Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»
Филиал
«Минский радиотехнический колледж»

Учебный предмет
«Конструирование программ и языки программирования»

Инструкция
по выполнению лабораторной работы №18
«Разработка, отладка и испытание классов с событиями»

Минск 2024 г.

Лабораторная работа № 18

Тема работы: «Разработка, отладка и испытание классов с событиями»

1 Цель работы

Сформировать умения разрабатывать классы с событиями и использовать их в программах.

2 Задание

Номер варианта соответствует номеру по списку в журнале.

Добавить к классам и объектам из 4 лабораторной работы обработчики событий.

3 Оснащение работы

ПК, среда Visual Studio 2019, MSword.

4 Основные теоретические сведения

События позволяют сигнализируют системе о том, что произошло определенное действие.

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

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public event AccountStateHandler Withdrawn;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

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

Итак, посмотрим на примере. Для этого возьмем класс Account из прошлой темы и изменим его следующим образом:

```
class Account
{
    // Объявляем делегат
    public delegate void AccountStateHandler(string
message);
    // Событие, возникающее при выводе денег
    public event AccountStateHandler Withdrawn;
    // Событие, возникающее при добавление на счет
    public event AccountStateHandler Added;

    int _sum; // Переменная для хранения суммы

    public Account(int sum)
    {
        _sum = sum;
```

```
    }

    public int CurrentSum
    {
        get { return _sum; }
    }

    public void Put(int sum)
    {
        _sum += sum;
        if (Added != null)
            Added($"На счет поступило {sum}");
    }
    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;
            if (Withdrawn != null)
                Withdrawn($"Сумма {sum} снята со
счета");
        }
        else
        {
            if (Withdrawn != null)
                Withdrawn("Недостаточно денег на
счете");
        }
    }
}
```

Здесь мы определили два события: Withdrawn и Added. Оба события объявлены как экземпляры делегата AccountStateHandler, поэтому для обработки этих событий потребуется метод, принимающий строку в качестве параметра.

Затем в методах Put и Withdraw мы вызываем эти события. Перед вызовом мы проверяем, закреплены ли за этими событиями обработчики (if (Withdrawn != null)). Так как эти события представляют делегат AccountStateHandler, принимающий в качестве параметра строку, то и при вызове событий мы передаем в них строку.

Теперь используем события в основной программе:

```
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200);
```

```
            // Добавляем обработчики события
            account.Added += Show_Message;
            account.Withdrawn += Show_Message;

            account.Withdraw(100);
            // Удаляем обработчик события
            account.Withdrawn -= 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 определенному событию используется операция += и соответственно для откреплення - операция -=: событие += метод_обработчика_события. Опять же обращаю внимание, что метод обработчика должен иметь такие же параметры, как и делегат события, и возвращать тот же тип. В итоге мы получим следующий консольный вывод:

Сумма 100 снята со счета

На счет поступило 150

Кроме использованного выше 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);
```

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 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 Withdrawn;
    // Событие, возникающее при добавлении на счет
    public event AccountStateHandler Added;

    int _sum; // Переменная для хранения суммы

    public Account(int sum)
    {
        _sum = sum;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }

    public void Put(int sum)
    {
        _sum += sum;
        if (Added != null)
            Added(this, new AccountEventArgs($"На счет
поступило {sum}", sum));
    }
    public void Withdraw(int sum)
    {
        if (_sum >= sum)
        {
            _sum -= sum;
            if (Withdrawn != null)
                Withdrawn(this, new
AccountEventArgs($"Сумма {sum} снята со счета", sum));
        }
        else
```

```
        {
            if (Withdrawn != null)
                Withdrawn(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 account = new Account(200);
        // Добавляем обработчики события
        account.Added += Show_Message;
        account.Withdrawn += Show_Message;

        account.Withdraw(100);
        // Удаляем обработчик события
        account.Withdrawn -= Show_Message;

        account.Withdraw(50);
        account.Put(150);

