Глава 2

Язык C# 1 — основа основ

В этой главе...

- Делегаты
- Характеристики системы типов
- Типы значений и ссылочные типы

В целом, язык C# 1 — это не новость. Давайте отбросим недомолвки в сторону. Я показал бы себя не с лучшей стороны, если бы попытался рассматривать все возможности первой версии языка C# в одной главе. Я писал эту книгу, подразумевая, что мои читатели, по крайней мере, относительно компетентны в языке C# 1. Под "относительной компетентностью" я подразумеваю, что вы, *по крайней мере*, смогли бы ответить на технические вопросы, как младший разработчик (junior developer) C#. Я ожидаю, что у многих читателей есть и больший практический опыт, вот какой уровень знаний я подразумевал.

В этой главе сосредоточимся на трех областях языка C# 1, которые особенно важны для понимания средств более поздних версий. Их можно считать "наименьшим общим знаменателем", на основе которого я мог бы сделать немного больше предположений далее в книге. С учетом данного наименьшего общего знаменателя, вы можете найти, что у вас уже есть полное понимание всех концепций этой главы. Если вы полагаете, что это так, то не стесняйтесь пропустить данную главу, даже не читая ее. Вы всегда можете вернуться к ней позже, если что-то окажется не так просто, как вы думали. Вы могли бы, по крайней мере, просмотреть резюме в конце каждого раздела, где подчеркнуты важнейшие пункты, и если любой из них покажется незнакомым, то стоит прочитать этот раздел подробнее.

Для начала рассмотрим делегаты, затем опишем, насколько система типов C# сравнима с некоторыми другими возможностями, и, наконец, выясним различия между ти-

пами значений и ссылочными типами. По каждой теме я опишу идеи и действия, а также воспользуюсь возможностью определить термины, чтобы я мог использовать их позже. После рассмотрения работы C# 1 я представлю краткий обзор того, как множество новых средств более поздних версий касаются тем, затронутых в этой главе.

2.1. Делегаты

Я уверен, что у вас уже есть интуитивное понимание концепции делегата, несмотря даже на то, что вам, возможно, трудно его четко сформулировать. Если вы знакомы с языком C и хотите описать делегаты другому программисту C, то термин *указатель на функцию* (function pointer), без сомнения, решил бы дело. По существу, делегаты обеспечивают уровень косвенного обращения: вместо того чтобы определить действие, которое будет выполнено непосредственно, его можно так или иначе "упаковать" в объект. Затем этот объект может использоваться как любой другой, а одной из операций, которую вы можете осуществить с ним, будет выполнение инкапсулируемого действия. В качестве альтернативы вы можете считать тип делегата интерфейсом одного метода, а экземпляр делегата — объектом, реализующим этот интерфейс.

Если для вас это только напыщенные речи, то, возможно, поможет пример. Возможно, он немного болезненный, но действительно схватывает суть делегатов. Рассмотрим ваше завещание — вашу последнюю волю. Это набор инструкций, например "оплатить счета, внести пожертвование на благотворительность, а остальную часть состояния на уход за котом". Вы пишете это *прежде*, чем умрете, и оставляете, соответственно, в надежном месте. *После* вашей смерти поверенный (вы надеетесь!) будет действовать по этим инструкциям.

Делегат в C# действует как завещание в реальном мире — это последовательность действий, которые будут выполнены в подходящее время. Делегаты обычно используются тогда, когда код должен выполнить действия, не зная точно, каковы эти действия должны быть. Например, единственная причина, по которой класс `Thread` знает, что при запуске нужно работать в новом потоке, — это предоставленный вами конструктор с экземпляром делегата `ThreadStart` или `ParameterizedThreadStart`.

Для начала рассмотрим четыре фундаментальные основы делегатов, без которых ни один из них не имел бы смысла.

2.1.1 Рецепт для простых делегатов

Чтобы делегаты могли что-нибудь сделать, должны выполняться четыре условия.

- Тип делегата должен быть объявлен.
- Должен существовать метод, содержащий код для выполнения.
- Экземпляр делегата должен быть создан.
- Экземпляр делегата должен быть вызван.

Давайте подробно рассмотрим каждый пункт этого рецепта.

Объявление типа делегата

Тип делегата (delegate type) — это фактически список типов параметров и типов возвращаемых значений. Он определяет действие, которое может быть представлено экземпляром типа. Предположим, например, что тип делегата объявлен так.

```
delegate void StringProcessor(string input);
```

Этот код говорит, что если мы хотим создать экземпляр делегата `StringProcessor`, необходим метод с одним параметром (строковым) и возвращаемым значением типа `void` (т.е. метод не возвращает ничего). Важно понять, что тип `StringProcessor` действительно происходит от типа `System.MulticastDelegate`, который, в свою очередь, происходит от типа `System.Delegate`. У типа есть методы, позволяющие создавать его экземпляры и передавать ссылки на эти экземпляры для работы. Есть, безусловно, несколько "специальных средств", но если вы когда-нибудь зададитесь вопросом, что происходит в конкретной ситуации, сначала подумайте о том, что произошло бы при использовании только обычного ссылочного типа.

