# Делегаты и события

# Делегаты

- это объект, предназначенный для хранения ссылок на методы(указатель на функцию C++)
- функции обратного вызова + без. типов

1) используются для поддержки событий
2) как самостоятельная конструкция языка

System.Delegate

```
[атрибуты]  [спецификаторы]
        delegate    тип имя_делегата([параметры] )
```

System.MulticastDelegate

new, public, protected, internal и private.

# Свойства делегата

- Тип данных (ссылочный)
- Наследовать от делегата нельзя
- Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса (в любом месте, где может быть определен класс)

```
public delegate void D(int i);

class GGG
{
    public delegate void GGGD(int i);
}
```

- может вызывать только такие методы, у которых тип возвращаемого значения и список параметров  совпадают
- Может быть статический метод класса
- Имеет тот же синтаксис, что и вызов метода
- Если делегат хранит ссылки на несколько методов, они вызываются последовательно в том порядке, в котором были добавлены в делегат (цепочки (chaining))

- делегат можно вызывать как обычный метод
- делегаты могут быть параметрами методов

```
public delegate void D(int i);
        class Class1
        {
              public void ReadDelegat (D _del) { _del(1); }
       }
```

# Назначение делегатов

- 1)возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы;
- 2)обеспечения связи между объектами по типу «источник — наблюдатель»;
- 3)создания универсальных методов, в которые можно передавать другие методы;
- 4)поддержки механизма обратных вызовов.

# Использование делегатов

```
public delegate void D(int i);
        class Class1
        {
            private static void HelloI(int i) { }
            static void Main()
            {
                D del;
                del = HelloI;
                del.Invoke(4);
            }
        }
```

Объявляем делегат

Создаем переменную делегата

Присваиваем этой переменной адрес метода

Вызываем метод

Или так

```
del = new D(HelloI);
del(4);
```

Чтобы использовать делегат, нам надо создать его объект с помощью конструктора, в который мы передаем адрес метода, вызываемого делегатом

```
delegate void Message(); // 1. Объявляем делегат
```

```
void Hello() => Console.WriteLine("Hello METANIT.COM");
```

```
Message mes;              // 2. Создаем переменную делегата
mes = Hello;              // 3. Присваиваем этой переменной адрес метода
mes();                    // 4. Вызываем метод
```

# Вызываемые делегатом методы должны соответствовать его сигнатуре:

```
1  delegate void SomeDel(int a, double b);
```

Этому делегату соответствует, например, следующий метод:

```
1  void SomeMethod1(int g, double n) { }
```

А следующие методы НЕ соответствуют:

```
1  double SomeMethod2(int g, double n) { return g + n; }
2  void SomeMethod3(double n, int g) { }
3  void SomeMethod4(ref int g, double n) { }
4  void SomeMethod5(out int g, double n) { g = 6; }
```

Следует учитывать, что если делегат пуст, то есть в его списке вызова нет ссылок ни на один из методов (то есть делегат равен Null), то при вызове такого делегата мы получим исключение

```
Message? mes;
//mes();          // ! Ошибка: делегат равен null

Operation? op = Add;
op -= Add;        // делегат op пуст
int n = op(3, 4);         // !Ошибка: делегат равен null
```

при вызове делегата всегда лучше проверять, не равен ли он null. Либо можно использовать метод Invoke и оператор условного null

```
Message? mes = null;
mes?.Invoke();         // ошибки нет, делегат просто не вызывается

Operation? op = Add;
op -= Add;           // делегат op пуст
int? n = op?.Invoke(3, 4);   // ошибки нет, делегат просто не вызывается, а n = null
```

