Программирование и обработка графического интерфейса

Лекция №5: Интерфейсы, делегаты, события.

Интерфейс

Интерфейсы — это абстрактные классы, которые не могут содержать реализации методов и свойств. Они используются для определения набора методов и свойств, которые должны быть реализованы в классах, которые реализуют этот интерфейс.

Интерфейс

Интерфейс определяется с помощью ключевого слова Interface

```
//интерфейс для логгирования
public interface ILogger
{
    void Log(string message);
    void Error(string errorMessage);
    string Name { get; }
}
```

Интерфейс

Так же, как и наследования для реализации интерфейса используется указатель «:» перед классом, после чего идет имя интерфейса.

```
//Класс, реализующий методы интерфейса
public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }

    public void Error(string errorMessage)
    {
        Console.WriteLine("ERROR: " + errorMessage);
    }

    public string Name => "Console Logger";
}
```

Интерфейс

Интерфейсы можно использовать для динамического создания объектов поддерживающего полиморфизм.

```
public interface IPrintable
{
    void Print();
}

public class ConsolePrint : IPrintable
{
    //реализация интерфейса
    public void Print()
    {
        Console.WriteLine("Сообщение выведено в консоль");
    }
}
```

```
IPrintable ConsolePrinter = new ConsolePrint();
ConsolePrinter.Вывести();
// выводит "Сообщение выведено в консоль"
```

Интерфейс

Интерфейсы могут наследоваться от других интерфейсов.

```
public interface IPrintable
{
    void Print();
}

public interface IEditable : IPrintable
{
    void Edit();
}
```

Интерфейс

Множественная реализация:

```
public interface ILogger
{
    void Log(string message);
    void Error(string errorMessage);

    string Name { get; }
}

public interface IWriter
{
    void Write(string text);
    void WriteLine(string text);
}

public interface IPrinter
{
    void Print(string text);
}
```

```
public class Printer : ILogger, IWriter, IPrinter
{
    public void Log(string message)
    {
        Console.WriteLine("LOG: " + message);
    }

    public void Error(string errorMessage)
    {
        Console.WriteLine("ERROR: " + errorMessage);
    }

    public string Name => "Printer";

    public void Write(string text)
    {
        Console.Write(text);
    }

    public void WriteLine(string text)
    {
        Console.WriteLine(text);
    }

    public void Print(string text)
    {
        Console.WriteLine(text);
    }
}
```

Интерфейс

Неявная реализация интерфейса:

```
Ссылок: 2
interface IMovable
{
    Ссылок: 2
    void Move();
}
Ссылок: 2
class Person : IMovable
{
    Ссылок: 1
    public void Move() => Console.WriteLine("Человек идет");
}
Ссылок: 2
struct Car : IMovable
{
    Ссылок: 2
    public void Move() => Console.WriteLine("Машина едет");
}
```

```
Ссылок: 0
public MainWindow()
{
    InitializeComponent();

    Person person = new Person();
    Car car = new Car();
    DoAction(person);
    DoAction(car);
}

Ссылок: 2
void DoAction(IMovable movable) => movable.Move();
```

Интерфейс

Явная реализация интерфейса:

```
Ссылок: 2
interface IMovable
{
    Ссылок: 2
    void Move();
}
Ссылок: 2
class Person : IMovable
{
    //Вызов метода от интерфейса
    Ссылок: 2
    void IMovable.Move() => Console.WriteLine("Человек идет");
}
```

```
Ссылок: 0
public MainWindow()
{
    InitializeComponent();

    Person person = new Person();

    //ошибка - у класса Person нет метода Move
    person.Move();
    //небезопасное приведение
    ((IMovable)person).Move();
    //безопасное приведение
    if (person is IMovable movable_person) movable_person.Move();
    //создание на основе интерфейса
    IMovable person2 = new Person();
    person2.Move();
}
```