Источник недоразумений: неоднозначный термин *делегат*

Делегаты могут быть неправильно поняты потому, что слово *делегат* (delegate) зачастую используется для описания как *типа делегата*, так и *экземпляра делегата*. Различие между этими концепциями точно такое же, как между любым другим типом и его экземпляром; сам тип `string`, например, — это вовсе не специфическая последовательность символов. В этой главе я использовал термины *тип делегата* и *экземпляр делегата*, чтобы попробовать сохранить ясность.

При рассмотрении следующего пункта мы будем использовать тип делегата `StringProcessor`.

Поиск соответствующего метода для действия экземпляра делегата

Наш следующий пункт — поиск (или написание) метода, который делает то, что мы хотим, и имеет ту же сигнатуру, что и тип используемого делегата. Идея в том, чтобы при попытке вызова экземпляра делегата гарантировать совпадение всех используемых нами параметров и возможность применения возвращаемого значения (если оно есть) таким способом, который мы ожидаем, точно так же как при обычном вызове метода.

Теперь предположим, что следующие пять сигнатур методов являются кандидатами на использование для экземпляра `StringProcessor`.

```
void PrintString(string x)
void PrintInteger(int x)
void PrintTwoStrings(string x, string y)
int GetStringLength(string x)
void PrintObject(object x)
```

У первого метода все правильно, таким образом, мы можем использовать его для создания экземпляра делегата. У второго метода есть один параметр, но не типа `string`, поэтому он несовместим с делегатом `StringProcessor`. У третьего метода правильный тип первого параметра, но у него есть второй параметр, поэтому он также несовместим.

У четвертого метода правильный список параметров, но неподходящий тип возвращаемого значения. (Если бы у нашего типа делегата был тип возвращаемого значения, то тип возвращаемого значения метода также должен был бы совпадать.) Пятый метод интересен. Каждый раз, обращаясь к экземпляру делегата `StringProcessor`, мы можем вызвать метод `PrintObject` с теми же аргументами, поскольку тип `string` происходит от типа `object`. Это могло бы позволить использовать его для экземпляра делегата `StringProcessor`, но язык C# 1 ограничивает делегат передачей параметра *точно* того же типа[1]. В языке C# 2 ситуация иная; более подробная информация по этой теме приведена в главе 5. Четвертый метод, до некоторой степени, подобен предыдущему, поскольку вы всегда можете

[1] Кроме типов параметров, должны совпадать их модификаторы `out` (значение по умолчанию) или `ref`. Хотя параметры с модификаторами `out` и `ref` используются для делегатов довольно редко.

игнорировать нежелательное возвращаемое значение. Но пустые и непустые типы возвращаемого значения, как здесь, всегда считаются несовместимыми. Частично это связано с тем, что другие аспекты системы (особенно JIT) должны знать, будет ли при выполнении метода значение оставлено в стеке как возвращаемое значение[2].

Предположим, что теперь есть тело метода (`PrintString`) с подходящей сигнатурой, и перейдем к следующему пункту, самому экземпляру делегата.

Создание экземпляра делегата

Теперь, когда имеется тип делегата и метод с правильной сигнатурой, мы можем создать экземпляр делегата этого типа, определив, что данный метод будет выполнен, когда будет вызван экземпляр делегата. Официально никакая терминология для этого не определена, но в данной книге я назову это *действием* (action) экземпляра делегата. Точная форма выражения, используемого для создания экземпляра делегата, зависит от того, использует ли действие метод экземпляра или статический метод. Предположим, что метод `PrintString` статический и определен в типе по имени `StaticMethods`, а метод экземпляра определен в типе по имени `InstanceMethods`. Вот два примера создания экземпляра делегата `StringProcessor`.

```
StringProcessor proc1, proc2;
proc1 = new StringProcessor(StaticMethods.PrintString);
InstanceMethods instance = new InstanceMethods();
proc2 = new StringProcessor(instance.PrintString);
```

Когда действие является статическим методом, вы должны определить только название типа. Когда действие является методом экземпляра, необходим экземпляр типа (или типа, производного от него), точно так же как при вызове метода обычным способом. Этот объект называется *целью* (target) действия, а при обращении к экземпляру делегата будет вызван метод этого объекта. Если действие находится в пределах того же класса (как это обычно бывает, особенно когда вы пишете обработчики событий в коде пользовательского интерфейса), вы не обязаны квалифицировать его так или иначе, для методов экземпляров неявно используется ссылка `this`[3]. Это правило тоже выполняется только при непосредственном вызове метода.

Совершенный "мусор"! (или нет, в зависимости от обстоятельств...)

Следует знать, что экземпляр делегата будет препятствовать удалению своей цели при сборе "мусора", если сам экземпляр делегата не может быть собран. Это может привести к утечке памяти, особенно когда "недолговечный" объект подписался на событие "долговечного" объекта, используя себя как цель. Долговечный объект косвенно удерживает ссылку на недолговечный объект, продлевая его существование.

