Лабораторная работа №5

СОБЫТИЙНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Цель работы: ознакомить с механизмами событийно-ориентированного программирования на языке C#, такими как механизм обработки событий и исключительные ситуации.

Краткие теоретические сведения

Кроме свойств и методов классы и интерфейсы могут содержать делегаты и события. Делегаты представляют такие объекты, которые указывают на другие методы, т. е. делегаты – это указатели на методы. С помощью делегатов мы можем вызвать определённые методы в ответ на некоторые произошедшие действия. Таким образом, по своей сути делегаты раскрывают функционал функций обратного вызова.

Методы, на которые ссылаются делегаты, должны иметь те же параметры и тот же тип возвращаемого значения. Создадим два делегата:

```
delegate int Operation(int x, int y);
delegate void GetMessage();
```

Для объявления делегата используется ключевое слово **delegate**, после которого идёт возвращаемый тип, название и параметры. Первый делегат ссылается на функцию, которая в качестве параметров принимает два значения типа **int** и возвращает некоторое число. Второй делегат ссылается на метод без параметров, который ничего не возвращает.

Чтобы использовать делегат, нам надо создать его объект с помощью конструктора, в который мы передаём адрес метода, вызываемого делегатом. Чтобы вызвать метод, на который указывает делегат, надо использовать его метод **Invoke**. Кроме того, делегаты могут выполняться в асинхронном режиме, при этом нам не надо создавать второй поток, нам надо лишь вместо метода **Invoke** использовать пару методов **BeginInvoke/EndInvoke**:

```
class Program
{
    delegate void GetMessage(); // 1. Объявляем делегат
    static void Main(string[] args)
    {
        GetMessage del; // 2. Создаём переменную делегата
        if (DateTime.Now.Hour < 12)
        {
            del = GoodMorning; // 3. Присваиваем этой переменной
адрес метода
        }
```

```
        else
        {
            del = GoodEvening;
        }
        del.Invoke(); // 4. Вызываем метод
        Console.ReadLine();
    }
    private static void GoodMorning()
    {
        Console.WriteLine("Good Morning");
    }
    private static void GoodEvening()
    {
        Console.WriteLine("Good Evening");
    }
}
```

С помощью свойства **DateTime.Now.Hour** получаем текущий час. И в зависимости от времени в делегат передаётся адрес определённого метода. Обратите внимание, что методы эти имеют то же возвращаемое значение и тот же набор параметров, в данном случае их отсутствие, что и делегат.

Рассмотрим на примере другого делегата:

```
class Program
{
    delegate int Operation(int x, int y);

    static void Main(string[] args)
    {
        // присваивание адреса метода через конструктор
        Operation del = new Operation(Add); // делегат указывает
на метод Add
        int result = del.Invoke(4, 5);
        Console.WriteLine(result);
        del = Multiply; // теперь делегат указывает на метод
Multiply
        result = del.Invoke(4, 5);
        Console.WriteLine(result);
        Console.Read();
    }
    private static int Add(int x, int y)
    {
        return x + y;
    }
    private static int Multiply(int x, int y)
    {
        return x * y;
    }
}
```

Здесь описан способ присваивания делегату адреса метода через конструктор. И поскольку связанный метод, как и делегат, имеет два параметра, то при вызове делегата в метод **Invoke** мы передаём два параметра. Кроме того, так как метод возвращает значение типа **int**, то мы можем присвоить результат работы метода **Invoke** какой-нибудь переменной.

Метод **Invoke()** при вызове делегата можно опустить и использовать сокращённую форму:

```
del = Multiply; // теперь делегат указывает на метод Multiply
result = del(4, 5);
```

Таким образом, делегат можно вызывать как обычный метод, передавая ему аргументы.

Так же делегаты могут быть параметрами методов:

```
class Program
{
    delegate void GetMessage();
    static void Main(string[] args)
    {
        if (DateTime.Now.Hour < 12)
        {
            Show_Message(GoodMorning);
        }
        else
        {
            Show_Message(GoodEvening);
        }
        Console.ReadLine();
    }
    private static void Show_Message(GetMessage _del)
    {
        _del.Invoke();
    }
    private static void GoodMorning()
    {
        Console.WriteLine("Good Morning");
    }
    private static void GoodEvening()
    {
        Console.WriteLine("Good Evening");
    }
}
```

Данные примеры, возможно, не показывают истинной силы делегатов, так как нужные методы в данном случае можно вызвать и напрямую без всяких делегатов. Однако наиболее сильная сторона делегатов состоит в том, что они

позволяют создать функционал методов обратного вызова, уведомляя другие объекты о произошедших событиях.

