Глава 15
Делегаты

Делегаты предназначены для случаев, в которых C++, Pascal и Modula используют указатели на функции. В отличие от последних в C++ делегаты являются полностью объектно-ориентированными. Также, в отличие от используемых в C++ указателей на элементы-функции, делегаты инкапсулируют как экземпляр объекта, так и метод.

БИЛЛ ВАГНЕР

Я испытываю странное чувство изумления, читая эту маленькую главу спецификации языка. Делегаты являются частью C#, начиная с версии 1.0. В то время большинство разработчиков из сообщества C# (включая меня) рассматривали делегаты как некую формальность, связанную с событиями. По прошествии десяти лет я уже не могу представить C# без делегатов. Они каждый день используются большинством из нас в LINQ-запросах, композиции функций, замыканиях и т. д. Делегаты и концепция работы с кодом как с данными являются столь неотъемлемой частью современной экосистемы .NET, что я не могу представить программирование на языке, который не позволяет мне выражать концепцию функций (и действий) как данные.

Объявление делегата определяет класс, унаследованный от класса `System.Delegate`. Экземпляр делегата инкапсулирует список вызова, состоящий из одного или нескольких методов, на каждый из которых ссылаются как на вызываемую сущность. Для методов экземпляра вызываемая сущность включает в себя экземпляр и метод этого экземпляра. Для статических методов она состоит только из метода. Вызов экземпляра делегата с подходящим набором аргументов приводит к вызову каждой из вызываемых сущностей делегата с передачей ей заданного набора аргументов.

Интересным и полезным свойством экземпляра делегата является то, что он не знает и не заботится о классах инкапсулируемых им методов; значение имеет только совместимость этих методов (раздел 15.1) с типом делегата. Это свойство делает делегаты идеальными для «анонимного» вызова.

ЭРИК ЛИППЕРТ

При первом знакомстве делегаты нередко сбивают разработчика с толку. Я предпочитаю думать о делегатах как о чем-то похожем на интерфейс с единственным методом; экземпляр типа делегата в таком случае является объектом, реализующим этот метод.

15.1. Объявления делегатов

Объявление-делегата — это *объявление-типа* (раздел 9.6), которое объявляет новый тип делегата.

объявление-делегата:
    атрибуты$_{opt}$ *модификаторы-делегата*$_{opt}$ **delegate** *тип-возвращаемого-значения*
        идентификатор список-параметров-вариантного-типа$_{opt}$
            (*список-формальных-параметров*$_{opt}$ **)** *ограничения-на-параметры-типа*$_{opt}$ **;**

модификаторы-делегата:
    модификаторы-делегата
    модификаторы-делегата *модификатор-делегата*

модификатор-делегата:
    new
    public
    protected
    internal
    private

Если один и тот же модификатор встречается в объявлении делегата несколько раз, возникает ошибка компиляции.

Модификатор `new` допускается только для делегатов, объявляемых внутри объявления другого типа. В этом случае модификатор указывает, что делегат скрывает унаследованный элемент с таким же именем, как описано в разделе 10.3.4.

Модификаторы `public`, `protected`, `internal` и `private` управляют доступом к делегату. В зависимости от контекста, в котором объявляется делегат, некоторые из этих модификаторов могут быть недопустимы (раздел 3.5.1).

Идентификатор определяет имя делегата.

Необязательный *список-формальных-параметров* определяет параметры делегата, а *тип-возвращаемого-значения* — тип возвращаемого значения делегата.

Необязательный *список-параметров-вариантного-типа* (раздел 13.1.3) определяет список параметров типа для самого делегата.

Тип возвращаемого значения типа делегата должен быть или `void`, или безопасным с точки зрения вывода (output-safe, раздел 13.1.3.1).

Все типы формальных параметров типа делегата должны быть безопасны с точки зрения ввода (input-safe). Помимо этого, любые типы параметров `out` или `ref` должны быть безопасны с точки зрения вывода. Заметьте, что даже параметры `out` должны быть безопасны с точки зрения ввода в силу ограничений базовой среды выполнения.

ЭРИК ЛИППЕРТ

Упомянутое здесь ограничение связано с тем, что на самом деле параметры `out` реализованы как параметры `ref`: вы можете и считывать, и записывать значения параметров `out`. Компилятор требует, чтобы в параметр `out` было записано значение, прежде чем его можно будет прочитать, однако это правило языка C#, а не среды выполнения. Если бы параметры `out` действительно предназначались «только для записи» и это

продолжение ⇨

ограничение присутствовало бы в среде выполнения, их теоретически можно было бы сделать ковариантными. Помните об этом, когда в следующий раз будете разрабатывать новую систему типов.