В создании экземпляра делегата нет особого смысла, если он не будет вызван в некоторый момент. Давайте перейдем к нашему последнему пункту.

Вызов экземпляра делегата

Эта часть действительно простая[4] — всего лишь вызов метода экземпляра делегата. Для самого вызова метода используется метод `Invoke`, он всегда присутствует в типе

---

[2] Слово *стек* здесь преднамеренно использовано неопределенно, чтобы избежать слишком большого количества ненужных пока подробностей. Более подробная информация по этой теме приведена в посте *The void is invariant* блога Эрика Липперта по адресу: `http://mng.bz/4g58`.

[3] Конечно, если действие является методом экземпляра и вы попытаетесь создавать экземпляр делегата из статического метода, то вы все же должны будете предоставить ссылку на цель.

[4] Во всяком случае, при синхронном вызове. Для асинхронного вызова экземпляра делегата вы можете использовать методы `BeginInvoke` и `EndInvoke`, но в данной главе это не рассматривается.

делегата с тем же списком параметров и типом возвращаемого значения, что и в описании типа делегата. В нашем случае есть такой метод.

```
void Invoke(string input)
```

Вызов метода `Invoke` выполнит действие экземпляра делегата, передав любые аргументы, которые вы определили, а также вернет возвращаемое значение действия (если тип возвращаемого значения не указан как `void`).

Как это ни просто, но язык C# еще упрощает дело, если у вас есть переменные[5], тип которых совпадает с типом делегата. Вы можете рассматривать их как сам метод. Происходящее можно также рассматривать как цепь событий, осуществляемых в соответствующий момент, как показано на рис. 2.1.

Рис. 2.1. Обработка обращения к экземпляру делегата, использующего сокращенный синтаксис C#

Вот и все, что нужно. Теперь все наши ингредиенты собраны, осталось лишь предварительно подогреть CLR до 200°C, перемешать их тщательно и посмотреть, что испечется.

Полный пример и некоторые мотивации

Все это лучше рассмотреть в действии на полном примере, и, наконец, мы сможем хоть что-то фактически запустить! На сей раз, вместо фрагмента, я включил в пример полный исходный код. В листинге 2.1 нет ничего сверхсложного, так что не ожидайте и ничего поразительного, только конкретный код, который можно теперь обсудить.

Листинг 2.1. Использование делегатов множеством простых способов

```
using System;
delegate void StringProcessor(string input); // #1 Объявление типа
                                             // делегата
class Person
{
    string name;
    public Person(string name) { this.name = name; }
    public void Say(string message)       // #2 Объявление совместимого
                                          // метода экземпляра
    {
        Console.WriteLine("{0} says: {1}", name, message);
    }
```

[5] Или любой другой вид выражения, но обычно это переменная.

```
}
class Background
{
    public static void Note(string note) // #3 Объявление совместимого
                                         // статического метода
    {
        Console.WriteLine("({0})", note);
    }
}
class SimpleDelegateUse
{
    static void Main()
    {
        Person jon = new Person("Jon");
        Person tom = new Person("Tom");
        StringProcessor jonsVoice, tomsVoice, background;  // #4 Создание
        jonsVoice = new StringProcessor(jon.Say);          // трех
        tomsVoice = new StringProcessor(tom.Say);          // экземпляров
        background = new StringProcessor(Background.Note); // делегата

        jonsVoice("Hello, son.");                          // #5 Вызовы
        tomsVoice.Invoke("Hello, Daddy!");                 // экземпляров
        background("An airplane flies past.");             // делегатов
    }
}
```

Для начала объявляем тип делегата #1. Затем создаем два метода (#2 и #3), совместимых с типом делегата. Имеется один метод экземпляра (`Person.Say`) и один статический метод (`Background.Note`), чтобы мы могли увидеть, как они используются по-разному, когда мы создаем экземпляры делегатов #4. Мы создали два экземпляра класса `Person`, чтобы увидеть значение, которое имеет целевой объект делегата. При обращении к экземпляру делегата `jonsVoice` #5 вызывается метод `Say` объекта `Person` по имени *Jon*; аналогично при обращении к экземпляру делегата `tomsVoice` он использует объект по имени *Tom*. Я продемонстрировал также разные способы обращения к экземпляру делегата: явный вызов метода и сокращенный синтаксис C#. Обычно используют только сокращение. Вывод листинга 2.1 довольно очевиден.

```
Jon says: Hello, son.
Tom says: Hello, Daddy!
(An airplane flies past.)
```

Откровенно говоря, в листинге 2.1 слишком много кода, чтобы отобразить три строки вывода. Даже если мы хотели использовать классы `Person` и `Background`, здесь нет никакой реальной необходимости в использовании делегатов. Так в чем же смысл? Почему мы не можем просто вызвать методы непосредственно? Ответ находится в нашем первоначальном примере с поверенным, выполняющим завещание только потому, что вы хотите, чтобы нечто произошло, когда вас не окажется в нужном месте в нужное время, чтобы самому поруководить происходящим. Иногда необходимо оставить инструкции, чтобы *делегировать* ответственность.