        Console.ReadLine();
    }
    private static void Show_Message(object sender,
AccountEventArgs e)
    {
        Console.WriteLine($"Сумма транзакции:
{e.Sum}");
        Console.WriteLine(e.Message);
    }
}
```

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

Анонимные методы

С делегатами тесно связаны Анонимные методы. Анонимные методы используются для создания экземпляров делегатов.

Определение анонимных методов начинается с ключевого слова delegate, после которого идет в скобках список параметров и тело метода в фигурных скобках:

```
delegate(параметры)
{
    // инструкции
}
```

Например:

```
class Program
{
    delegate void MessageHandler(string message);
    static void Main(string[] args)
    {
        MessageHandler handler = delegate(string mes)
        {
            Console.WriteLine(mes);
        };
        handler("hello world!");

        Console.Read();
    }
}
```

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата, как в данном случае переменная handler представляет анонимный метод. И через эту переменную делегата можно вызвать данный анонимный метод.

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.

Другой пример анонимных методов - передача в качестве аргумента для параметра, который представляет делегат:

```
class Program
{
    delegate void MessageHandler(string message);
    static void Main(string[] args)
    {
        ShowMessage("hello!", delegate(string mes)
        {
            Console.WriteLine(mes);
        });

        Console.Read();
    }
```

```
    static void ShowMessage(string mes, MessageHandler
handler)
    {
        handler(mes);
    }
}
```

Если анонимный метод использует параметры, то они должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе опустить параметры:

```
class Program
{
    delegate void MessageHandler(string message);
    static void Main(string[] args)
    {
        MessageHandler handler = delegate
        {
            Console.WriteLine("анонимный метод");
        };
        handler("hello world!");    // анонимный метод

        Console.Read();
    }
}
```

То есть если анонимный метод содержит параметры, они обязательно должны соответствовать параметрам делегата. Либо анонимный метод вообще может не содержать никаких параметров, тогда он соответствует любому делегату, который имеет тот же тип возвращаемого значения.

При этом параметры анонимного метода не могут быть опущены, если один или несколько параметров определены с модификатором out.

Также, как и обычные методы, анонимные могут возвращать результат:

```
delegate int Operation(int x, int y);
static void Main(string[] args)
{
    Operation operation = delegate (int x, int y)
    {
        return x + y;
    };
    int d = operation(4, 5);
    Console.WriteLine(d);       // 9
    Console.Read();
}
```

При этом анонимный метод имеет доступ ко всем переменным, определенным во внешнем коде:

```
delegate int Operation(int x, int y);
static void Main(string[] args)
{
    int z = 8;
    Operation operation = delegate (int x, int y)
    {
        return x + y + z;
    };
    int d = operation(4, 5);
    Console.WriteLine(d);       // 17
    Console.Read();
}
```

В каких ситуациях используются анонимные методы? Когда нам надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется. Иногда такие методы нужны для обработки одного события и больше ценности не представляют и нигде не применяются. Например, применим анонимные методы для обработки событий:

```
delegate void AccountStateHandler(object sender,
AccountEventArgs e);
class AccountEventArgs
{
    public string Message { get;}
    public int Sum { get;}
    public AccountEventArgs(string message, int sum)
    {
        Message = message; Sum = sum;
    }
}
class Account
{
    int _sum;
    public event AccountStateHandler Added;
    public event AccountStateHandler Withdrawn;
    public Account(int sum)
    {
        _sum = sum;
    }
    public void Put(int sum)
    {
        _sum += sum;
        if (Added != null) Added(this,
            new AccountEventArgs($"На счет пришло
{sum}", sum));
    }
    public void Withdraw(int sum)
```

```
    {
        if(_sum >=sum)
        {
            _sum -= sum;
            if (Withdrawn != null)
                Withdrawn(this, new
AccountEventArgs($"Со счета снято {sum}", sum));
        }
        else
        {
            if (Withdrawn != null)
                Withdrawn(this, new
AccountEventArgs("На счете недостаточно средств", 0));
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200);
        // Добавляем обработчики события
        account.Added += delegate (object sender,
AccountEventArgs e)
        {
            Console.WriteLine($"Сумма транзакции:
{e.Sum}");
            Console.WriteLine(e.Message);
        };
        account.Put(230);