Эквивалентность типов делегатов в C# определяется именем, а не структурой. А именно, два различных типа делегатов, имеющих одинаковые списки параметров и тип возвращаемого значения, считаются разными типами делегатов. Однако экземпляры двух отдельных, но структурно эквивалентных типов делегатов при сравнении могут рассматриваться как равные (раздел 7.9.8).

Например:

```
delegate int D1(int i, double d);

class A
{
   public static int M1(int a, double b) {...}
}
class B

{
   delegate int D2(int c, double d);
   public static int M1(int f, double g) {...}
   public static void M2(int k, double l) {...}
   public static int M3(int g) {...}
   public static void M4(int g) {...}
}
```

Оба типа делегатов `D1` и `D2` совместимы с методами `A.M1` и `B.M1`, так как имеют тот же тип возвращаемого значения и список параметров; однако эти типы делегатов являются двумя разными типами, поэтому они не взаимозаменяемы. Типы делегатов `D1` и `D2` несовместимы с методами `B.M2`, `B.M3` и `B.M4`, так как имеют разные типы возвращаемого значения и списки параметров.

ЭРИК ЛИППЕРТ

Когда вы проектируете новую систему типов, вы не знаете точно, как она будет использоваться в будущем. Кто-то может представить сценарии использования, включающие различные категории делегатов (такие как «делегат для метода с наблюдаемыми побочными эффектами» или «делегат для метода, который может вызываться параллельно»). В этом случае имело бы смысл запретить присваивания между этими категориями. На практике оказывается, что структурная типизация делегатов очень полезна; на самом деле не существует таких семантических различий между `Predicate<int>` и `Func<int, bool>`, которые должны навязываться средой исполнения. Если бы нам пришлось разрабатывать все сначала, типы делегатов могли бы быть более структурно совместимы, чем они есть сегодня.

Как и в объявлениях других обобщенных типов, для создания сконструированного типа делегата должны быть заданы аргументы типа. Типы параметров и тип возвращаемого значения сконструированного типа делегата создаются путем

замены каждого параметра типа в объявлении делегата соответствующим ему аргументом типа сконструированного типа делегата. Получившиеся в результате тип возвращаемого значения и типы параметров используются для определения того, какие методы совместимы со сконструированным типом делегата. Например:

```
delegate bool Predicate<T>(T value);

class X
{
   static bool F(int i) {...}
   static bool G(string s) {...}
}
```

Тип делегата `Predicate<int>` совместим с методом `X.F`, а тип делегата `Predicate<string>` совместим с методом `X.G`.

Единственным способом объявить делегат является *объявление-делегата*. Делегат — это класс, унаследованный от `System.Delegate`. Из-за наличия неявного модификатора `sealed` не допускается наследование каких-либо типов от типа делегата. Также недопустимо наследовать от `System.Delegate` классы, не являющиеся делегатами. Обратите внимание, что `System.Delegate` сам по себе не является делегатом; это класс, от которого унаследованы все делегаты.

В C# определен специальный синтаксис для создания экземпляра и вызова делегата. За исключением создания экземпляра, любая операция, применимая к классу или экземпляру класса, также может быть применена соответственно к классу или экземпляру делегата. В частности, можно получать доступ к элементам типа `System.Delegate`, используя обычный синтаксис доступа к элементу.

Набор методов, инкапсулированных в экземпляре делегата, называется списком вызова. При создании экземпляра делегата (раздел 15.2) из единственного метода он инкапсулирует этот метод, и его список вызова содержит только одну запись. Однако когда объединяются два ненулевых экземпляра делегата, их списки вызова объединяются в порядке следования операндов — сначала левый, затем правый — для формирования нового списка, содержащего две или более записей.

ДЖОН СКИТ

Эта двойственность одноадресности и многоадресности в некоторых случаях крайне неудобна, а в других — невероятно полезна. Например, имеет смысл говорить о «цели» одноадресного делегата, но в то же время каждая запись в списке вызова многоадресного делегата может иметь отдельную цель. На практике это редко является проблемой, однако может затруднить простое и точное описание делегатов.

Делегаты объединяются с помощью бинарных операций `+` (раздел 7.8.4) и `+=` (раздел 7.17.2). Делегат можно удалить из объединения делегатов с помощью бинарных операций `-` (раздел 7.8.5) и `-=` (раздел 7.17.2). Делегаты также можно проверять на равенство (раздел 7.10.8).