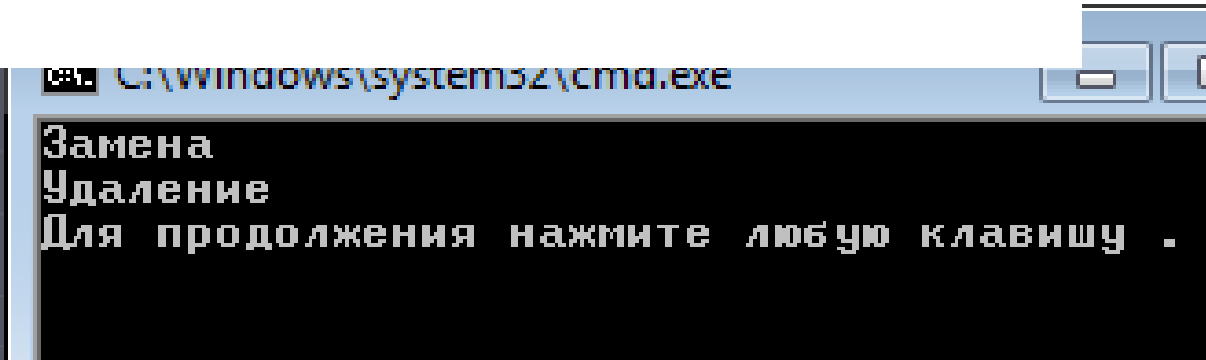
# Передача делегатов в методы

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

// объявление делегата
class Class1
{
    public static void Table(Fun F, double x, double b)
    {
        Console.WriteLine(" X           Y ");
        while (x <= b)
        {
            Console.WriteLine("| {0,8:0.000} | {1,8:0.000} |", x, F(x));
            x += 1;
        }
        Console.WriteLine(" ");
    }


    static void Main()
    {
        Table(new Fun(Math.Sin), -2, 2);
    }

}
```

создает экземпляр делегата

# Пример

```
delegate string strMod(string stx);

class DelegateTest
{
    static string replaceSpaces(string a) { Console.WriteLine("Замена"); return a; }
    static string removeSpaces(string a) { Console.WriteLine("Удаление"); return a; }
    static string reverse(string a) { Console.WriteLine("Реверс"); return a; }

    public static void Main()
    {
      public static void Main()
      {
          strMod strOp = replaceSpaces;
          string str; str = strOp("ЭТО простой тест.");
          strOp = DelegateTest.removeSpaces;
          str = strOp("Это простой тест.");
      }
}
```



# Операции над делегатами

- можно *сравнивать на равенство и неравенство* (не содержат ссылок или если ссылки на одни и те же методы в одном и том же порядке)
- *выполнять операции* простого и составного *присваивания* (один тип д.и.)

```
del += HelloI; // добавляем делегат
del -= HelloI; // удаляем делегат
```

- является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.

# Объединение делегатов

при вызове делегата mes3 все эти методы одновременно будут вызваны

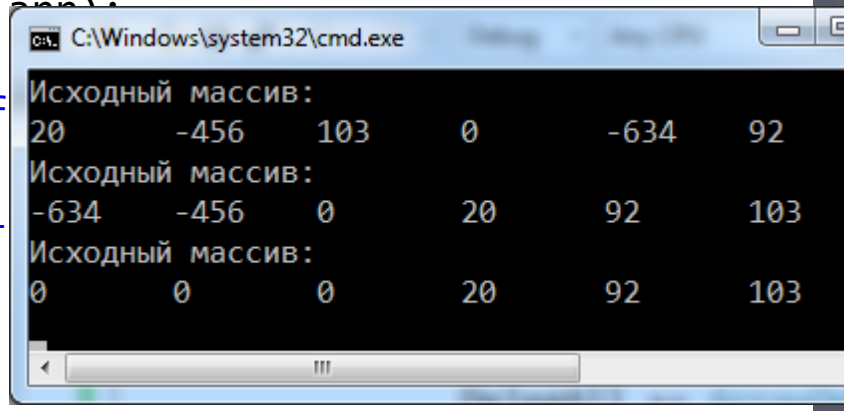
Делегаты можно объединять в другие делегаты. Например:

```
1  Message mes1 = Hello;
2  Message mes2 = HowAreYou;
3  Message mes3 = mes1 + mes2; // объединяем делегаты
4  mes3(); // вызываются все методы из mes1 и mes2
5
6  void Hello() => Console.WriteLine("Hello");
7  void HowAreYou() => Console.WriteLine("How are you?");
8
9  delegate void Message();
```

# Групповая адресация

Создание списка, или цепочки вызовов, для методов, которые вызываются автоматически при обращении к делегату

