§7.14 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:

Листинг 7.25.

```
1  class Program
2  {
3      delegate void MessageHandler(string message);
4      static void Main()
5      {
6          MessageHandler handler = delegate(string mes)
7          {
8              Console.WriteLine(mes);
9          };
10         handler("hello world!");
11         Console.Read();
12     }
13 }
```

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.

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

Листинг 7.26.

```
1  class Program
2  {
```

```
3  delegate void MessageHandler(string message);
4  static void Main(string[] args)
5  {
6  ShowMessage("hello!", delegate(string mes)
7  {
8  Console.WriteLine(mes);
9  });
10 Console.Read();
11 }
12 static void ShowMessage(string mes, MessageHandler
   handler)
13 {
14 handler(mes);
15 }
16 }
```

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.

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:

```
1  delegate int Operation(int x, int y);
2  static void Main(string[] args)
3  {
4      Operation operation = delegate (int x, int y)
5      {
6          return x * y;
7      };
8      int d = operation(4, 5);
9      Console.WriteLine(d);       // 20
10     Console.Read();
```

```
11  }
```

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

```
1   delegate int Operation(int x, int y);
2   static void Main(string[] args)
3   {
4       int z = 15;
5       Operation operation = delegate (int x, int y)
6       {
7           return x + y + z;
8       };
9       int d = operation(10, 5);
10      Console.WriteLine(d);       // 30
11      Console.Read();
12  }
```

В каких ситуациях используются анонимные методы? Когда нам надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется. В частности, их можно использовать для обработки событий.

§7.15 Лямбды

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.

Лямбда-выражения имеют следующий синтаксис: слева от лямбда-оператора => определяется список параметров, а справа блок выражений, использующий эти параметры:

(список_параметров) => выражение.

Например:

Листинг 7.27.

```
1   class Program
2   {
3   delegate int Operation(int x, int y);
4   static void Main(string[] args)
5   {
```

```
6   Operation operation = (x, y) => x + y;
7   Console.WriteLine(operation(10, 20));       // 30
8   Console.WriteLine(operation(40, 20));       // 60
9   Console.Read();
10  }
11  }
```

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.

Если параметры в лямбда-выражении не требуется, то вместо параметра в выражении используются пустые скобки. Также бывает, что лямбда-выражение не возвращает никакого значения.

§7.16 События

События сигнализируют системе о том, что произошло определенное действие. И если нам надо отследить эти действия, то как раз мы можем применять события.

Например, возьмем следующий класс, который описывает банковский счет:

Листинг 7.28.

```
1   class Account
2   {
```

```
3       public Account(int sum)
4       {
5           Sum = sum;
6       }
7       // сумма на счете
8       public int Sum { get; private set;}
9       // добавление средств на счет
10      public void Put(int sum)
11      {
12          Sum += sum;
13      }
14      // списание средств со счета
15      public void Take(int sum)
16      {
17          if (Sum >= sum)
18          {
19              Sum -= sum;
20          }
21      }
22  }
```

В конструкторе устанавливаем начальную сумму, которая хранится в свойстве Sum. С помощью метода Put мы можем добавить средства на счет, а с помощью метода Take, наоборот, снять деньги со счета. Попробуем использовать класс в программе - создать счет, положить и снять с него деньги:

Листинг 7.29.

```
1   static void Main(string[] args)
2   {
3       Account acc = new Account(100);
4       acc.Put(20);    // добавляем на счет 20
5       Console.WriteLine($"Сумма на счете: {acc.Sum}");
6       acc.Take(70);   // пытаемся снять со счета 70
7       Console.WriteLine($"Сумма на счете: {acc.Sum}");
8       acc.Take(180);  // пытаемся снять со счета 180
9       Console.WriteLine($"Сумма на счете: {acc.Sum}");
10      Console.Read();
11  }
```

Все операции работают как и положено. Но что, если мы хотим уведомлять пользователя о результатах его операций. Мы могли бы, например, для этого изменить метод `Put` следующим образом:

Листинг 7.30.

```
1  public void Put(int sum)
2  {
3      Sum += sum;
4      Console.WriteLine($"На счет поступило: {sum}");
5  }
```

Казалось, теперь мы будем извещены об операции, увидев соответствующее сообщение на консоли. Но тут есть ряд замечаний. На момент определения класса мы можем точно не знать, какое действие мы хотим произвести в методе **Put** в ответ на добавление денег. Это может вывод на консоль, а может быть мы захотим уведомить пользователя по email или sms. Более того мы можем создать отдельную библиотеку классов, которая будет содержать этот класс, и добавлять ее в другие проекты. И уже из этих проектов решать, какое действие должно выполняться. Возможно, мы захотим использовать класс `Account` в графическом приложении и выводить при добавлении на счет в графическом сообщении, а не консоль. Или нашу библиотеку классов будет использовать другой разработчик, у которого свое мнение, что именно делать при добавлении на счет. И все эти вопросы мы можем решить, используя события.

