

Разработка приложений на платформе .NET

Лекция 7

Делегаты
События

Сегодня

- **Делегаты**
 - Одиночные делегаты
 - Цепочка делегатов
 - Обобщенные делегаты
 - Анонимные методы
 - Лямбда выражения
 - Замыкания
- **Ковариантность и контрвариантность**
 - Делегатов
 - Интерфейсов
- **События**

Сегодня

- **Делегаты**
 - **Одиночные делегаты**
 - Цепочка делегатов
 - Обобщенные делегаты
 - Анонимные методы
 - Лямбда выражения
 - Замыкания
- **Ковариантность и контрвариантность**
 - Делегатов
 - Интерфейсов
- **События**

Ссылки на функции

Обратный вызов



При работе с Win API

C, C++

- Вызов глобальных функций
- Вызов статических функций
- Вызов функций объекта. Необходима ссылка на объект.

Делегат

- Делегат – **объект**, безопасный в отношении типов, указывающий на метод
- Содержит:
 - ссылку на объект
 - ссылку на метод
 - жестко определяет типы и количество параметров метода
 - жестко определяет тип возвращаемого значения
- Может указывать на статический метод или метод экземпляра
 - Если ссылка на объект равна null, это означает, что вызываемый метод статический
- Обеспечивает обратный вызов
- Вызов делегата синтаксически такой же как вызов обычной функции

Работа с делегатами

1. **Объявить (описать) делегат**
 - Описать тип делегата, используя специальный синтаксис (описать класс)
2. **Создать экземпляр делегата**
 - Объявление обычной переменной типа
 - Вызов конструктора
 - Передача конструктору ссылки на метод экземпляра или статический метод
3. **Вызвать делегат**

Объявление типа делегата

Синтаксис

```
[attributes] [modifiers] delegate  
return-type type-name(args-list);
```

- похож на определения абстрактного метода, но это определение типа
- жестко задает сигнатуру вызываемого метода

Примеры

```
public delegate double Function2d(double x, double y);  
public delegate Complex ComplexFunction(Complex z);  
public delegate void EventHandler(object o, EventArgs e);
```

За кулисами

- Пример:

```
public delegate double ProcessResults(double x, double y);
```

- За кулисами (создается класс наследник от MulticastDelegate)

```
public sealed class ProcessResults : System.MulticastDelegate
{
    public ProcessResults (object target, uint funcAdress); // для понимания
    public object Target { get; }
    public MethodInfo Method { get; }
    public double Invoke (double x, double y); // Сигнатура совпадает
    .....
}
```

- Вызов делегата синтаксически такой же как вызов обычной функции, но реально будет вызываться метод Invoke
- Создаются еще методы BeginInvoke() и EndInvoke() для асинхронного вызова метода
- Самостоятельно нельзя создать класс наследник от MulticastDelegate или от Delegate. Только через синтаксис delegate

Создание (экземпляра) делегата

- При создании требуется связать с вызываемым методом (передать в конструктор)
- Для метода экземпляра
 - `ProcessResults del = new ProcessResults(objectName.Function);`
- Для статического метода
 - `ProcessResults del = new ProcessResults(typeName.Function);`
- Сокращенная запись
 - `ProcessResults del = objectName.Function;`
 - `ProcessResults del = typeName.Function;`
- Сигнатура метода и делегата должна совпадать

Вызов делегата

- Как и вызов обычной функции, где в качестве вызываемой функции указывается экземпляр делегата
 - `double d = del(x, y);`
 - `double d = del(4+12, 37);`
- Вызывать метод `Invoke` не рекомендуется, но не возбраняется. В некоторых случаях это необходимо.
 - `double d = del.Invoke(4+12, 37);`

Демонстрации

Одиночный делегат

Работа с делегатами

1. **Объявить (описать) делегат**
 - Описать тип делегата, используя специальный синтаксис (описать класс)
2. **Создать экземпляр делегата**
 - Объявление обычной переменной типа
 - Вызов конструктора
 - Передача конструктору ссылки на метод экземпляра или статический метод
3. **Вызвать делегат**