```
delegate void OperationWithArray(ref int[] arr);
        public class ArrayOperation
        {  public static void WriteArray(ref
            // Сортировка массива
            public static void IncSort(ref i
            // Заменяем отрицательные числа
            public static void NegatArr(ref
        }
        class Program
        { static void Main()
            {  int[] someArr = new int[] { 20, -456, 103, 0, -634, 92 };
                // Структуируем делегаты
                OperationWithArray DelegAll;      // Групповая адресация
                DelegAll = ArrayOperation.WriteArray;
                DelegAll += ArrayOperation.IncSort;
                DelegAll += ArrayOperation.WriteArray;
                DelegAll += ArrayOperation.NegatArr;
                DelegAll += ArrayOperation.WriteArray;
                // Выполняем делегат
                DelegAll(ref someArr);
                Console.ReadLine();
```



# Анонимные функции

- представляет собой безымянный кодовый блок, передаваемый конструктору делегата
  - Анонимные методы
  - Лямбда - выражения

```
delegate int Summator(int b);

          static int result = 0;
           Summator someDelegat = delegate (int number)
           {
               for (int i = 0; i <= number; i++)
                   result += i; //захват переменной
               return result;
          };
```

метод, встроенный в код

- Параметры должны соответствовать параметрам делегата
- может не содержать никаких параметров
- метод имеет доступ ко всем переменным, определенным во внешнем коде

# Лямбда-выражения

- упрощенная запись анонимных методов

*параметр => выражение*

*(список_параметров) => выражение*

```
(x, y) => x + y;
```

```
i => i * i;
```

не надо указывать тип параметров, не надо использовать оператор return

# Лямбда-выражения

```
//объявление
            delegate int  FindMax(int s1, int s2);
    private delegate bool isNegate(int a);
 static void Main()
     {
 //определение
         FindMax maxint = (s1, s2) => s1 > s2 ? s1 : s2;
         isNegate nega = s3 => s3 < 0;
 //вызов
                 int a = 10;
                 int b = -8;
                 maxint(a, b);
                 nega(b);
```

упрощенная запись анонимных методов

# Блочные лямбда-выражения лямбда -оператор

```
 //объявление
private delegate bool isOdd(int a);

    static void Main()
            {
                //определение
      isOdd testIsOdd = s4 =>
                {
                    if (s4%2 == 0) return true;
                    else return false;
                };
                //вызов
                int a = 10;
            testIsOdd(a); //true
```

- Список параметров может быть пустым

```
public delegate void D();


        class Class1
        {
            static void Main()
            {
                D del = () =>  Console.Write("Hello");
                del();
            }
        }
```

- можно передавать в качестве аргументов методу
- может принимать ссылку на метод

```
public delegate void D();


        class Class1
        {
            private static void SayHello() { Console.Write("Hello");
}
            static void Main()
            {
                D del = () => SayHello();
                del();
            }
        }
```

лямбда-выражения можно передавать в качестве параметров методу

# Обобщенные делегаты

```
delegate T Operation<T, K>(K val);
```

T- тип возвращаемого значения.
K -тип передаваемого в делегат параметра.

```
decimal Square(int n) => n * n;
int Double(int n) => n + n;
```

```
Operation<decimal, int> squareOperation = Square;
decimal result1 = squareOperation(5);
Console.WriteLine(result1);  // 25

Operation<int, int> doubleOperation = Double;
int result2 = doubleOperation(5);
Console.WriteLine(result2);  // 10
```

# Обобщённые делегаты .NET Action<T> и Func<T>, Predicate<T>

Определенные в пространстве имен System

- void Action<in T1, in T2, in T3 ….in T16 >
- TResult  Func<out TResult>
- TResult Func<in T1,……T16, out TResult>

До 16 параметров

Вместо определения собственных типов делегатов рекомендуется по мере возможности использовать обобщенные делегаты

```
static public void Sort<T>
  (IList<T> sortArray,
        Func<T, T, bool> res) { ....}
```

# Action

До 16 параметров

```
public delegate void Action<T>(T obj)
```

представляет некоторое действие, которое ничего не возвращает, то есть в качестве возвращаемого типа имеет тип void:

```
class Test
        {
         static void Operation(int x1, int x2, Action<int, int> op)
            =>      op(x1, x2);

          static void Main(string[] args)
            {
                Action<int, int> op;
                op = (int a, int b) => {int c = a + b; };
                Operation(10, 6, op);

            }
```

предусматривает вызов определенных действий

Два параметра

делегат передается в качестве параметра метода и предусматривает вызов определенных действий в ответ на произошедшие действия

