: {sum}"); // 2.Вызов события
        }
        else
        {
            Notify?.Invoke($"Недостаточно денег на счете. Текущий баланс: {Sum}"); ;
        }
    }
}
```

Теперь с помощью события Notify мы уведомляем систему о том, что были добавлены средства и о том, что средства сняты со счета или на счете недостаточно средств.

Добавление обработчика события

С событием может быть связан один или несколько обработчиков. Обработчики событий - это именно то, что выполняется при вызове событий. Нередко в качестве обработчиков событий применяются методы. Каждый обработчик событий по списку параметров и возвращаемому типу должен соответствовать делегату, который представляет событие. Для добавления обработчика события применяется операция +=:

Notify += обработчик события;

Определим обработчики для события Notify, чтобы получить в программе нужные уведомления:

```
class Program
{
    static void Main(string[] args)
    {
        Account acc = new Account(100);
        acc.Notify += DisplayMessage; // Добавляем обработчик для события Notify
        acc.Put(20); // добавляем на счет 20
        Console.WriteLine($"Сумма на счете: {acc.Sum}");
        acc.Take(70); // пытаемся снять со счета 70
        Console.WriteLine($"Сумма на счете: {acc.Sum}");
        acc.Take(180); // пытаемся снять со счета 180
        Console.WriteLine($"Сумма на счете: {acc.Sum}");
        Console.Read();
    }
    private static void DisplayMessage(string message)
    {
        Console.WriteLine(message);
    }
}
```

```

    }
}

```

В данном случае в качестве обработчика используется метод `DisplayMessage`, который соответствует по списку параметров и возвращаемому типу делегату `AccountHandler`. В итоге при вызове события `Notify?.Invoke()` будет вызываться метод `DisplayMessage`, которому для параметра `message` будет передаваться строка, которая передается в `Notify?.Invoke()`. В `DisplayMessage` просто выводим полученное от события сообщение, но можно было бы определить любую логику.

Если бы в данном случае обработчик не был бы установлен, то при вызове события `Notify?.Invoke()` ничего не происходило, так как событие `Notify` было бы равно `null`.

Консольный вывод программы:

```

На счет поступило: 20
Сумма на счете: 120
Со счета снято: 70
Сумма на счете: 50
Недостаточно денег на счете. Текущий баланс: 50
Сумма на счете: 50

```

Теперь мы можем выделить класс `Account` в отдельную библиотеку классов и добавлять в любой проект.

Добавление и удаление обработчиков

Для одного события можно установить несколько обработчиков и потом в любой момент времени их удалить. Для удаления обработчиков применяется операция `-=`. Например:

```

class Program
{
    static void Main(string[] args)
    {
        Account acc = new Account(100);
        acc.Notify += DisplayMessage;    // добавляем обработчик DisplayMessage
        acc.Notify += DisplayRedMessage; // добавляем обработчик DisplayMessage
        acc.Put(20);    // добавляем на счет 20
        acc.Notify -= DisplayRedMessage; // удаляем обработчик DisplayRedMessage
        acc.Put(20);    // добавляем на счет 20
        Console.Read();
    }

    private static void DisplayMessage(string message)
    {
        Console.WriteLine(message);
    }

    private static void DisplayRedMessage(String message)
    {
        // Устанавливаем красный цвет символов
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(message);
        // Сбрасываем настройки цвета
        Console.ResetColor();
    }
}

```

```
}
```

В качестве обработчиков могут использоваться не только обычные методы, но также делегаты, анонимные методы и лямбда-выражения. Использование делегатов и методов:

```
static void Main(string[] args)
{
    Account acc = new Account(100);
    // установка делегата, который указывает на метод DisplayMessage
    acc.Notify += new ActionHandler(DisplayMessage);
    // установка в качестве обработчика метода DisplayMessage
    acc.Notify += DisplayMessage;      // добавляем обработчик DisplayMessage

    acc.Put(20);    // добавляем на счет 20
    Console.Read();
}

private static void DisplayMessage(string message)
{
    Console.WriteLine(message);
}
```

В данном случае разницы между двумя обработчиками никакой не будет.

Установка в качестве обработчика анонимного метода:

```
static void Main(string[] args)
{
    Account acc = new Account(100);
    acc.Notify += delegate (string mes)
    {
        Console.WriteLine(mes);
    };

    acc.Put(20);
    Console.Read();
}
```

Установка в качестве обработчика лямбда-выражения:

```
static void Main(string[] args)
{
    Account acc = new Account(100);
    acc.Notify += mes => Console.WriteLine(mes);

    acc.Put(20);
    Console.Read();
}
```

Управление обработчиками

С помощью специальных аксессоров add/remove мы можем управлять добавлением и удалением обработчиков. Как правило, подобная функциональность редко требуется, но тем не менее мы ее можем использовать. Например:

```
class Account
{
    public delegate void AccountHandler(string message);
    private event AccountHandler _notify;
```

```

public event AccountHandler Notify
{
    add
    {
        _notify += value;
        Console.WriteLine($"{value.Method.Name} добавлен");
    }
    remove
    {
        _notify -= value;
        Console.WriteLine($"{value.Method.Name} удален");
    }
}
public Account(int sum)
{
    Sum = sum;
}
public int Sum { get; private set; }
public void Put(int sum)
{
    Sum += sum;
    _notify?.Invoke($"На счет поступило: {sum}");
}

public void Take(int sum)
{
    if (Sum >= sum)
    {
        Sum -= sum;
        _notify?.Invoke($"Со счета снято: {sum}");
    }
    else
    {
        _notify?.Invoke($"Недостаточно денег на счете. Текущий баланс: {Sum}"); ;
    }
}
}

```

Теперь определение события разбивается на две части. Вначале просто определяется переменная, через которую мы можем вызывать связанные обработчики:

```
private event AccountHandler _notify;
```

Во второй части определяем аксессоры add и remove. Аксессор add вызывается при добавлении обработчика, то есть при операции +=. Добавляемый обработчик доступен через ключевое слово value. Здесь мы можем получить информацию об обработчике (например, имя метода через value.Method.Name) и определить некоторую логику. В данном случае для простоты просто выводится сообщение на консоль:

```

add
{
    _notify += value;
    Console.WriteLine($"{value.Method.Name} добавлен");
}

```

Блок remove вызывается при удалении обработчика. Аналогично здесь можно задать некоторую дополнительную логику:

```

remove
{
    _notify -= value;
    Console.WriteLine($"{value.Method.Name} удален");
}

```

Внутри класса событие вызывается также через переменную `_notify`. Но для добавления и удаления обработчиков в программе используется как раз `Notify`:

```
class Program
{
    static void Main(string[] args)
    {
        Account acc = new Account(100);
        acc.Notify += DisplayMessage; // добавляем обработчик DisplayMessage
        acc.Put(20); // добавляем на счет 20
        acc.Notify -= DisplayMessage; // удаляем обработчик DisplayRedMessage
        acc.Put(20); // добавляем на счет 20

        Console.Read();
    }
    private static void DisplayMessage(string message) => Console.WriteLine(message);
}
```

Класс данных события AccountEventArgs

Нередко при возникновении события обработчику события требуется передать некоторую информацию о событии. Например, добавим и в нашу программу новый класс `AccountEventArgs` со следующим кодом:

```
class AccountEventArgs
{
    // Сообщение
    public string Message { get; }
    // Сумма, на которую изменился счет
    public int Sum { get; }

    public AccountEventArgs(string mes, int sum)
    {
        Message = mes;
        Sum = sum;
    }
}
```

Данный класс имеет два свойства: `Message` - для хранения выводимого сообщения и `Sum` - для хранения суммы, на которую изменился счет.

Теперь применим класс `AccountEventArgs`, изменив класс `Account` следующим образом:

```
class Account
{
    public delegate void AccountHandler(object sender, AccountEventArgs e);
    public event AccountHandler Notify;
    public Account(int sum)
    {
        Sum = sum;
    }
    public int Sum { get; private set; }
    public void Put(int sum)
    {
        Sum += sum;
        Notify?.Invoke(this, new AccountEventArgs($"На счет поступило {sum}", sum));
    }
    public void Take(int sum)
    {
        if (Sum >= sum)
        {
            Sum -= sum;
        }
    }
}
```

```

        Notify?.Invoke(this, new AccountEventArgs($"Сумма {sum} снята со счета", sum));
    }
    else
    {
        Notify?.Invoke(this, new AccountEventArgs("Недостаточно денег на счете", sum)); ;
    }
}
}

```

По сравнению с предыдущей версией класса Account здесь изменилось только количество параметров у делегата и соответственно количество параметров при вызове события. Теперь они также принимают объект AccountEventArgs, который хранит информацию о событии, получаемую через конструктор.

Теперь изменим основную программу:

```

class Program
{
    static void Main(string[] args)
    {
        Account acc = new Account(100);
        acc.Notify += DisplayMessage;
        acc.Put(20);
        acc.Take(70);
        acc.Take(150);
        Console.Read();
    }
    private static void DisplayMessage(object sender, AccountEventArgs e)
    {
        Console.WriteLine($"Сумма транзакции: {e.Sum}");
        Console.WriteLine(e.Message);
    }
}

```

Задание

Написать C# программу на основе лабораторной №1,2,3,4,5 реализующую работу с классами по вариантам.

1. Заменить взаимодействие через интерфейс IEventHandler на взаимодействие через события.
2. Реализуемые события должны предоставлять изменяемые данные через EventArgs.

Разработанная программа должна быть консольной, позволяющей на выбор добавлять/изменять/удалять объекты классов, при наличии точно такого же экземпляра сообщать об этом пользователю. После совершения действия должно выводиться подтверждение о его выполнении. Кроме того должна быть возможность просмотреть все объекты заданного класса.

В отчёт обязательно построить диаграмму классов, описывающую иерархию созданных классов.

Варианты:

1. Студент, преподаватель, персона, заведующий кафедрой
2. Служащий, персона, рабочий, инженер
3. Рабочий, кадры, инженер, администрация
4. Деталь, механизм, изделие, узел
5. Организация, страховая компания, нефтегазовая компания, завод

6. Журнал, книга, печатное издание, учебник
7. Тест, экзамен, выпускной экзамен, испытание
8. Место, область, город, мегаполис
9. Игрушка, продукт, товар, молочный продукт
10. Квитанция, накладная, документ, счет
11. Автомобиль, поезд, транспортное средство, экспресс
12. Двигатель, двигатель внутреннего сгорания, дизель, реактивный двигатель
13. Республика, монархия, королевство, государство
14. Млекопитающее, парнокопытное, птица, животное
15. Корабль, пароход, парусник, корвет