Делегаты как параметры функции

- Делегаты можно использовать для передачи функций как параметров

```
public delegate double RealFunc (double x);

public double Integrate (double a, double b, int n,
                        RealFunc f)
{
    double dx = (b - a) / n, res = 0.0;
    for(int j = 0; j < n; j++)
        res += f (a + j * dx) * dx;
    return res;
} // end of Integrate()

double s = Integrate(0, 1, 1000, Math.Sin);
```

Сегодня

- **Делегаты**
 - Одиночные делегаты
 - Цепочка делегатов
 - Обобщенные делегаты
 - Анонимные методы
 - Лямбда выражения
 - Замыкания
- Ковариантность и контрвариантность
 - Делегатов
 - Интерфейсов
- События

Цепочка делегатов

- Позволяет, вызвав один делегат, последовательно вызвать несколько методов

(с одинаковой сигнатурой)

```
public abstract class MulticastDelegate : Delegate
{
    public sealed override Delegate[] GetInvocationList(); // возвращает список
        делегатов
    private IntPtr _invocationCount;
    private object _invocationList;
    ...
}
```

- MulticastDelegate может хранить ссылку не на одну функцию, а на несколько
- При вызове делегата функции могут выполняться в произвольном порядке
- Если функции возвращают значение, то только последнее значение можно будет использовать
- В случае возникновения необработанной исключительной ситуации, прерывается

Класс Delegate

```
public abstract class Delegate : ICloneable
{
    public static Delegate Combine (params Delegate[] delegates);
    public static Delegate Combine (Delegate delegate1, Delegate delegate2);
    public static Delegate Remove (Delegate source, Delegate value);
    public static Delegate RemoveAll (Delegate source, Delegate value);
    public virtual Delegate[] GetInvocationList();
    ...
}
```

- Классы Delegate и MulticastDelegate неизменяемые, поэтому все методы комбинации делегатов статические и возвращают новый экземпляр делегата
- Combine() – объединяет делегаты или цепочки делегатов в новую цепочку делегатов
- Remove() – удаляет указанный делегат из цепочки (первый встретившийся с конца)
- RemoveAll() – удаляет все копии указанного делегата из цепочки
- GetInvocationList() – возвращает цепочку делегатов в виде массива одиночных делегатов

Сокращение записи создания цепочки делегатов

- Использование операция сложения и вычитания +, -.
- Использование += и -=
- Примеры:
 - `ProcessResult delegate1 = new, delegate2 = new`
 - `ProcessResult chain = delegate1 + delegate2;`
 - `chain += delegate3;`
 - `ProcessResult chain =
(ProcessResult)Delegate.Combine(delegate1, delegate2);`

Цепочка делегатов

- Вызов цепочки делегатов такой же (последовательно вызываются все методы в этой цепочке)

```
double d = chain(5, 10);
```

- Итерация по цепочке делегатов

```
Delegate[] delegates = chain.GetInvocationList();  
ProcessResult pr = (ProcessResult) delegates[0];  
double result = pr(5, 6);
```

```
foreach (ProcessResult del in chain.GetInvocationList())  
{  
    Console.WriteLine(del(x,y));  
}
```

Демонстрации

Цепочка делегатов

Сегодня

- **Делегаты**

- Одиночные делегаты
- Цепочка делегатов
- **Обобщенные делегаты**
- Анонимные методы
- Лямбда выражения
- Замыкания

- **Ковариантность и контрвариантность**

- Делегатов
- Интерфейсов

- **События**

Обобщенный делегат

- Аналогично обобщенным методам
- Значение типа параметра указывается при создании экземпляра делегата (и только там)
- Вызов делегата при этом ничем не отличается от вызова необобщенного делегата
- Примеры:

- Описание:

```
delegate T UnarOperation<T>(T t);  
delegate T SumValueDelegate<T>(T t1, T t2) where T : struct;  
delegate List<T> Delegatishе<T, K, N, X, Z>(X x, Z[] z, IEnumerable<N> ns, K k);
```

- Создание:

```
UnarOperation<Employee> uo = emp.SetEmployee;  
SumValueDelegate<int> sd = new SumValueDelegate<int>(vector.Sum);  
Delegatishе<int, int, Employee, long, string> d = new Delegatishе<int, int, Employee, long,  
string>(variable.Method);
```