```
public delegate void Action()
public delegate void Action<in T>(T obj)
```

```
DoOperation(10, 6, Add);          // 10 + 6 = 16
DoOperation(10, 6, Multiply);     // 10 * 6 = 60

void DoOperation(int a, int b, Action<int, int> op) => op(a, b);

void Add(int x, int y) => Console.WriteLine($"{x} + {y} = {x + y}");
void Multiply(int x, int y) => Console.WriteLine($"{x} * {y} = {x * y}");
```

# Func

Он возвращает результат действия и может принимать параметры

## Func<in T1, in T2,...in T16, out TResult>()

```
class Test
        {
            static void Main(string[] args)
            {

                Console.WriteLine(Operation(6, x => x * x)); // 36
                Console.WriteLine(Operation(10, x => x / 2)); // 5
            }

            static int Operation(int x1, Func<int, int> retF)
            =>  x1 < 0 ? 0: retF(x1);


        }
```

принимает число int и возвращает int

```
Func<int, int, string> createString = (a, b) => $"{a}{b}";
Console.WriteLine(createString(1, 5));  // 15
Console.WriteLine(createString(3, 5));  // 35
```

# Predicate

принимает один параметр и возвращает значение типа bool.

используется для сравнения, сопоставления некоторого объекта T определенному условию.

```
delegate bool Predicate<in T>(T obj);
```

```
Predicate<int> isPositive = (int x) => x > 0;


Console.WriteLine(isPositive(20));
Console.WriteLine(isPositive(-20));
```

true или false в зависимости от того, больше нуля число или нет.

# События

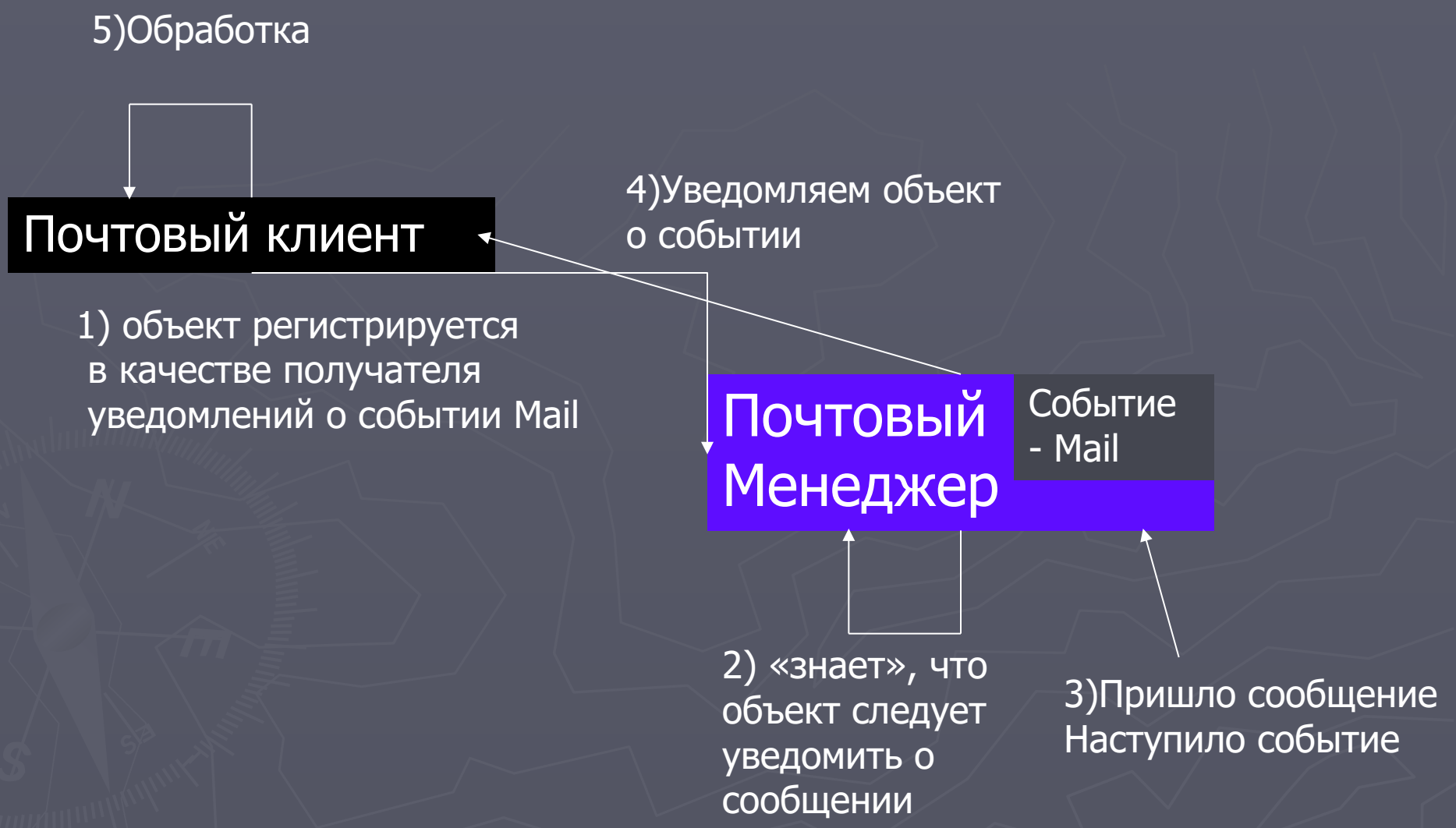
- это элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния

модель «публикация — подписка»или паттерн «наблюдатель»,

| класс Button | - | событие Click |
|---|---|---|

класс, являющийся отправителем (sender) сообщения, публикует события, которые он может инициировать,

а другие классы, являющиеся получателями (receivers) сообщения, подписываются на получение этих событий.


5)Обработка
Почтовый клиент
4)Уведомляем объект
о событии
1) объект регистрируется
в качестве получателя
уведомлений о событии Mail
Почтовый
Менеджер
Событие
- Mail
2) «знает», что
объект следует
уведомить о
сообщении
3)Пришло сообщение
Наступило событие