Должен подчеркнуть, что в мире программного обеспечения объекты не оставляют завещаний. Нередко объект, который первоначально создал экземпляр делегата, все еще жив и здравствует, когда вызывается экземпляр делегата. Вместо этого определяется некоторый код, который будет выполнен в определенное время, когда вы окажетесь не в состоянии (или, возможно, не захотите) изменить код, который выполняется в этот момент. Если необходимо, чтобы нечто произошло при щелчке на кнопке, не нужно изменять код *кнопки*, достаточно указать, чтобы кнопка вызвала один из моих методов, а он

предпримет соответствующее действие. Это вопрос дополнительного уровня *косвенного доступа*, на котором построена большая часть объектно-ориентированного программирования. Безусловно, это дополнительная сложность (посмотрите, сколько потребовалось строк кода для такого небольшого вывода!), но также и гибкость.

Теперь, когда мы больше знаем о простых делегатах, давайте коротко рассмотрим объединение делегатов, выполняющих целую связку действий, вместо одного.

2.1.2. Объединение и удаление делегатов

У всех рассмотренных до сих пор экземпляров делегатов было только одно действие. Реальность немного сложнее: у экземпляра делегата фактически есть список действий. Он называется *списком вызовов* (invocation list) экземпляра делегата. Статические методы `Combine` и `Remove` типа `System.Delegate` несут ответственность, соответственно, за создание новых экземпляров делегата за счет объединения списков вызовов двух экземпляров делегатов или удаления списка вызовов одного экземпляра делегата из другого.

Делегаты неизменяемы

Как только вы создали экземпляр делегата, уже ничто в нем не может быть изменено. Это позволяет безопасно распространять ссылки на экземпляры делегатов и объединять их с другими делегатами, не заботясь о целостности, безопасности потоков и попытках изменить их действия. Это аналогично строкам, которые также неизменяемы. Я упоминаю об этом потому, что метод `Delegate.Combine` аналогичен методу `String.Concat`, оба они объединяют существующие экземпляры, чтобы сформировать новый экземпляр, не изменяя исходные объекты. В случае экземпляров делегатов, объединяются исходные списки вызовов. Обратите внимание, если вы попытаетесь объединить с экземпляром делегата значение `null`, оно рассматривается как экземпляр делегата с пустым списком вызовов.

Вы не часто увидите явный вызов метода `Delegate.Combine`, в коде C# обычно используются операторы `+` и `+=`. На рис. 2.2 демонстрируется процесс преобразования, где `x` и `y` — две переменные того же (или совместимого) типа делегата. Все это осуществляет компилятор C#.

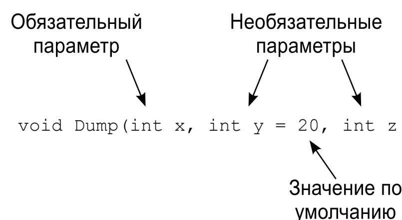

Рис. 2.2. Процесс преобразования, используемый для сокращенного синтаксиса C# при объединении экземпляров делегатов

Как можно заметить, это простое преобразование, но оно действительно делает код намного опрятнее. Подобно тому, как вы можете объединить экземпляры делегатов, вы можете удалить один из них из другого при помощи метода `Delegate.Remove`, а C# использует сокращенный синтаксис операторов `-` и `-=` вполне очевидным способом. Код `Delegate.Remove(source, value)` создает новый делегат, список вызовов которого состоит из списка `source`, но без списка `value`. Если в результате получается пустой список вызовов, возвращается значение `null`.

Когда вызывается экземпляр делегата, все его действия выполняются по порядку. Если тип возвращаемого значения сигнатуры делегата не `void`, метод `Invoke` вернет значение, возвращаемое *последним* выполняемым действием. Экземпляр делегата с типом, отличным от `void`, с более чем одним действием в списке вызовов встречается редко, поскольку это означает, что возвращаемые значения всех других действий никогда не доступны, если вызывающий код не выполняет их явно, используя метод `Delegate.GetInvocationList` для выбора действия из списка.

Если любое действие в списке вызовов передаст исключение, это воспрепятствует выполнению любого последующего действия. Например, если вызывается экземпляр делегата со списком вызовов `[a, b, c]` и действие `b` передает исключение, то исключение начнет распространяться немедленно и действие `c` не будет выполнено.

Объединение и удаление экземпляров делегатов особенно полезно тогда, когда дело доходит до событий. Теперь, когда мы понимаем, что подразумевает объединение и удаление, можно переходить к разговору о событиях.