- Вызов:

```
Employee e = uo(new Employee());  
int i = sd(3, 4);  
List<int> result = d(23, myString, EmployeeList, 5);
```

Стандартные делегаты

- Уже описанные типы делегатов
- Делегаты принимающие параметры и ничего не возвращающие
 - **Action**
 - Action<T1>
 - Action<T1, T2> `public delegate void Action<T1, T2>(T1 arg1, T2 arg2);`
 - ...
 - Action <T1, T2, ..., T16 >
- Делегаты, возвращающие данные (тип-параметр TResult), и, принимающие параметры:
 - **Func**<TResult>
 - Func<T, TResult>
 - Func<(T1, T2, TResult> `public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);`
 - ...
 - Func<(T1, T2, ..., T16, TResult>
- Предикаты (возвращает bool. Выполняется ли условие):
 - **Predicate**<T> `public delegate bool Predicate<T>(T obj);`
- Обработчики событий:
 - **EventHandler** `public delegate void EventHandler(object sender, EventArgs e);`
 - **EventHandler**<TEventArgs>
 - **RoutedEventHandler**

Демонстрации

Обобщенные и стандартные
делегаты

Сегодня

- **Делегаты**

- Одиночные делегаты
- Цепочка делегатов
- Обобщенные делегаты
- **Анонимные методы**
- Лямбда выражения
- Замыкания

- **Ковариантность и контрвариантность**

- Делегатов
- Интерфейсов

- **События**

Анонимные методы

- Обеспечивают более простой и компактный способ определения простых делегатов
- Позволяет создать тело метода делегата в месте создания экземпляра делегата
- Анонимный метод – выражение типа «кусочек кода», которое может быть присвоено переменной-делегату
- Синтаксис:

delegate_var = **delegate**(*arg_list*) { *method body* };

```
Func<int, int, int> del = delegate(int x, int y) { return x + y; };
```

```
Action<string> act = delegate(string s) {
```

```
Console.WriteLine(s); };
```

```
act += delegate(string a) { Console.WriteLine(a); };
```

Демонстрации

Анонимные методы

Сегодня

- **Делегаты**

- Одиночные делегаты
- Цепочка делегатов
- Обобщенные делегаты
- Анонимные методы
- Лямбда выражения
- Замыкания

- **Ковариантность и контрвариантность**

- Делегатов
- Интерфейсов

- **События**

Лямбда выражения

- Краткая запись анонимных делегатов
- Элементы функционального программирования
- Два вида записи:
 - “Лямбда оператор”
 - “Лямбда выражение”
- Преобразуются в анонимный метод

СИНТАКСИС ЛЯМБДА ВЫРАЖЕНИЙ

- **Лямбда оператор** (если тела метода состоит из более одного оператора):
`(in_arg_list) => {method body}`
- Пример:
 - `Func<int, int> del = (int x) => { x++; return -x; }`
 - `Action<string> del = (string s) => { Console.WriteLine(s); }`
- Типы входных параметров обычно не указываются
 - `Func<double, double> del = (x) => { x++; return -x; }`
 - `Func<DateTime> del = () => {DateTime dt = DateTime.Now; return dt; }`
- Количество и типы входных и возвращаемых параметров определяются по делегату. Только если нужен другой (совместимый тип), указывается тип входного параметра.
- Если входной параметр один, то скобки можно опустить:
 - `Func<int, int> del = x => { x++; return x*4; }`
- **Лямбда выражение:**
`(in_arg_list) => return value`
- Нет оператора return
- Пример:
 - `Func<int, int> del = x => x*x;`
 - `Func<long, long, long> del = (x, y) => x+y;`
 - `List<Complex> complexList =;`
`Complex finded = complexList.Find(compl => compl.Re > 5);`

Демонстрации

Лямбда выражения

Сегодня

- **Делегаты**

- Одиночные делегаты
- Цепочка делегатов
- Обобщенные делегаты
- Анонимные методы
- Лямбда выражения
- **Замыкания**