Интерфейс

Класс реализует интерфейсы с одинаковой сигнатурой:

```
Ссылок: 1
interface ISchool
{
    Ссылок: 1
    void Study();
}
Ссылок: 1
interface IUniversity
{
    Ссылок: 1
    void Study();
}
Ссылок: 1
class Person : ISchool, IUniversity
{
    Ссылок: 0
    public void Study() => Console.WriteLine("Учеба в школе или в университете");
}
```

```
Ссылок: 1
interface ISchool
{
    Ссылок: 1
    void Study();
}
Ссылок: 1
interface IUniversity
{
    Ссылок: 1
    void Study();
}
Ссылок: 1
class Person : ISchool, IUniversity
{
    Ссылок: 1
    void ISchool.Study() => Console.WriteLine("Учеба в школе");
    Ссылок: 1
    void IUniversity.Study() => Console.WriteLine("Учеба в университете");
}
```

Интерфейс

Можно не реализовать методы, сделав их абстрактными, переложив право их реализации на производные классы:

```
Ссылок: 1
interface IMovable
{
    Ссылок: 2
    void Move();
}
Ссылок: 1
abstract class Person : IMovable
{
    Ссылок: 2
    public abstract void Move();
}
Ссылок: 0
class Driver : Person
{
    Ссылок: 2
    public override void Move() => Console.WriteLine("Шофер ведет машину");
}
```

Интерфейс

Класс наследник может переопределить реализацию интерфейса:

```
Ссылок: 2
interface IAction
{
    Ссылок: 4
    void Move();
}
Ссылок: 2
class BaseAction : IAction
{
    Ссылок: 4
    public virtual void Move() => Console.WriteLine("Move in Base");
}
Ссылок: 2
class AdvancedAction : BaseAction
{
    Ссылок: 4
    public override void Move() => Console.WriteLine("Move in Advanced");
}
```

```
BaseAction action1 = new AdvancedAction();
action1.Move();            // Move in Advanced

IAction action2 = new AdvancedAction();
action2.Move();            // Move in Advanced
```

Интерфейс

Через ключевое слово new можно скрыть реализацию в дочернем классе:

```
BaseAction action1 = new AdvancedAction();
action1.Move();             // Move in Base

IAction action2 = new AdvancedAction();
action2.Move();             // Move in Advanced

AdvancedAction action3 = new AdvancedAction();
action3.Move();             // Move in Advanced
```

```
Ссылок: 1
interface IAction
{
    Ссылок: 1
    void Move();
}
Ссылок: 1
class BaseAction : IAction
{
    Ссылок: 1
    public void Move() => Console.WriteLine("Move in Base");

}
Ссылок: 0
class AdvancedAction : BaseAction
{
    Ссылок: 0
    public new void Move() => Console.WriteLine("Move in Advanced");
}
```

Интерфейс

Повторная реализация интерфейса в классе-наследнике:

```
BaseAction action1 = new AdvancedAction();
action1.Move();             // Move in Base

IAction action2 = new AdvancedAction();
action2.Move();             // Move in Advanced

AdvancedAction action3 = new AdvancedAction();
action3.Move();             // Move in Advanced
```

```
Ссылок: 3
interface IAction
{
    Ссылок: 5
    void Move();
}
Ссылок: 2
class BaseAction : IAction
{
    Ссылок: 3
    public void Move() => Console.WriteLine("Move in BaseAction");
}
Ссылок: 4
class AdvancedAction : BaseAction, IAction
{
    Ссылок: 3
    public new void Move() => Console.WriteLine("Move in Advanced");
}
```

Интерфейс

Как и классы, интерфейсы могут использовать в своей основе обобщенные параметры:

```
interface IUser<T>
{

    T Id { get; }
}
Ссылок: 1
class User<T> : IUser<T>
{
    Ссылок: 2
    public T Id { get; }
    Ссылок: 0
    public User(T id) => Id = id;
}


IUser<int> user1 = new User<int>(6789);
Console.WriteLine(user1.Id);    // 6789

IUser<string> user2 = new User<string>("12345");
Console.WriteLine(user2.Id);    // 12345
```

Делегаты

Делегаты — это типы, которые представляют ссылки на методы. Они позволяют хранить и вызывать методы как объекты. Делегат может ссылаться на любой метод с подходящей сигнатурой.