2.1.3. Кратко о событиях

У вас, вероятно, есть интуитивное понимание общей *точки* (point) событий, особенно если вам приходилось писать какой-нибудь пользовательский интерфейс. Идея в том, что событие позволяет коду реагировать, когда нечто происходит, например сохранять файл при щелчке на соответствующей кнопке. В данном случае, событие — это щелчок на кнопке, а действие — сохранение файла. Понимание задачи концепции — это еще не понимание того, как язык C# определяет события в терминах языка.

Разработчики часто путают события и экземпляры делегатов или события и поля, объявленные с типом делегата. Различие существенно: *события — это не поля*. Причина недоразумения в том, что язык C# снова и снова предоставляет сокращения в форме *полеподобных событий*. Мы еще вернемся к ним, но сначала давайте рассмотрим, что такое события, из чего они состоят и как их понимает компилятор C#.

События можно считать подобием свойств. Они оба объявляются как определенный тип, который в случае события рассматривается как тип делегата. Когда вы используете свойства, это только *выглядит*, как будто вы выбираете или присваиваете значение непосредственно полю, но *фактически* вы вызываете методы (*методы получения значения* (getter) и *методы установки значения* (setter)). Реализация свойства может делать что угодно в пределах этих методов, хотя, как правило, для простых свойств реализуют поддержку их внутренних полей, иногда некую проверку значения в методе установки значения, а иногда и средства обеспечения безопасности потока.

Аналогично, когда вы подписываетесь на события или аннулируете подписку, это только *выглядит*, как будто вы используете поле типа делегата с операторами `+=` и `-=`. Тем не менее вы фактически вызываете методы (`add` и `remove`).[6] Это все, что вы можете сделать с событием, — подписаться на него (добавить обработчик события) или аннулировать подписку (удалить обработчик событий). Чтобы методы событий делали нечто полезное, например замечали обработчики событий, вы попытаетесь добавить или удалить их и сделать доступными в другом месте в пределах класса.

Причина существования событий, в первую очередь, очень похожа на причину существования свойств: это дополнительный слой инкапсуляции, реализующий шаблон

[6] Они не имеют таких имен в откомпилированной программе; в противном случае у вас могло бы быть только одно событие на каждый тип. Компилятор создает два метода с именами, которые не используются в другом месте, а специальная часть метаданных позволяет другим типам узнать, что существует событие с таким именем и что вызываются методы `add` и `remove`.

"публикация/подписка" (см. `http://mng.bz/otVt`). Вы не хотите, чтобы другой код был в состоянии установить значение поля без его владельца, по крайней мере, без возможности проверки правильности нового значения, вам также не часто нужно, чтобы код вне класса был в состоянии произвольно изменять (или вызывать) обработчики события. Конечно, в класс *можно* добавить методы для дополнительного доступа, например для сброса списка обработчиков события, или срабатывания события (другими словами, вызова его обработчика). Например, метод `BackgroundWorker.OnProgressChanged` просто вызывает обработчик события `ProgressChanged`. Но если вы предоставляете только само событие, у кода вне класса есть способность *только* добавить и удалить обработчики.

Полеподобные события существенно упрощают реализацию всего этого — одно объявление, и все готово. Компилятор превращает объявление в событие со стандартными реализациями методов добавления и удаления, а также закрытое поле того же типа. Код внутри класса видит поле; код вне класса видит только событие. Это создает *впечатление*, что вы можете вызвать событие, но на самом деле для фактического вызова обработчика события осуществляется вызов экземпляра делегата, хранящегося в поле.

Подробности событий не рассматриваются в этой главе, в более поздних версиях языка C# события изменились не очень сильно[7], но я хотел привлечь внимание к различию между экземплярами делегата и событиями сейчас, чтобы предотвратить недоразумения в будущем.

2.1.4. Резюме по делегатам

Таким образом, мы рассмотрели подробно следующее.

- Делегаты инкапсулируют действие со специфическим типом возвращаемого значения и набором параметров подобно интерфейсу одиночного метода.
- Сигнатура типа, описанная объявлением типа делегата, определяет, какие методы применяются для создания экземпляров делегата, а также сигнатуру для их вызова.
- Для создания экземпляра делегата требуется метод (для методов экземпляра) и цель обращения к методу.
- Экземпляры делегата неизменяемы.
- Каждый экземпляр делегата содержит список вызовов; он же список действий.
- Экземпляры делегата могут быть объединены или удалены друг из друга.
- События не являются экземплярами делегата, это только пары методов `add/remove` (считайте свойства методами получения и установки значений).

Делегаты — одна из специфических возможностей языка C# и инфраструктуры .NET, а также деталь грандиозного механизма. Следующих два раздела этой главы посвящены более широким темам. Сначала мы рассмотрим то, что означает термин *статически типизированный* (statically typed) язык и какое значение это имеет.

2.2. Характеристики системы типов

Почти у каждого языка программирования есть система типов некоторого вида. На протяжении довольно долгого времени они классифицировались как строгие и нестрогие, безопасные и небезопасные, статические и динамические, были, без сомнения, и

[7] В языке C# 4 есть очень небольшие изменения в событиях, подобных полям. Более подробная информация по этой теме приведена в разделе 4.2.