        Console.Read();
    }
}
```

В данном случае, допустим, мы не собираемся использовать код анонимного метода в каких-то других ситуациях, кроме обработки события. Поэтому нет смысла оформлять его в полноценный метод.

Лямбда-выражения представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы.

Ламбда-выражения имеют следующий синтаксис: слева от лямбда-оператора => определяется список параметров, а справа блок выражений, использующий эти параметры: (список_параметров) => выражение. Например:

```
class Program
{
    delegate int Operation(int x, int y);
    static void Main(string[] args)
    {
        Operation operation = (x, y) => x + y;
        Console.WriteLine(operation(10, 20));       //
30
        Console.WriteLine(operation(40, 20));       //
60
        Console.Read();
    }
}
```

Здесь код (x, y) => x + y; представляет лямбда-выражение, где x и y - это параметры, а x + y - выражение. При этом нам не надо указывать тип параметров, а при возвращении результата не надо использовать оператор return.

При этом надо учитывать, что каждый параметр в лямбда-выражении неявно преобразуется в соответствующий параметр делегата, поэтому типы параметров должны быть одинаковыми. Кроме того, количество параметров должно быть таким же, как и у делегата. И возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата. То есть в данном случае использованное лямбда-выражение соответствует делегату Operation как по типу возвращаемого значения, так и по типу и количеству параметров.

Если лямбда-выражение принимает один параметр, то скобки вокруг параметра можно опустить:

```
class Program
{
    delegate int Square(int x); // объявляем делегат,
принимающий int и возвращающий int
    static void Main(string[] args)
    {
        Square square = i => i * i; // объекту делегата
присваивается лямбда-выражение

        int z = square(6); // используем делегат
        Console.WriteLine(z); // выводит число 36
        Console.Read();
    }
}
```

Бывает, что параметров не требуется. В этом случае вместо параметра в лямбда-выражении используются пустые скобки. Также бывает, что лямбда-выражение не возвращает никакого значения:

```
class Program
{
```

```
    delegate void Hello(); // делегат без параметров
    static void Main(string[] args)
    {
        Hello hello1 = () =>
Console.WriteLine("Hello");
        Hello hello2 = () =>
Console.WriteLine("Welcome");
        hello1();       // Hello
        hello2();       // Welcome
        Console.Read();
    }
}
```

В данном случае лямда-выражение ничего не возвращает, так как после лямбда-оператора идет действие, которое ничего не возвращает.

Как видно, из примеров выше, нам необязательно указывать тип параметров у лямбда-выражения. Однако, нам обязательно нужно указывать тип, если делегат, которому должно соответствовать лямбда-выражение, имеет параметры с модификаторамиref и out:

```
class Program
{
    delegate void ChangeHandler(ref int x);
    static void Main(string[] args)
    {
        int x = 9;
        ChangeHandler ch = (ref int n) => n = n * 2;
        ch(ref x);
        Console.WriteLine(x);   // 18
        Console.Read();
    }
}
```

Лямбда-выражения также могут выполнять другие методы:

```
class Program
{
    delegate void Hello(); // делегат без параметров
    static void Main(string[] args)
    {
        Hello message = () => Show_Message();
        message();
    }
    private static void Show_Message()
    {
        Console.WriteLine("Привет мир!");
    }
}
```

Лямбда-выражения в обработке событий

Одним из частых примеров использования лямбда-выражений является обработка событий. Возьмем класс Account из прошлой темы:

```
delegate void AccountStateHandler(object sender,
AccountEventArgs e);

    class AccountEventArgs
    {
        public string Message { get;}
        public int Sum { get;}
        public AccountEventArgs(string message, int
sum)
        {
            Message = message; Sum = sum;
        }
    }
    class Account
    {
        int _sum;
        public event AccountStateHandler Added;
        public event AccountStateHandler Withdrawn;
        public Account(int sum)
        {
            _sum = sum;
        }
        public void Put(int sum)
        {
            _sum += sum;
            if (Added != null) Added(this,
                new AccountEventArgs($"На счет пришло
{sum}", sum));
        }
        public void Withdraw(int sum)
        {
            if(_sum >=sum)
            {
                _sum -= sum;
                if (Withdrawn != null)
                    Withdrawn(this, new
AccountEventArgs($"Со счета снято {sum}", sum));
            }
            else
            {
                if (Withdrawn != null)
                    Withdrawn(this,
```