- Ковариантность и контрвариантность

- Делегатов
- Интерфейсов

- События

Замыкания

- Лямбда выражения и анонимные методы могут использовать внутри себя переменные окружения.
 - `int i = 5;`
 - `Func<int, int> del = x => x+i;`
 - Происходит захват внешней переменной
 - Осторожно, переменная может изменяться внутри анонимного метода и снаружи
- Анонимная рекурсия
 - `Func<int, int> fact;`
 - `fact = x => x>1 ? x*fact(x-1) : 1;`
 - `Console.WriteLine(fact(5));`

Демонстрации

Замыкания

Сегодня

● Делегаты

- Одиночные делегаты
- Цепочка делегатов
- Обобщенные делегаты
- Анонимные методы
- Лямбда выражения
- Замыкания

● Ковариантность и контрвариантность

- Делегатов
- Интерфейсов

● События

ПОНЯТИЯ

Не .NET. Общее понимание.

```
class Person{...}  
class Student : Person {...}
```

Ковариантность

```
class People  
{  
    Person GetPersons();  
}  
class CoursePeople : People  
{  
    Student GetPersons();  
}
```

Соблюдение контракта базового класса

```
class People  
{  
    void SetPersons(Person p);  
}  
class CoursePeople : People  
{  
    void SetPersons(Student p);  
}
```

*Нарушение контракта базового класса
Нарушение принципа ООП*

Возможна только в выходных параметрах

Контрвариантность

```
class ClassPeople  
{  
    Student GetStudents();  
}  
class TodayPeople : ClassPeople  
{  
    Person GetStudents();  
}
```

*Нарушение контракта базового класса
Нарушение принципа ООП*

```
class ClassPeople  
{  
    void SetStudents(Student p);  
}  
class TodayPeople : ClassPeople  
{  
    void SetStudents (Person p);  
}
```

Соблюдение контракта базового класса

Возможна только во входных параметрах

Ковариантность делегатов

- Ковариантность – приведение частного к общему
 - В терминах ООП: Там где требуется базовый тип можно присвоить экземпляр типа наследника
- **Делегаты ковариантны по возвращаемому типу**
- Ковариантность позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата.

```
class Employee { }  
class Programmer : Employee { }
```

```
delegate Employee GetEmployeeDelegate();
```

```
class Person  
{  
    public static Employee GetEmployee() {...}  
    public static Programmer GetProgrammer() {...}  
}
```

```
GetEmployeeDelegate del = Person.GetEmployee;  
Employee emp = del();
```

```
del = Person.GetProgrammer;  
Programmer prgmr = (Programmer)del();
```

Контрвариантность делегатов

- Контрвариантность – приведение общего к частному
 - В терминах ООП: Там где требуется тип можно присвоить экземпляр базового типа
- **Делегаты контрвариантны по типам входных параметров**
- Контрвариантность позволяет присвоить делегату метод, типом параметра которого служит класс, являющийся базовым для класса, указываемого в объявлении делегата.

```
class Employee { }  
class Programmer : Employee { }
```

```
delegate void SetProgrammerDelegate(Programmer emp);
```

```
class Person  
{  
    public static void SetEmployee(Employee emp) { }  
    public static void SetProgrammer(Programmer prog) { }  
}
```

```
SetProgrammerDelegate del = Person.SetEmployee;  
del(new Programmer());  
del = Person.SetProgrammer;  
del(new Programmer());
```

КОНТРВАРИАНТНОСТЬ ИНТЕРФЕЙСОВ

- **out** - обозначение ковариантного типа-параметра.
Тип, обозначенный как **out**, может присутствовать только как возвращаемый параметр

```
interface IEnumerable<out T>
{
    IEnumerator<T> GetEnumerator();
}
```

- **in**- обозначения контрвариантного типа- параметра.
Тип, обозначенный как **in**, может присутствовать только как входной параметр

```
interface IComparable<in T>
{
    int CompareTo(T other );
}
```

- Параметры **in** и **out** могут быть в одном интерфейсе

```
interface IMyInterface<in T, out K>
{
    int SetT(T t);
    K GetK();
    K Convert(T t);
}
```

- Преобразования

```
class Person{...};
class Student : Person {...}
```

```
IEnumerable<Student> students = new List<Student>();
IEnumerable<Person> persons = students;
```

```
IComparable<Student> students = ...
IComparable<Person> persons = students;
```