Рассмотрим другой пример. Пусть у нас есть класс, описывающий счёт в банке:

```
class Account
{
    int _sum; // Переменная для хранения суммы
    int _percentage; // Переменная для хранения процента
    public Account(int sum, int percentage)
    {
        _sum = sum;
        _percentage = percentage;
    }
    public int CurrentSum
    {
        get { return _sum; }
    }
    public void Put(int sum)
    {
        _sum += sum;
    }
    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;
        }
    }
    public int Percentage
    {
        get { return _percentage; }
    }
}
```

Допустим, в случае вывода денег с помощью метода **Withdraw** нам надо как-то уведомлять об этом самого клиента и, может быть, другие объекты. Для этого создадим делегат **AccountStateHandler**. Чтобы использовать делегат, нам надо создать переменную этого делегата, а затем присвоить ему метод, который будет вызываться делегатом.

Итак, добавим в класс **Account** следующие строки:

```
class Account
{
    // Объявляем делегат
    public delegate void AccountStateHandler(string message);
    // Создаём переменную делегата
    AccountStateHandler del;
```

```
    // Регистрируем делегат
    public void RegisterHandler(AccountStateHandler _del)
    {
        del = _del;
    }

    // Далее остальные строки класса Account
```

Здесь фактически проделываются те же шаги, что были выше, и есть практически все, кроме вызова делегата. В данном случае делегат принимает параметр типа **string**. Теперь изменим метод **Withdraw** следующим образом:

```
public void Withdraw(int sum)
{
    if (sum <= _sum)
    {
        _sum -= sum;
        if (del != null)
            del("Сумма " + sum.ToString() + " снята со счёта");
    }
    else
    {
        if (del != null)
            del("Недостаточно денег на счёте");
    }
}
```

При снятии денег через метод **Withdraw** сначала проверяем, имеет ли делегат ссылку на какой-либо метод, иначе он имеет значение **null**. И если метод установлен, то вызываем его, передавая соответствующее сообщение в качестве параметра.

Теперь протестируем класс в основной программе:

```
class Program
{
    static void Main(string[] args)
    {
        // создаём банковский счёт
        Account account = new Account(200, 6);
        // Добавляем в делегат ссылку на метод Show_Message
        // а сам делегат передаётся в качестве параметра метода
RegisterHandler
        account.RegisterHandler(new
Account.AccountStateHandler(Show_Message));
        // Два раза подряд пытаемся снять деньги
        account.Withdraw(100);
        account.Withdraw(150);
        Console.ReadLine();
    }
```

```
    private static void Show_Message(String message)
    {
        Console.WriteLine(message);
    }
}
```

Запустив программу, получим два разных сообщения:

```
Сумма 100 снята со счёта
Недостаточно денег на счёте
```

Таким образом, создали механизм обратного вызова для класса **Account**, который срабатывает в случае снятия денег. Поскольку делегат объявлен внутри класса **Account**, то чтобы к нему получить доступ, используется выражение **Account.AccountStateHandler**.

Опять же может возникнуть вопрос: почему бы в коде метода **Withdraw()** не выводить сообщение о снятии денег? Зачем нужно задействовать какой-то делегат?

Дело в том, что не всегда есть доступ к коду классов. Например, часть классов может создаваться и компилироваться одним человеком, который не будет знать, как они будут использоваться. А использовать их будет другой разработчик.

Поэтому необходимо выводить сообщение на консоль. Однако для класса **Account** не важно, как это сообщение выводится. Классу **Account** даже не известно, что вообще будет делаться в результате списания денег. Он просто посылает уведомление об этом через делегат.

Так, через делегата можно выводить сообщение на консоль, создавая консольное приложение. Если создаём графическое приложение **Windows Forms** или **WPF**, то можно выводить сообщение в виде графического окна или записать при списании информацию об этом действии в файл, или отправить уведомление на электронную почту. В общем, любыми способами обработать вызов делегата. И способ обработки не будет зависеть от класса **Account**.