Определение и вызов событий

События объявляются в классе с помощью ключевого слова `event`, после которого указывается тип делегата, который представляет событие:

Листинг 7.31.

```
1  delegate void AccountHandler(string message)
2  event AccountHandler Notify
```

В данном случае вначале определяется делегат `AccountHandler`, который принимает один параметр типа `string`. Затем с помощью ключевого слова `event` определяется событие с именем `Notify`, которое представляет

делегат AccountHandler. Название для события может быть произвольным, но в любом случае оно должно представлять некоторый делегат.

Определив событие, мы можем его вызвать в программе как метод, используя имя события:

```
Notify("Произошло действие");
```

Поскольку событие Notify представляет делегат AccountHandler, который принимает один параметр типа string – строку, то при вызове события нам надо передать в него строку.

Однако при вызове событий мы можем столкнуться с тем, что событие равно null в случае, если для его не определен обработчик. Поэтому при вызове события лучше его всегда проверять на null. Например, так:

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

Или так:

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

В этом случае поскольку событие представляет делегат, то мы можем его вызвать с помощью метода Invoke(), передав в него необходимые значения для параметров.

Объединим все вместе и создадим, и вызовем событие:

Листинг 7.32.

```
1  class Account
2  {
3      public delegate void AccountHandler(string
   message);
4      public event AccountHandler Notify; //1.
   Определение события
5      public Account(int sum)
6      {
7          Sum = sum;
8      }
9      public int Sum { get; private set;}
10     public void Put(int sum)
11     {
12         Sum += sum;
```

```
13          Notify?.Invoke($"На счет поступило: {sum}");   // 2.Вызов события
14      }
15      public void Take(int sum)
16      {
17          if (Sum >= sum)
18          {
19              Sum -= sum;
20              Notify?.Invoke($"Со счета снято: {sum}");   // 2.Вызов события
21          }
22          else
23          {
24              Notify?.Invoke($"Недостаточно денег на счете. Текущий баланс: {Sum}"); ;
25          }
26      }
27  }
```

Теперь с помощью события `Notify` мы уведомляем систему о том, что были добавлены средства и о том, что средства сняты со счета или на счете недостаточно средств.

Добавление обработчика события

С событием может быть связан один или несколько обработчиков. Обработчики событий - это именно то, что выполняется при вызове событий. Нередко в качестве обработчиков событий применяются методы. Каждый обработчик событий по списку параметров и возвращаемому типу должен соответствовать делегату, который представляет событие. Для добавления обработчика события применяется операция `+=`:

```
Notify += обработчик события;
```

Определим обработчики для события `Notify`, чтобы получить в программе нужные уведомления:

Листинг 7.33.

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
```

```
5           Account acc = new Account(100);
6           acc.Notify += DisplayMessage;    // Добавляем обработчик для события Notify
7           acc.Put(20);    // добавляем на счет 20
8           acc.Take(70);   // пытаемся снять со счета 70
9           acc.Take(180);  // пытаемся снять со счета 180
10          Console.Read();
11      }
12      private static void DisplayMessage(string message)
13      {
14          Console.WriteLine(message);
15      }
16  }
```

В данном случае в качестве обработчика используется метод `DisplayMessage`, который соответствует по списку параметров и возвращаемому типу делегату `AccountHandler`. В итоге при вызове события `Notify?.Invoke()` будет вызываться метод `DisplayMessage`, которому для параметра `message` будет передаваться строка, которая передается в `Notify?.Invoke()`. В `DisplayMessage` просто выводим полученное от события сообщение, но можно было бы определить любую логику.

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

Для одного события можно установить несколько обработчиков и потом в любой момент времени их удалить. Для удаления обработчиков применяется операция -=.

В качестве обработчиков могут использоваться не только обычные методы, но также делегаты, анонимные методы и лямбда-выражения.