```
new AccountEventArgs("На счете недостаточно средств",
0));} } }
```

И перепишем прикрепление обработчика события с помощью лямбды-выражения:

```
static void Main(string[] args)
{
    Account account = new Account(100);
    account.Added += (sender, e) =>
    {
        Console.WriteLine($"Сумма транзакции:
{e.Sum}");
        Console.WriteLine(e.Message);
    };
    account.Put(200);
    account.Put(109);
}
```

Поскольку здесь используется несколько параметров, то они заключаются в скобки. И так как в теле лямбда-выражения применяется несколько выражений, то они заключаются в блок из фигурных скобок.

Лямбда-выражения как аргументы методов

Как и делегаты, лямбда-выражения можно передавать в качестве аргументов методу для тех параметров, которые представляют делегат, что довольно удобно:

```
class Program
{
    delegate bool IsEqual(int x);

    static void Main(string[] args)
    {
        int[] integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // найдем сумму чисел больше 5
        int result1 = Sum(integers, x => x > 5);
        Console.WriteLine(result1); // 30

        // найдем сумму четных чисел
        int result2 = Sum(integers, x => x % 2 == 0);
        Console.WriteLine(result2);  //20

        Console.Read();
    }

    private static int Sum (int[] numbers, IsEqual
func)
    {
```

```
        int result = 0;
        foreach(int i in numbers)
        {
            if (func(i))
                result += i;
        }
        return result;
    }
}
```

Метод Sum принимает в качестве параметра массив чисел и делегат IsEqual и возвращает сумму чисел массива в виде объекта int. В цикле проходим по всем числам и складываем их. Причем складываем только те числа, для которых делегат IsEqual funcвозвращает true. То есть делегат IsEqual здесь фактически задает условие, которому должны соответствовать значения массива. Но на момент написания метода Sum нам неизвестно, что это за условие.

При вызове метода Sum ему передается массив и лямбда-выражение:

```
int result1 = Sum(integers, x => x > 5);
```

То есть параметр x здесь будет представлять число, которое передается в делегат:

```
if (func(i))
```

А выражение `x > 5` представляет условие, которому должно соответствовать число. Если число соответствует этому условию, то лямбда-выражение возвращает true, а переданное число складывается с другими числами.

Подобным образом работает второй вызов метода Sum, только здесь уже идет проверка числа на четность, то есть если остаток от деления на 2 равен нулю:

```
int result2 = Sum(integers, x => x % 2 == 0);
```

Ковариантность и контравариантность делегатов

Делегаты могут быть ковариантными и контравариантными. Ковариантность делегата предполагает, что возвращаемым типом может быть более производный тип. Контрвариантность делегата предполагает, что типом параметра может быть более универсальный тип.

Ковариантность

Ковариантность позволяет возвращать из метода объект, тип которого является производным от типа, возвращаемого делегатом.

Допустим, имеется следующая структура классов:

```
class Person
{
    public string Name { get; set; }
}
class Client : Person { }
```

В этом случае ковариантность делегата может выглядеть следующим образом:

```
delegate Person PersonFactory(string name);
static void Main(string[] args)
{
    PersonFactory personDel;
    personDel = BuildClient; // ковариантность
    Person p = personDel("Tom");
    Console.WriteLine(p.Name);
    Console.Read();
}
private static Client BuildClient(string name)
{
    return new Client {Name = name};
}
```

То есть здесь делегат возвращает объект Person. Однако благодаря ковариантности данный делегат может указывать на метод, который возвращает объект производного типа, например, Client.

Контрвариантность

Контрвариантность предполагает возможность передавать в метод объект, тип которого является более универсальным по отношению к типу параметра делегата. Например, возьмем выше определенные классы Person и Client и используем их в следующем примере:

```
delegate void ClientInfo(Client client);
static void Main(string[] args)
{
    ClientInfo clientInfo = GetPersonInfo; //
контравариантность
    Client client = new Client{Name = "Alice"};
    clientInfo(client);
    Console.Read();
}
private static void GetPersonInfo(Person p)
{
    Console.WriteLine(p.Name);
}
```

Несмотря на то, что делегат в качестве параметра принимает объект Client, ему можно присвоить метод, принимающий в качестве параметра объект базового типа Person. Может показаться на первый взгляд, что здесь есть некоторое противоречие, то есть использование более универсального тип вместо более производного. Однако в реальности в делегат при его вызове мы все равно можем передать только объекты типа Client, а любой объект типа Client является объектом типа Person, который используется в методе.

Ковариантность и контравариантность в обобщенных делегатах

Обобщенные делегаты также могут быть ковариантными и контравариантными, что дает нам больше гибкости в их использовании.

Например, возьмем следующую иерархию классов:

```
class Person
{
    public string Name { get; set; }
    public virtual void Display() =>
Console.WriteLine($"Person {Name}");
}
class Client : Person
{
    public override void Display() =>
Console.WriteLine($"Client {Name}");
}
```

Теперь объявим и используем ковариантный обобщенный делегат:

```
class Program
{
    delegate T Builder<out T>(string name);
    static void Main(string[] args)
    {
        Builder<Client> clientBuilder = GetClient;
        Builder<Person> personBuilder1 =
clientBuilder;     // ковариантность
        Builder<Person> personBuilder2 =
GetClient;         // ковариантность

        Person p = personBuilder1("Tom"); // вызов
делегата
        p.Display(); // Client: Tom

        Console.Read();
    }
    private static Person GetPerson(string name)
    {
        return new Person {Name = name};
    }
    private static Client GetClient(string name)
    {
        return new Client {Name = name};
    }
}
```

Благодаря использованию out мы можем присвоить делегату типа Builder<Person> делегат типа Builder<Client> или ссылку на метод, который возвращает значение Client.

Рассмотрим контравариантный обобщенный делегат:

```
class Program
{
    delegate void GetInfo<in T>(T item);
    static void Main(string[] args)
    {
        GetInfo<Person> personInfo = PersonInfo;
        GetInfo<Client> clientInfo =
personInfo;      // контравариантность

        Client client = new Client { Name = "Tom" };
        clientInfo(client); // Client: Tom

        Console.Read();
    }
    private static void PersonInfo(Person p) =>
p.Display();
    private static void ClientInfo(Client cl) =>
cl.Display();
}
```

Использование ключевого слова in позволяет присвоить делегат с более универсальным типом (GetInfo<Person>) делегату с производным типом (GetInfo<Client>).

Как и в случае с обобщенными интерфейсами параметр ковариантного типа применяется только к типу значения, которые возвращается делегатом. А параметр контравариантного типа применяется только к входным аргументам делегата.

5. Порядок выполнения работы

1. Выделить ключевые моменты задачи.
2. Построить алгоритм и теоретическую объектную модель решения задачи.
3. Запрограммировать полученные алгоритмы и объектную модель.

6. Форма отчета о работе

Лабораторная работа № ___

Номер учебной группы _____

Фамилия, инициалы учащегося _____

Дата выполнения работы _____

Тема работы: _____

Цель работы: _____

Оснащение работы: _____

Результат выполнения работы: _____

7. Контрольные вопросы и задания

1. Что такое событие?
2. Где и для чего используется событие?
3. Как можно вызвать событие?
4. Что такое анонимный метод?
5. Где и для чего используется анонимный метод?
6. Что такое лямбда-выражение?
7. Чем лучше лямбда-выражение анонимных методов?
8. Что такое ковариантность делегатов?
9. Что такое контравариантность делегатов?

8. Рекомендуемая литература

1. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# / Дж. Рихтер. СПб. : Изд-во Питер, 2021. 896 с.

2. Прайс, М. Дж. C# 10 и .NET 6. Современная кросс-платформенная разработка / М. Дж. Прайс. СПб : Изд-во Питер, 2023. 848 с.

3. Васильев, А.Н. Программирование на C# для начинающих. Особенности языка / А.Н. Васильев. М. : Эксмо, 2022. 528 с.

4. Фримен, А. ASP.NET Core 3 с примерами на C# для профессионалов / А. Фримен. СПб. : Изд-во Вильямс, 2021. 1184 с.