Хотя в примере делегат принимал адрес на один метод, в действительности он может указывать сразу на несколько. Кроме того, при необходимости можно указать ссылки на адреса определённых методов, чтобы они не вызывались при вызове делегата. Итак, изменим в классе **Account** метод **RegisterHandler** и добавим новый метод **UnregisterHandler**, который будет удалять методы из списка методов делегата:

```
public void RegisterHandler(AccountStateHandler _del)
{
    Delegate mainDel = System.Delegate.Combine(_del, del);
    del = mainDel as AccountStateHandler;
}
// Отмена регистрации делегата
public void UnregisterHandler(AccountStateHandler _del)
{
    Delegate mainDel = System.Delegate.Remove(del, _del);
    del = mainDel as AccountStateHandler;
}
```

Метод **Combine** в первом методе объединяет делегаты **_del** и **del** в один, который потом присваивается переменной **del**. Во втором методе метод **Remove** возвращает делегат, из списка вызовов которого удалён делегат **_del**. Теперь перейдём к основной программе:

```
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200, 6);
        Account.AccountStateHandler colorDelegate = new
Account.AccountStateHandler(Color_Message);
        // Добавляем в делегат ссылку на методы
        account.RegisterHandler(new
Account.AccountStateHandler(Show_Message));
        account.RegisterHandler(colorDelegate);
        // Два раза подряд пытаемся снять деньги
        account.Withdraw(100);
        account.Withdraw(150);
        // Удаляем делегат
        account.UnregisterHandler(colorDelegate);
        account.Withdraw(50);

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

В целях тестирования создадим ещё один метод – **Color_Message**, который выводит то же самое сообщение только красным цветом. Для первого делегата создаётся отдельная переменная. Но большой разницы между передачей обоих в метод **account.RegisterHandler** нет: просто в одном случае мы сразу передаём объект, создаваемый конструктором **account.RegisterHandler(new Account.AccountStateHandler(Show_Message))**.

Во втором случае создаём переменную и её уже передаём в метод **account.RegisterHandler(colorDelegate)**.

В строке **account.UnregisterHandler(colorDelegate)** этот метод удаляется из списка вызовов делегата, поэтому этот метод больше не будет срабатывать. Консольный вывод будет иметь следующую форму:

```
Сумма 150 снята со счёта
Сумма 150 снята со счёта
Недостаточно денег на счёте
Недостаточно денег на счёте
Сумма 50 снята со счёта
```

Также можно использовать сокращённую форму добавления и удаления делегатов. Для этого перепишем методы **RegisterHandler** и **UnregisterHandler** следующим образом:

```
public void RegisterHandler(AccountStateHandler _del)
{
    del += _del; // добавляем делегат
}
// Отмена регистрации делегата
public void UnregisterHandler(AccountStateHandler _del)
{
    del -= _del; // удаляем делегат
}
```

С помощью делегатов можно создавать механизм обратных вызовов в программе. Однако C# для той же цели предоставляет более удобные и простые конструкции под названием события, которые сигнализируют системе о том, что произошло определённое действие.

События объявляются в классе с помощью ключевого слова **event**, после которого идёт название делегата:

```
// Объявляем делегат
public delegate void AccountStateHandler(string message);
// Событие, возникающее при выводе денег
public event AccountStateHandler Withdrowed;
```

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры и возвращать тот же тип, что и делегат.

Итак, посмотрим на примере. Для этого возьмём класс **Account** и изменим его следующим образом:

```
class Account
{
    // Объявляем делегат
    public delegate void AccountStateHandler(string message);
    // Событие, возникающее при выводе денег
    public event AccountStateHandler Withdrowed;
    // Событие, возникающее при добавление на счёт
    public event AccountStateHandler Added;
    int _sum; // Переменная для хранения суммы
    int _percentage; // Переменная для хранения процента
    public Account(int sum, int percentage)
    {
        _sum = sum;
        _percentage = percentage;
    }
    public int CurrentSum
    {
        get { return _sum; }
    }
    public void Put(int sum)
    {
        _sum += sum;
        if (Added != null)
            Added("На счёт поступило " + sum);
    }
    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;
            if (Withdrowed != null)
                Withdrowed("Сумма " + sum + " снята со счёта");
        }
        else
        {
            if (Withdrowed != null)
                Withdrowed("Недостаточно денег на счёте");
        }
    }
    public int Percentage
    {
        get { return _percentage; }
    }
}
```

Здесь определены два события – **Withdrowed** и **Added**, которые объявлены как экземпляры делегата **AccountStateHandler**. Поэтому для обработки этих событий потребуется метод, принимающий строку в качестве параметра.

Затем в методах **Put** и **Withdraw** вызываем эти события. Перед вызовом проверяем, закреплены ли за ними обработчики:

```
if (Withdrowed != null)
```

Так как эти события представляют делегат **AccountStateHandler**, принимающий в качестве параметра строку, то и при вызове событий передаём в них строку.

Теперь используем события в основной программе:

```
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200, 6);
        // Добавляем обработчики события
        account.Added += Show_Message;
        account.Withdrowed += Show_Message;
        account.Withdraw(100);
        // Удаляем обработчик события
        account.Withdrowed -= Show_Message;
        account.Withdraw(50);
        account.Put(150);
        Console.ReadLine();
    }
    private static void Show_Message(string message)
    {
        Console.WriteLine(message);
    }
}
```

Для прикрепления обработчика события к определённому событию используется операция «+=» и соответственно для откреплення «– =»: событие += метод_обработчика_события. Опять обращаем внимание, что метод обработчика должен иметь такие же параметры, как и делегат события, и возвращать тот же тип. В итоге получим следующий консольный вывод:

```
Сумма 100 снята со счёта
На счёт поступило 150
```

Кроме использованного способа прикрепления обработчиков есть и другой с использованием делегата. Но оба способа будут равноценны:

```
account.Added += Show_Message;
account.Added += new Account.AccountStateHandler(Show_Message);
```

При создании графических приложений с помощью Windows Forms или WPF, можно столкнуться с обработчиками, которые в качестве параметра принимают аргумент типа **EventArgs**. Например, обработчик нажатия кнопки **private void button1_Click(object sender, System.EventArgs e){}**. Параметр **«e»**, будучи объектом класса **EventArgs**, содержит все данные события. Добавим в программу подобный класс, назовём его **AccountEventArgs** и дополним его следующим кодом:

```
class AccountEventArgs
{
    // Сообщение
    public string message;
    // Сумма, на которую изменился счёт
    public int sum;
    public AccountEventArgs(string _mes, int _sum)
    {
        message = _mes;
        sum = _sum;
    }
}
```

Данный класс имеет два поля: **message** – для хранения выводимого сообщения и **sum** – для хранения суммы, на которую изменился счёт.

Теперь применим класс **AccoutEventArgs**, изменив класс **Account** следующим образом:

```
class Account
{
    // Объявляем делегат
    public delegate void AccountStateHandler(object sender,
AccountEventArgs e);
    // Событие, возникающее при выводе денег
    public event AccountStateHandler Withdrowed;
    // Событие, возникающее при добавлении на счёт
    public event AccountStateHandler Added;
    int _sum; // Переменная для хранения суммы
    int _percentage; // Переменная для хранения процента
    public Account(int sum, int percentage)
    {
        _sum = sum;
        _percentage = percentage;
    }
```

```
    public int CurrentSum
    {
        get { return _sum; }
    }
    public void Put(int sum)
    {
        _sum += sum;
        if (Added != null)
            Added(this, new AccountEventArgs("На счёт поступило "
+ sum, sum));
    }
    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;
            if (Withdrowed != null)
                Withdrowed(this, new AccountEventArgs("Сумма " +
sum + " снята со счёта", sum));
        }
        else
        {
            if (Withdrowed != null)
                Withdrowed(this, new
AccountEventArgs("Недостаточно денег на счёте", sum));
        }
    }
    public int Percentage
    {
        get { return _percentage; }
    }
}
```

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 account = new Account(200, 6);
        // Добавляем обработчики события
        account.Added += Show_Message;
        account.Withdrowed += Show_Message;
        account.Withdraw(100);
```

```
        // Удаляем обработчик события
        account.Withdrowed -= Show_Message;
        account.Withdraw(50);
        account.Put(150);
        Console.ReadLine();
    }
    private static void Show_Message(object sender,
AccountEventArgs e)
    {
        Console.WriteLine("Сумма транзакции: {0}", e.sum);
        Console.WriteLine(e.message);
    }
}
```

По сравнению с предыдущим вариантом здесь изменяем только количество параметров и сущность их использования в обработчике **Show_Message**.