Следующий пример демонстрирует создание нескольких экземпляров делегата и соответствующие им списки вызова:

```
delegate void D(int x);
class C
{
   public static void M1(int i) {...}
   public static void M2(int i) {...}
}

class Test
{
   static void Main()
   {
      D cd1 = new D(C.M1);      // M1
      D cd2 = new D(C.M2);      // M2
      D cd3 = cd1 + cd2;      // M1 + M2
      D cd4 = cd3 + cd1;      // M1 + M2 + M1
      D cd5 = cd4 + cd3;      // M1 + M2 + M1 + M1 + M2
   }
}
```

При создании экземпляров с именами `cd1` и `cd2` каждый из них инкапсулирует один метод. Когда создается `cd3`, его список вызова содержит два метода, `M1` и `M2`, именно в таком порядке. Список вызова `cd4` содержит `M1`, `M2` и `M1` в этом порядке. И наконец, список вызова `cd5` содержит `M1`, `M2`, `M1`, `M1` и `M2`, опять же в таком порядке. Другие примеры объединения (а также удаления) делегатов приведены в разделе 15.4.

15.2. Совместимость делегатов

Метод или делегат `M` *совместим* с делегатом типа `D`, если верны все следующие утверждения:

- `D` и `M` имеют одинаковое число параметров, и каждый параметр `D` имеет такой же модификатор `ref` или `out`, что и соответствующий ему параметр `M`.
- Для каждого параметра-значения (то есть не имеющего модификатора `ref` или `out`) существует тождественное преобразование (раздел 6.1.1) или неявное ссылочное преобразование (раздел 6.1.6) из типа параметра `D` в тип соответствующего параметра `M`.
- Для каждого параметра `ref` или `out` параметр `D` имеет такой же тип, что и `M`.
- Существует тождественное преобразование или неявное преобразование ссылки из типа возвращаемого значения `M` в тип возвращаемого значения `D`.

15.3. Создание экземпляра делегата

Экземпляр делегата создается *выражением-создания-делегата* (раздел 7.6.10.5) или преобразованием в тип делегата. Созданный экземпляр делегата ссылается на одну из следующих сущностей:

- статический метод, указанный в *выражении-создания-делегата*;
- целевой объект (который не может иметь значение `null`) и метод экземпляра, указанные в *выражении-создания-делегата*;
- другой делегат.

Например:

```
delegate void D(int x);

class C
{
   public static void M1(int i) {...}
   public void M2(int i) {...}
}

class Test
{
   static void Main()
   {
      D cd1 = new D(C.M1);      // Статический метод
      C t = new C();
      D cd2 = new D(t.M2);      // Метод экземпляра
      D cd3 = new D(cd2);       // Другой делегат
   }
}
```

После создания экземпляры делегатов всегда ссылаются на один и тот же целевой объект и метод. Помните, что при объединении двух делегатов или удалении одного из другого получившийся делегат содержит собственный список вызова; списки вызова объединяемых или удаляемых делегатов остаются неизменными.

15.4. Вызов делегата

Для вызова делегата в C# существует специальный синтаксис. Когда вызывается ненулевой экземпляр делегата, список вызова которого содержит одну запись, он вызывает один метод с теми же аргументами, которые были ему переданы, и возвращает то же значение, что и этот метод. (См. раздел 7.6.5.3 для более подробной информации о вызове делегата.) Если в процессе вызова такого делегата возникает исключение и оно не перехватывается внутри вызываемого метода, поиск оператора `catch` продолжается в вызвавшем делегат методе, как если бы он непосредственно вызывал метод, на который ссылается делегат.

Вызов экземпляра делегата, список вызова которого содержит несколько записей, осуществляется путем синхронного вызова каждого метода из списка по порядку. Каждому вызываемому методу передается тот же набор аргументов, который был передан экземпляру делегата. Если при вызове такого делегата используются параметры-ссылки (раздел 10.6.1.2), каждому методу передается ссылка на одну и ту же переменную; изменения этой переменной, сделанные одним методом из списка вызова, будут видимы всем последующим методам. Если при вызове деле-

гата используются выходные параметры или возвращаемое значение, их итоговое значение будет зависеть от вызова последнего делегата в списке.

Если в процессе обработки вызова такого делегата возникает исключение и оно не перехватывается внутри вызываемого метода, поиск оператора `catch` продолжается в методе, вызвавшем делегат, а все остальные методы из списка вызова не вызываются.

БИЛЛ ВАГНЕР

По причине такого поведения, в общем случае вы должны стремиться создавать такие методы, реализующие делегаты, которые ни при каких обстоятельствах не выбрасывают исключений. Это может привести к ошибкам, после которых, скорее всего, будет невозможно надежно восстановить нормальное состояние программы. Данный фактор не так важен, если вы знаете, что ваш делегат будет использоваться только как одноадресный.

КРИСТИАН НЕЙГЕЛ

Существует способ решить проблему с исключениями, выбрасываемыми в методах-обработчиках, на которые ссылается делегат. Вместо того чтобы напрямую вызывать экземпляр делегата, можно воспользоваться его методом `GetInvocationList`, чтобы вызвать каждый делегат из списка вызова по отдельности. Каждый такой вызов можно обезопасить, поместив его в блок `try-catch`. Одним из вариантов действий при возникновении исключения может быть удаление вызвавшего его метода-обработчика из списка вызова.