некоторые более экзотические варианты. Вполне очевидно, что важно понимать систему типов, с которой работаешь, а также имеет смысл знать недостатки языка и любую полезную информацию о нем. Но поскольку разными людьми используются разные термины, отсутствие взаимопонимания почти неизбежно. Чтобы избежать недоразумений, я постараюсь *точно* оговорить, что подразумеваются под каждым термином.

В этом разделе важно обратить внимание на то, что он применим только к "*безопасному*" ("safe") коду, а это весь код C#, который не расположен непосредственно в пределах небезопасного контекста. Как можно судить по названию, код в пределах небезопасного контекста может делать такое, на что неспособен безопасный код, а также может нарушать аспекты нормальной системы безопасности типов, хотя система типов все еще безопасна во всех других отношениях. Большинству разработчиков редко приходится писать небезопасный код, и характеристики системы типов существенно упрощены в части их описания и объяснения, когда рассматривается только безопасный код.

Этот раздел демонстрирует, какие ограничения налагаются в языке C# 1, а какие нет, при определении некоторых терминов для описания действий. Затем мы рассмотрим, что невозможно сделать в языке C# 1, сначала с точки зрения того, что мы *не можем* указать компилятору, а затем с точки зрения того, что мы *не должны* указывать компилятору.

Давайте начнем с того, что язык C# 1 делает и какая терминология обычно используется для описания этого.

2.2.1. Место языка C# в мире систем типов

Проще всего высказать утверждение, а затем разъяснить то, что оно фактически означает, и возможные варианты.

Система типов языка C# 1 является статической, явной и безопасной.

Вы, вероятно, ожидали увидеть здесь слово *строгой*, и я был бы не прочь включить его. Хотя большинство людей способны разумно выяснить, есть ли у языка перечисленные выше характеристики, и решить, строго ли он типизирован, это *может* вызвать горячие дебаты, поскольку мнения иногда существенно различаются. Некоторые критерии (не допускающие никаких толкований, ни явных, ни неявных) однозначно исключили бы язык C#, тогда как другие весьма приближают его (и даже весьма) к *статически типизированным*, включая язык C# 1. Большинство статей и книг, которые я читал, описывают язык C# как строго типизированный, фактически имея в виду, что он статически типизированный.

Давайте переберем все термины определения по одному и прольем на них некоторый свет.

Статическая типизация против динамической

Язык C# 1 *статически типизированный*: каждая переменная[8] имеет определенный тип, и этот тип известен на момент компиляции. Допустимы только те операции, которым известен тип, и это поддерживается компилятором. Рассмотрим пример поддержки.

```
object o = "hello";
Console.WriteLine(o.Length);
```

Как разработчики, мы, глядя на этот код, безусловно, понимаем, что значение o является строкой, а у типа string есть свойство Length, но компилятор считает o только именем сущности типа object. Если мы захотим добраться до свойства Length, придется указать компилятору, что значение o фактически является строкой.

[8] Это относится и к большинству выражений, но не ко всем. Некоторые выражения не имеют типа, например вызовы методов void, но это никак не влияет на статус языка C# 1 как статически типизированного. Я использую всюду в этом разделе термин *переменная*, чтобы избежать ненужного напряжения мозгов.

```
object o = "hello";
Console.WriteLine(((string)o).Length);
```

Теперь компилятор в состоянии найти свойство Length типа System.String. Это позволяет ему проверить правильность вызова и создать соответствующий код IL, а также использовать тип большего выражения. Тип времени компиляции выражения называется также статическим типом, поэтому мы можем сказать, например, так: “статическим типом o является System.Object”.

Почему типизация называется статической

Слово *статическая* (static) используется для описания типизации этого вида, потому что анализ доступных операций выполняется с использованием *неизменяемых* данных: типов выражений времени компиляции. Предположим, что объявлена переменная типа Stream. Тип переменной не изменится, даже если значение переменной сменится ссылкой на MemoryStream, FileStream или вообще не на поток (нулевую ссылку). Даже в пределах систем статических типов, конечно, могут быть некоторые динамические действия: фактическая *реализация*, исполняемая вызовом виртуального метода, будет зависеть от значения, к которому обращаются. Хотя идея неизменяемой информации также является причиной применения модификатора static, статические члены, как правило, проще считать принадлежностью самого типа, а не любого из его экземпляров. В большинстве случаев на практике вы можете считать эти два смысла одного слова совершенно разными.

Альтернатива статической типизации — это *динамическая типизация* (dynamic typing), которая может принимать множество обликов. Сущность динамической типизации в том, что у переменных есть только значения, они не ограничиваются специфическими типами. Таким образом, компилятор не будет выполнять проверки некоторого вида. Вместо этого среда выполнения пытается понять любые данные выражения соответствующим способом. Например, если бы язык C# 1 *был* динамически типизированным, то мы могли бы сделать следующее.