С делегатами тесно связано понятие анонимных методов. Анонимные методы представляют сокращённую запись методов. Например, в лабораторной работе №4 было создано событие и обработчик к нему, который выглядел следующим образом:

```
private static void Show_Message(object sender, AccountEventArgs
e)
{
    Console.WriteLine("Сумма транзакции: {0}", e.sum);
    Console.WriteLine(e.message);
}
```

Иногда такие методы нужны для обработки только одного события и больше ценности не представляют, и нигде не используются. Анонимные методы позволяют встроить код там, где он вызывается, например:

```
Account account = new Account(200, 6);
// Добавляем обработчики события
account.Added += delegate (object sender, AccountEventArgs e)
{
    Console.WriteLine("Сумма транзакции: {0}", e.sum);
    Console.WriteLine(e.message);
};
```

Практически это тот же самый метод, только встроенный в код. Встраивание происходит с помощью ключевого слова **delegate**, после которого идёт список параметров и далее сам код анонимного метода. Итоговый результат будет такой же, как будто мы подключали обработчик события **Show_Message**.

И важно отметить, что в отличие от блока методов или условных и циклических конструкций, блок анонимных методов должен заканчиваться точкой с запятой после закрывающей фигурной скобки.

Если для анонимного метода не требуется параметров, то он используется без скобок:

```
delegate void GetMessage();
static void Main(string[] args)
{
    GetMessage message = delegate
    {
        Console.WriteLine("анонимный делегат");
    };
    message();

    Console.Read();
}
```

Задания на лабораторную работу

Переопределив с помощью наследования событие, реализуйте обработку ошибок для лабораторной работы №4:

1) StackOverflowException;
2) ArrayTypeMismatchException;
3) DivideByZeroException;
4) IndexOutOfRangeException;
5) InvalidCastException;
6) OutOfMemoryException;
7) OverflowException.

Контрольные вопросы

1. Что понимается под термином «событие»?
2. Являются ли события членами классов?
3. Какое ключевое слово языка C# используется для описания событий?
4. Опишите механизм поддержки событий языка C#.
5. Приведите синтаксис описания события в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
6. Что понимается под термином «широковещательное событие»?
7. На основе какого механизма языка C# строятся широковещательные события?
8. Приведите синтаксис описания широковещательного события в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

9. Что понимается под термином «исключительная ситуация (исключение)»?

10. В чём состоит значение механизма исключений в языке C#?

11. Какие операторы языка C# используются для обработки исключений?

12. Какие операторы языка C# являются важнейшими для обработки исключений?

13. Приведите синтаксис блока try…catch в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

14. Приведите пять видов основных системных исключений.

15. Необходимо ли обеспечивать соответствие типов исключения в операторе catch типу перехватываемого исключения?

16. Что происходит в случае неудачного перехвата исключения?

17. В каком случае возможно использование оператора catch языка C# без параметров?

18. Каким образом осуществляется возврат в программу после обработки исключительной ситуации?

19. Какой оператор языка C# используется для обеспечения возврата в программу после обработки исключения?

20. Приведите синтаксис блока finally в составе оператора try…catch в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

21. Зависит ли вызов блока finally от наличия исключения?

22. Какие способы генерации исключений Вам известны?

23. Что является источником автоматически генерируемых неявных исключений?

24. Каким образом возможно осуществить явную генерацию исключений?

25. Какой оператор языка C# используется для явной генерации исключений?

26. Приведите синтаксис оператора throw в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

27. Каким образом осуществляется повторный перехват исключений в языке C#?

28. Возможно ли создавать специализированные исключения для обработки ошибок в коде пользователя?

29. Какой системный класс является базовым для создания исключений?

30. На основе какого системного класса осуществляется генерация пользовательских исключений?

31. Необходима ли явная реализация классов, наследуемых от системных исключений?

32. Каким образом обеспечивается обращение к свойствам и методам системных исключений?