Попытка вызвать экземпляр делегата, имеющий значение `null`, приводит к возникновению исключения типа `System.NullReferenceException`.

ДЖОН СКИТ

В некоторой степени такой результат полностью обоснован; с другой стороны, он выглядит нелогичным, поскольку нулевое значение ссылки является нормальным представлением пустого списка вызова. Этой путаницы, вероятно, можно было бы избежать, если бы у каждого типа делегата был статический метод `Invoke` — возможно, даже метод расширения, — который выполнял бы соответствующую проверку на нулевое значение. Данное решение было бы более элегантным, если бы компилятор C# автоматически осуществлял эту проверку в каждом вызывающем выражении. Честно говоря, все усложняется, когда тип делегата имеет отличный от `void` тип возвращаемого значения или использует параметр `out`.
Каким бы ни было лучшее решение, необходимость проверки на нулевое значение при каждом вызове делегата, безусловно, раздражает.

ЭРИК ЛИППЕРТ

Распространенным шаблоном кода для «потокобезопасных» делегатов является что-то вроде этого:

```
var temp = this.mydelegate;
if (temp != null) temp();
```

вместо более очевидного решения:

```
if (this.mydelegate != null) this.mydelegate();
```

Первый фрагмент безопаснее, так как если существует возможность изменения `mydelegate` в другом потоке, его значение может поменяться на `null` между проверкой и вызовом. Кроме того, такой подход позволяет избежать только одного состояния гонки. Предположим, что мы используем первый вариант кода. Временная переменная сохраняет текущее значение `mydelegate`. В другом потоке `mydelegate` присваивается значение `null`, и глобальное состояние, требующееся предыдущему содержимому `mydelegate` для успешного выполнения, уничтожается. Но именно предыдущее содержимое мы собираемся вызвать! Если ваши делегаты (в частности, делегаты, ассоциированные с событиями) восприимчивы к этой проблеме, вы должны реализовать какой-то другой механизм, позволяющий удостовериться, что при возникновении гонки не произойдет ничего плохого. Возможно, наилучшим решением будет писать код таким образом, чтобы гарантировать, что если будет вызван такой «просроченный» делегат, ничего плохого не случится.

Следующий пример демонстрирует, как создавать экземпляры делегатов, объединять, удалять и вызывать их:

```
using System;
delegate void D(int x);

class C
{
   public static void M1(int i)
   {
      Console.WriteLine("C.M1: " + i);
   }

   public static void M2(int i)
   {
      Console.WriteLine("C.M2: " + i);
   }

   public void M3(int i)
   {
      Console.WriteLine("C.M3: " + i);
   }
}

class Test
{
   static void Main()
   {
      D cd1 = new D(C.M1);
      cd1(-1);                    // Вызвать M1

      D cd2 = new D(C.M2);
      cd2(-2);                    // Вызвать M2

      D cd3 = cd1 + cd2;
      cd3(10);                    // Вызвать сначала M1, затем M2
            cd3 += cd1;
      cd3(20);                    // Вызвать M1, M2, затем M1
```

продолжение ⇗

```
      C c = new C();
      D cd4 = new D(c.M3);
      cd3 += cd4;
      cd3(30);                    // Вызвать M1, M2, M1, затем M3

      cd3 -= cd1;                 // Удалить последний M1
      cd3(40);                    // Вызвать M1, M2, затем M3

      cd3 -= cd4;
      cd3(50);                    // Вызвать M1, затем M2

      cd3 -= cd2;
      cd3(60);                    // Вызвать M1

      cd3 -= cd2;                 // Невозможность удаления не приводит к ошибке
      cd3(60);                    // Вызвать M1

      cd3 -= cd1;                 // Список вызова пуст, поэтому
                                  // cd3 имеет значение null

      //      cd3(70);            // Выбрасывается исключение
                                  // System.NullReferenceException

      cd3 -= cd1;                 // Невозможность удаления не приводит к ошибке
   }
}
```

Как показано в операторе `cd3 += cd1;`, делегат может входить в список вызова несколько раз. В этом случае он просто вызывается по одному разу для каждого вхождения. При удалении этого делегата из такого списка вызова фактически удаляется его последнее вхождение.

Непосредственно перед выполнением последнего оператора, `cd3 -= cd1;`, делегат `cd3` ссылается на пустой список вызова. Попытка удаления делегата из пустого списка (или удаления несуществующего делегата из непустого списка) не является ошибкой.

Результатом выполнения программы будет следующий вывод:

```
C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60
```