Демонстрации

Ковариантность и контрвариантность

Сегодня

● Делегаты

- Одиночные делегаты
- Цепочка делегатов
- Обобщенные делегаты
- Анонимные методы
- Лямбда выражения
- Замыкания

● Ковариантность и контрвариантность

- Делегатов
- Интерфейсов

● События

Пример

```
class Car
{
    public delegate void PetrolIsOver(string message);
    public PetrolIsOver PetrolIsOverCallBack;
    const float lPer100 = 10;
    private float petrol = 50;

    public void Drive(int km)
    {
        for (int i = km; i > 0; i--)
        {
            petrol -= 1 * lPer100 / 100;
            if (petrol <= 0) { PetrolIsOverCallBack("Приехали"); break; }
            if (petrol < 5) PetrolIsOverCallBack("Бензин заканчивается");
        }
    }
}
```

```
Car opel = new Car();
opel.PetrolIsOverCallBack = message => Console.WriteLine(message);
opel.Drive(600);
```

```
opel.PetrolIsOverCallBack += message => Console.WriteLine("Можно добавить подписку: " + message);
opel.Drive(10);
```

```
opel.PetrolIsOverCallBack = message => Console.WriteLine("А Можно и затереть подписку");
opel.Drive(10);
```

```
opel.PetrolIsOverCallBack += message => Console.WriteLine(message);
opel.PetrolIsOverCallBack("Более того можно и самим вызвать callback, т.е. симулировать событие");
```

- Возможность снаружи управлять подписками
- Возможность снаружи вызвать делегат
- Нарушение инкапсуляции

Демонстрации

Недостаток public переменной-экземпляра
делегата

События

- Событие – некоторая программная конструкция, которая упрощает создание делегатов и методов работы с ним, служащая для оповещения заинтересованных подписчиков о возникновении некоторой интересной ситуации (события)
- На событие можно подписаться и от него можно получать оповещения
- Оповещения приходят в виде вызовов зарегистрированных методов

Объявление события

```
[attributes] [modifiers]  
event delegate-type event-name  
[ { add { accessor-body }  
remove { accessor-body } } ];
```

- Примеры создания:

```
public event EventHandler Selected;  
public event PaintEventHandler Paint;  
public event MouseEventHandler MouseUp;  
public event MyDelegate MyEvent  
{  
    add { MyEvent += value; }  
    remove { MyEvent -= value; }  
}
```

Изнутри и снаружи

Изнутри

- Событие – свойство-делегат, с которым можно обращаться точно так же

- Вызов делегата – инициация события

```
public delegate void MyDelegate(string message);  
public event MyDelegate MyEvent;
```

...

```
MyDelegate e = MyEvent; // для потокобезопасности  
if (e != null) e("Параметры делегата"); // обязательна проверка на null
```

Снаружи

- С событием можно общаться только при помощи двух аксессоров

- += подписаться на событие

- -= отписаться от события

```
myVar.MyEvent += new MyDelegate(MyHandler);  
myVar.MyEvent += message => Console.WriteLine(message);  
myVar.MyEvent -= MyHandler;
```

- Когда происходит событие, вызывается ваш метод

Демонстрации

События

Частное событие

Соглашения о событиях

- Тип делегата-события:
 - `delegate void EventHandler(object sender, EventArgs e);`
 - `delegate void EventHandler<TEventArgs>(object sender, TEventArgs e) where TEventArgs : EventArgs;`
- `sender` – объект, породивший событие
 - Один обработчик на несколько событий
- `EventArgs` – дополнительная передаваемая информация о событии
 - Для передачи своей информации о событии необходима создать класс наследник от `EventArgs` и расширить его для передачи дополнительной информации о событии

Демонстрации

События

Стандартные делегаты

Сегодня рассмотрим

- **Делегаты**
 - Одиночные делегаты
 - Цепочка делегатов
 - Обобщенные делегаты
 - Анонимные методы
 - Лямбда выражения
 - Замыкания
- **Ковариантность и контрвариантность**
 - Делегатов
 - Интерфейсов
- **События**