```
class Account
{
    // сумма на счете
    public int Sum { get; private set; }
    // в конструкторе устанавливаем начальную сумму на счете
    public Account(int sum) => Sum = sum;
    // 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;
        }
    }
}
```

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

```
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}");
```

закрытый статический класс, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата

```
[атрибуты]   [спецификаторы]
          event имяделегата имясобытия
```

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

- События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому *создание события* в классе состоит из следующих частей:
  - описание делегата, задающего сигнатуру обработчиков событий;
  - описание события;
  - описание метода (методов), инициирующих событие.

new, public, protected, internal, private, static, virtual, sealed, override, abstract и extern

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

можем его вызвать в методах класса как метод, используя имя события

```
Notify("Произошло действие");
```

событие равно null в случае, если для его не определен обработчик. Поэтому при вызове события лучше его всегда проверять на null.

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

Или так:

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

```
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}"); ;
        }
    }
}
```

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

# Добавление обработчика события

- событием может быть связан один или несколько обработчиков;

Обработчики событий - это именно то, что выполняется при вызове событий.

- в качестве обработчиков событий применяются методы.
- обработчик событий по списку параметров и возвращаемому типу должен соответствовать делегату, который представляет событие
- Для добавления обработчика события применяется операция +=

```
Notify += обработчик события;
```

```
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}");

void DisplayMessage(string message) => Console.WriteLine(message);
```

```
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
```

# удаление обработчиков
## применяется операция -=

```
Account account = new Account(100);
account.Notify += DisplayMessage;        // добавляем обработчик DisplayMessage
account.Notify += DisplayRedMessage;     // добавляем обработчик DisplayRedMessage
account.Put(20);     // добавляем на счет 20
account.Notify -= DisplayRedMessage;      // удаляем обработчик DisplayRedMessage
account.Put(50);     // добавляем на счет 50

void DisplayMessage(string message) => Console.WriteLine(message);
void DisplayRedMessage(string message)
{
    // Устанавливаем красный цвет символов
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(message);
    // Сбрасываем настройки цвета
    Console.ResetColor();
}
```

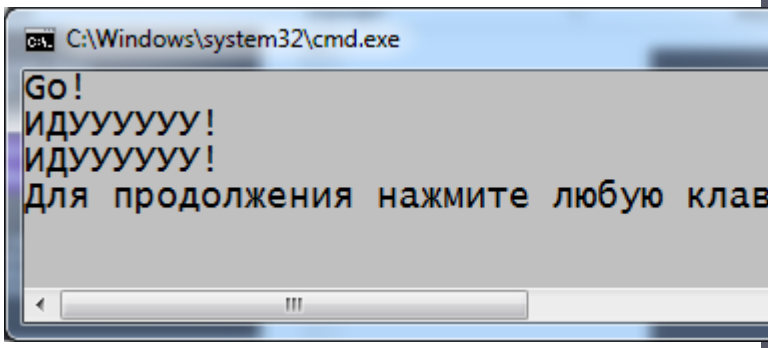
```
public delegate void Doing(object o); // объявление делегата

class A
{
    public event Doing Oppa; // объявление события
}
```

1) Обработка событий выполняется в классах-получателях
2) сигнатура методов-обработчиков событий == типу делегата
3) Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод :+= ( -=)

методы `add_.....`, `remove_`

4) Поддерживается групповая адресация