ЧТО, ЕСЛИ?

```
o = "hello";
Console.WriteLine(o.Length);
o = new string[] {"hi", "there"};
Console.WriteLine(o.Length);
```

Это два совершенно несвязанных свойства Length типов String.Length и Array.Length, исследуемых динамически во время выполнения. Подобно многим областям системы определения типов, есть разные уровни динамической типизации. Некоторые языки позволяют определять типы где угодно — возможна даже их динамическая обработка, кроме присвоения, — и использовать нетипизированные переменные повсеместно.

Хотя я регулярно упоминаю в этом описании только язык C# 1, язык C# является полностью статически типизированным вплоть до версии C# 3. Позже мы увидим, что язык C# 4 вводит некоторую динамическую типизацию, хотя подавляющее большинство кода приложений C# 4 все еще использует статическую типизацию.

Явная типизация против неявной

Различие между *явной типизацией* (explicit typing) и *неявной типизацией* (implicit typing) уместно только в статически типизированных языках. При явной типизации тип каждой переменной должен быть явно указан в объявлении. Неявная типизация позволяет компилятору выяснять тип переменной на основании способа ее использования. Например, язык может продиктовать, что тип переменной — это тип выражения, используемого для присвоения исходного значения.

Рассмотрим гипотетический язык, который использует ключевое слово var для обозначения выведения типов[9]. В табл. 2.1 демонстрируется, как код на таком языке мог быть переписан на языке C# 1. Код в левом столбце *недопустим* в языке C# 1, а код в правом столбце — эквивалентный допустимый код.

Таблица 2.1. Пример демонстрирует различия между неявной и явной типизацией

Недопустимо в C# 1 — неявная типизация	Допустимо в C# 1 — явная типизация
`var s = "hello";`	`string s = "hello";`
`var x = s.Length;`	`int x = s.Length;`
`var twiceX = x * 2;`	`int twiceX = x * 2;`

Полагаю, теперь понятно, почему это допустимо только для статически типизированных ситуаций: и при неявной, и при явной типизации тип переменной *известен* на момент компиляции, даже если он не заявлен непосредственно. В динамическом контексте переменная на момент компиляции *не имеет* типа.

Типизированный против нетипизированного

Самый простой способ описать типизированную систему состоит в описании ее противоположности. Некоторые языки (особенно C и C++) действительно позволяют делать некоторые удивительные вещи. Потенциально они в определенных ситуациях чрезвычайно могущественны, но, как и бесплатный пакет пончиков, встречаются относительно редко. Некоторые из этих удивительных вещей способны сильно навредить вам, если обращаться с ними неправильно. Одной из них является нарушение системы типов.

Правильный ритуал вуду может убедить эти языки считать значение одного типа значением *совершенно другого* типа, без всяких преобразований. Я не имею в виду просто вызов метода, у которого оказалось то же имя, как в нашем примере динамической типизации ранее. Я подразумеваю код, который видит набор байтов в пределах значения, но интерпретирует их "неправильно". Листинг 2.2, простой пример кода C, демонстрирует, что я имею в виду.

Листинг 2.2. Демонстрация нетипизированной системы в коде C

```
#include <stdio.h>
int main(int argc, char**argv)
{
    char *first_arg = argv[1];
    int *first_arg_as_int = (int *)first_arg;
    printf ("%d", *first_arg_as_int);
}
```

Если вы откомпилируете листинг 2.2 и запустите его с простым аргументом "hello", то увидите значение 1819043176, по крайней мере, на архитектуре с порядком байтов от младшего к старшему (little-endian), с компилятором, рассматривающим тип int как 32-битовый, а тип char как 8-битовый, и где текст представлен кодами ASCII или UTF-8. Код считает указатель на тип char указателем на тип int. Таким образом, оператор обращения к значению возвращает первые 4 байт текста как число.

Фактически этот крошечный пример незначителен, по сравнению с другой потенциальной проблемой — приведение типов между совершенно несвязанными структурами может легко привести к полному краху. Не то, чтобы в реальной жизни это случалось

---

[9] Не столь уж и гипотетический. См. в разделе 8.2 возможности неявно типизированных локальных переменных языка C# 3.

очень часто, но некоторые элементы системы типов языка C нередко требуют, чтобы вы точно указали компилятору, что делать, не оставляя ему никаких возможностей, кроме как доверять вам даже во время выполнения.

К счастью, в языке C# ничего подобного не происходит. Да, доступно много преобразований, но вы не можете притвориться, что данные объекта одного типа фактически являются данными другого типа. Вы можете *попробовать* добавить приведения, чтобы предоставить компилятору дополнительную (и неправильную) информацию, но если компилятор решит, что это приведение фактически *невозможно*, произойдет ошибка компиляции, а если это теоретически возможно, но фактически неправильно во времени выполнения, то исключение передаст среда CLR.