Делегаты

Для объявления делегата используется ключевое слово **delegate**, после которого идет возвращаемый тип, название и параметры:

```
[доступность] delegate [возвращаемый тип] Название([параметры]);
```

```
delegate void Message();

private delegate string Decode(string param1, int param2);
```

Делегаты

Последовательность действий для создания и использования делегата:

1. Объявление делегата
2. Создание переменной делегата
3. Присваивание переменной адреса метода
4. Возов методов делегата

```
//Обьявление делегата
delegate void Message();
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    //Создание переменной делегата
    Message mes;
    //Присваивание переменной адреса метода
    mes = Hello;
    //Возов методов делегата
    mes();
}
//Метод
Ссылок: 1
void Hello() => Console.WriteLine("Hello world");
```

Делегаты

Использование делегата с возвращаемыми данными:

```
delegate int Operation(int x, int y);
Ссылок: 1
int Add(int x, int y) => x + y;
Ссылок: 1
int Multiply(int x, int y) => x * y;
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    //делегат указывает на метод Add
    Operation operation = Add;
    //вызов Add(4, 5)
    int result = operation(4, 5);
    //Вывод результата "9"
    Console.WriteLine(result);

    //теперь делегат указывает на метод Multiply
    operation = Multiply;
    //вызов Multiply(4, 5)
    result = operation(4, 5);
    //Вывод результата "20"
    Console.WriteLine(result);
}
```

Делегаты

Добавление нескольких методов в делегат:

```
delegate void Message();
Ссылок: 1
void Hello() => Console.WriteLine("Hello");
Ссылок: 1
void HowAreYou() => Console.WriteLine("How are you?");
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    Message message = Hello;
    //теперь message указывает на два метода
    message += HowAreYou;
    //Выведется сначала "Hello", затем "How are you?"
    message();
}
```

Делегаты

Удаление методов из делегата:

```
delegate void Message();
Ссылок: 1
void Hello() => Console.WriteLine("Hello");
Ссылок: 2
void HowAreYou() => Console.WriteLine("How are you?");
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    Message message = Hello;
    message += HowAreYou;
    //вызываются все методы из message
    message();
    //удаляем метод HowAreYou
    message -= HowAreYou;
    //вызывается метод Hello
    if (message != null) message();
}
```

Делегаты

Объединение делегатов:

```
delegate void Message();
Ссылок: 1
void Hello() => Console.WriteLine("Hello");
Ссылок: 1
void HowAreYou() => Console.WriteLine("How are you?");
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    Message mes1 = Hello;
    Message mes2 = HowAreYou;
    //объединяем делегаты
    Message mes3 = mes1 + mes2;
    //вызываются все методы из mes1 и mes2
    mes3();
}
```

Делегаты

Вызов делегата через метод Invoke():

Если делегат принимает параметры они передаются внутрь метода Invoke()

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

Ссылок: 1
void Hello() => Console.WriteLine("Hello");
Ссылок: 1
int Add(int x, int y) => x + y;

Ссылок: 0
public MainWindow()
{
    InitializeComponent();

    Message mes = Hello;
    mes.Invoke();

    Operation op = Add;
    int n = op.Invoke(3, 4);
    Console.WriteLine(n);
}
```

Делегаты

Вызов делегата через метод Invoke() позволяет избежать ошибки в том случае когда делегат равен null

```
delegate void Message();
Ссылок: 0
void Hello() => Console.WriteLine("Hello");

Ссылок: 0
public MainWindow()
{
    InitializeComponent();

    //Пустой делегат
    Message mes = null;
    //Вызовет ошибку
    mes();
    //Безопасный вызов
    mes?.Invoke();
}
```

Делегаты

Если делегат возвращает некоторое значение, то возвращается значение последнего метода из списка вызова:

```
delegate int Operation(int x, int y);
Ссылок: 1
int Add(int x, int y) => x + y;
Ссылок: 1
int Subtract(int x, int y) => x - y;
Ссылок: 1
int Multiply(int x, int y) => x * y;

Ссылок: 0
public MainWindow()
{
    InitializeComponent();

    Operation op = Subtract;
    op += Multiply;
    op += Add;
    //Будет выведено 9
    Console.WriteLine(op(7, 2));
}
```

Делегаты

Делегаты могут быть обобщенными:

Здесь делегат **Operation** типизируется двумя параметрами типов. Параметр **T** представляет тип возвращаемого значения. А параметр **K** представляет тип передаваемого в делегат параметра.

```
//обобщенный делегат
delegate T Operation<T, K>(K val);
Ссылок: 1
int Square(int n) => n * n;
Ссылок: 1
double Tripple(double n) => n + n + n;

Ссылок: 0
public MainWindow()
{
    InitializeComponent();

    Operation<int, int> squareOperation = Square;
    double result1 = squareOperation(5);
    Console.WriteLine(result1);

    Operation<double, double> trippleOperation = Tripple;
    double result2 = trippleOperation(5.26);
    Console.WriteLine(result2);
}
```

Делегаты

Также делегаты могут быть параметрами методов. Благодаря этому один метод в качестве параметров может получать действия - другие методы:

```
delegate int Operation(int x, int y);
Ссылок: 1
int Add(int x, int y) => x + y;
Ссылок: 1
int Subtract(int x, int y) => x - y;
Ссылок: 1
int Multiply(int x, int y) => x * y;
Ссылок: 3
void DoOperation(int a, int b, Operation operation)
{
    Console.WriteLine(operation(a, b));
}
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    DoOperation(5, 4, Add);
    DoOperation(5, 4, Subtract);
    DoOperation(5, 4, Multiply);
}
```

Делегаты

Наиболее сильная сторона делегатов состоит в том, что они позволяют **делегировать** выполнение некоторому коду извне.
На момент написания программы можно не знать, что за код будет выполняться. Но можно просто вызвать делегат. А какой метод будет непосредственно выполняться при вызове делегата, будет решаться потом.

Делегаты

```
//Класс банковского счета
public delegate void AccountHandler(string message);
Ссылок: 3
public class Account
{
    int sum;
    // Создаем переменную делегата
    AccountHandler taken;
    Ссылок: 1
    public Account(int sum) => this.sum = sum;

    // Регистрируем делегат
    Ссылок: 1
    public void RegisterHandler(AccountHandler del)
    {
        taken = del;
    }
    Ссылок: 0
    public void Add(int sum) => this.sum += sum;
    Ссылок: 2
    public void Take(int sum)
    {
        if (this.sum >= sum)
        {
            this.sum -= sum;
            // вызываем делегат, передавая ему сообщение
            taken?.Invoke($"Со счета списано {sum} у.е.");
        }
        else
        {
            taken?.Invoke($"Недостаточно средств. Баланс: {this.sum} у.е.");
        }
    }
}
```

```
//Метод для делегата
Ссылок: 1
void PrintSimpleMessage(string message) => Console.WriteLine(message);
Ссылок: 0
public MainWindow()
{
    InitializeComponent();

    // создаем класс Account
    Account account = new Account(200);

    // Добавляем в делегат ссылку на метод PrintSimpleMessage
    account.RegisterHandler(PrintSimpleMessage);

    // Два раза подряд пытаемся снять деньги
    account.Take(100);
    account.Take(150);
}
```

События

Событие - это специальный тип делегата, предназначенный для реализации событийной модели. События позволяют объектам оповещать других о том, что произошло определенное действие, например, нажатие кнопки, завершение процесса или изменение состояния.

События

Событие объявляется с помощью ключевого слова **event:**

```
// Определяем делегат
public delegate void Notify();

Ссылок: 1
public class Process
{
    // Объявляем событие
    public event Notify ProcessCompleted;