```
class Mind            // ---- Класс-источник события ---------
       {   public event EventHandler Go; // Описание соб. станд. типа
           public void CommandGo() // Метод, инициирующий событие
           {  Console.WriteLine("Go!");
              if (Go != null)
                  Go(this, null);
           }
       }
class Leg     // ------- Класс-наблюдатель ------------
       {  public void OnGo(object sender, EventArgs e)
           {Console.WriteLine( "ИДУУУУУУ!" ); // Обработчик соб-я
           }
       }
class Class1
       {static void Main()
           {   Mind s = new Mind();
               Leg o1 = new Leg();
               Leg o2 = new Leg();
               Leg o3 = new Leg();
               s.Go += o1.OnGo;          // регистрация обработчика
               s.Go += o2.OnGo;          // регистрация обработчика
               s.CommandGo();
           }
       }
```



```
class Mind              // ---- Класс-источник события ---------
      {   public event EventHandler Go; // Описание соб. станд. типа
          public void CommandGo() // Метод, инициирующий событие
          {  Console.WriteLine("Go!");
              if (Go != null)
                  Go(this, null);
          }
      }
      class Leg     // -------- Класс-наблюдатель -----------
      {  public void OnGo(object sender, EventArgs e)
          {Console.WriteLine( "ИДУУУУУУ!" ); // Обработчик соб-я
          }
      }
      class Class1
      {static void Main()
          {   Mind s = new Mind();
              Leg o1 = new Leg();
              Leg o2 = new Leg();
              Leg o3 = new Leg();
              s.Go += o1.OnGo;          // регистрация обработчика
              s.Go += o2.OnGo;          // регистрация обработчика
              s.CommandGo();
          }
      }
```

Стандартный класс System.EventHandler.

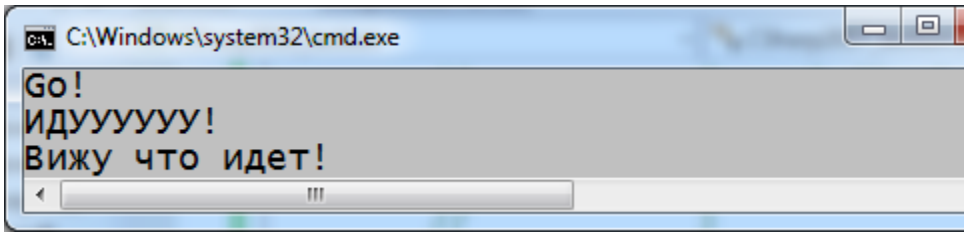
Значение события по умолчанию — null

тип результата void

источник события - имеет тип object
аргументы события и имеет тип EventArgs или производный от него

В библиотеке .NET описано много стандартных делегатов, предназначенных для реализации механизма обработки событий

```
public delegate void SomeDelegat();
class Mind
{   public event SomeDelegat Go;
    public void ComandGo()
    {    Console.WriteLine("Go!");
        if (Go != null)
            Go();
    }
}
class Leg
{   public void OnGo()  {   Console.WriteLine( "ИДУУУУУУ!" );   }
}
class Eye
{       public void OnGo() { Console.WriteLine("Вижу что идет!");}
}
class Class1
{ static void Main()
    {   Mind s = new Mind();
        Leg o1 = new Leg();
        Eye o2 = new Eye();
        s.Go += new SomeDelegat(o1.OnGo);
        s.Go += new SomeDelegat(o2.OnGo);
        s.ComandGo();
    }
}
```



# +делегаты

- Делегат можно вызвать асинхронно (в отдельном потоке), при этом в исходном потоке можно продолжать действия.
- Анонимный делегат (без создания класса-наблюдателя):

```
s.Go += new SomeDelegat(()=>
    { Console.WriteLine("Я слышу что идете!"); });
```

- Делегаты и события обеспечивают взаимодействие взаимосвязанных объектов.
- События включены во многие стандартные классы .NET, используемые для разработки Windows-приложений.