    Ссылок: 0
    public void StartProcess()
    {
        // Генерация события
        ProcessCompleted?.Invoke();
    }
}
```

События

Далее требуется подписать данное событие на выполнение некоторого метода:

```
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    Process process = new Process();

    // Подписка на событие
    process.ProcessCompleted += OnProcessCompleted;

    process.StartProcess();
}
//Метод для подписки
Ссылок: 1
public static void OnProcessCompleted()
{
    Console.WriteLine("Process completed!");
}
```

События

Вариант без использования событий:

```
Ссылок: 3
class Account
{
    // сумма на счете
    Ссылок: 7
    public int Sum { get; private set; }
    // в конструкторе устанавливаем начальную сумму на счете
    Ссылок: 1
    public Account(int sum) => Sum = sum;
    // добавление средств на счет
    Ссылок: 1
    public void Put(int sum) => Sum += sum;
    // списание средств со счета
    Ссылок: 2
    public void Take(int sum)
    {
        if (Sum >= sum)
        {
            Sum -= sum;
        }
    }
}
Ссылок: 0
```

```
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    Account account = new Account(100);
    account.Put(20);    // добавляем на счет 20
    Console.WriteLine($"Сумма на счете: {account.Sum}");
    account.Take(70);   // пытаемся снять со счета 70
    Console.WriteLine($"Сумма на счете: {account.Sum}");
    account.Take(180);  // пытаемся снять со счета 180
    Console.WriteLine($"Сумма на счете: {account.Sum}");
}
```

События

Вариант с использованием событий:

```
Ссылок: 3
class Account
{
    public delegate void AccountHandler(string message);

    //Определение события
    public event AccountHandler Notify;
    Ссылок: 1
    public Account(int sum) => Sum = sum;
    Ссылок: 8
    public int Sum { get; private set; }
    Ссылок: 1
    public void Put(int sum)
    {
        Sum += sum;
        //Вызов события
        Notify?.Invoke($"На счет поступило: {sum}");
    }
    Ссылок: 2
    public void Take(int sum)
    {
        if (Sum >= sum)
        {
            Sum -= sum;
            //Вызов события
            Notify?.Invoke($"Со счета снято: {sum}");
        }
        else
        {
            //Вызов события
            Notify?.Invoke($"Недостаточно денег на счете. Текущий баланс: {Sum}"); ;
        }
    }
}
```

```
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    Account account = new Account(100);
    account.Notify += DisplayMessage;   // Добавляем обработчик для события Notify
    account.Put(20);    // добавляем на счет 20
    Console.WriteLine($"Сумма на счете: {account.Sum}");
    account.Take(70);   // пытаемся снять со счета 70
    Console.WriteLine($"Сумма на счете: {account.Sum}");
    account.Take(180);  // пытаемся снять со счета 180
    Console.WriteLine($"Сумма на счете: {account.Sum}");
}
Ссылок: 1
void DisplayMessage(string message) => Console.WriteLine(message);
```

События и делегаты

Делегаты представляют собой типы, которые хранят ссылки на методы. Они полезны в ситуациях, когда требуется передать метод в качестве аргумента или сохранить метод для последующего вызова:

- **Передача поведения в методах**: Когда нужно передать метод в качестве параметра, чтобы другой метод мог вызвать его.

События и делегаты

События создаются на основе делегатов, но предназначены для других целей. События обеспечивают механизм подписки и уведомления, часто используются в контексте взаимодействия между объектами:

- **Подписка на изменения состояния**: Если нужно уведомить другие части программы об изменении состояния или завершении какого-либо действия.
- **Событийная модель**: Используется в пользовательских интерфейсах (UI). Например, нажатие кнопки, изменение текста, перемещение мыши и т. д.

События и делегаты

Используйте **делегаты**, когда вам нужно передавать или сохранять методы для вызова в будущем.

Используйте **события**, когда нужно уведомлять другие объекты о произошедших изменениях или действиях.

Спасибо за внимание