Теперь, когда известно немного больше о том, как язык C# 1 вписывается в общую картину систем типов, я хотел бы упомянуть о некоторых недостатках его выбора. Я не хочу сказать, что выбор *абсолютно неправильный*, но некоторые ограничения существуют. Зачастую разработчики языка вынуждены выбирать между различными путями, что налагает некоторые ограничения и может иметь другие нежелательные последствия. Я начну со случая, когда вы *хотите* сообщить компилятору дополнительную информацию, но нет никакого способа сделать это.

2.2.2. Когда система типов C# 1 недостаточно богата

Есть две распространенные ситуации, когда может понадобиться предоставить вызывающей стороне подробную информацию о методе или, возможно, вынудить вызывающую сторону ограничить то, что она передает в аргументах. Первое подразумевает коллекции, а второе — наследование и переопределение методов или реализацию интерфейсов. Мы исследуем их по очереди.

Коллекции, строго и слабо типизированные

Избежав терминов *строгий* (strong) и *слабый* (weak) в описании системы типов C# вообще, я буду использовать их при разговоре о коллекциях. Они используются в этом контексте почти повсеместно, но с небольшой двусмысленностью. В широком смысле инфраструктура .NET 1.1 имеет три типа коллекций.

- Массивы. Строго типизированы, входят в состав и языка, и среды выполнения.
- Слабо типизированные коллекции пространства имен `System.Collections`.
- Строго типизированные коллекции пространства имен `System.Collections.Specialized`.

Массивы строго типизированы,[10] поэтому во время компиляции вы не можете заставить элемент `string[]` стать, например, `FileStream`. Но массивы ссылочного типа поддерживают также *ковариацию* (covariance), которая обеспечивает неявное преобразование из одного типа массива в другой, если возможно преобразование между типами их элементов. Проверки осуществляются во время выполнения, чтобы предотвратить сохранение ссылки неправильного типа, как показано в листинге 2.3.

Листинг 2.3. Демонстрация ковариации массивов и проверка времени выполнения

```
string[] strings = new string[5];
object[] objects = strings; // #1 Применение ковариантного преобразования
objects[0] = new Button();  // #2 Попытка сохранить ссылку Button
```

[10] По крайней мере, язык позволяет им быть таковыми. Для слабо типизированного доступа к массивам вы все же можете использовать тип `Array`.

Если вы запустите листинг 2.3, то увидите, что строка #2 передает исключение ArrayTypeMismatchException. Дело в том, что преобразование элемента string[] в элемент object[] (строка #1) возвращает исходную ссылку, а strings и objects ссылаются на тот же массив. Сам массив знает, что он строковый, и отклонит попытки сохранить ссылки на нестроковый тип. Ковариация массива иногда полезна, но она осуществляется частично за счет безопасности типов и реализуется во время выполнения, а не во время компиляции.

Давайте сравним это с ситуацией, когда используются слабо типизированные коллекции, такие как ArrayList и Hashtable. Функции API этих коллекций используют тип object и как тип ключей, и как тип значений. Когда вы пишете метод, использующий тип ArrayList, например, во время компиляции нет никакого способа гарантировать, что вызывающая сторона передаст ему список строк. Вы можете задокументировать это, и система безопасности типов среды выполнения претворит это в жизнь, если вы приведете каждый элемент списка к типу string, но во время компиляции вы не получаете безопасности типов. Аналогично, если вы возвращаете список ArrayList, можете указать в документации, что он будет содержать только строки, но вызывающая сторона вынуждена будет полагаться только на вашу честность и применять приведения при обращении к элементам списка.

И наконец, рассмотрим строго типизированные коллекции, такие как StringCollection. Они предоставляют строго типизированные API, поэтому можете быть уверены, что при получении коллекции StringCollection в качестве параметра или возвращаемого значения она будет содержать только строки, и вы не должны приводить элементы при выборке из коллекции. Это кажется идеальным, но есть две проблемы. Во-первых, она реализует интерфейс Ilist. Таким образом, вы можете *попытаться* добавить в нее нестроковые элементы (хотя во время выполнения это потерпит неудачу). Во-вторых, эта коллекция имеет дело только со строками. Существует еще много других специализированных коллекций, но все они отличаются не принципиально. Есть тип CollectionBase, который применяется для создания собственных строго типизированных коллекций, но это означает создание нового типа коллекции для каждого типа элементов, что также не идеально.

Теперь, когда мы увидели проблему с коллекциями, давайте рассмотрим затруднение, которое может произойти при переопределении методов и при реализации интерфейсов. Оно связано с идеей ковариации, которую мы уже видели у массивов.

Нехватка ковариантных типов возвращаемого значения

Интерфейс ICloneable является одним из самых простых интерфейсов инфраструктуры. У него есть один метод, Clone, который должен вернуть копию объекта, для которого вызван метод. Теперь, не принимая во внимание, должно ли это быть глубоким копированием или поверхностным, давайте рассмотрим сигнатуру метода Clone.

```
object Clone()
```

Конечно, это простая сигнатура, но, как я уже упомянул, метод должен вернуть копию объекта, для которого он был вызван. Это означает, что метод должен вернуть объект того же типа или, по крайней мере, совместимого (когда значение изменяется в зависимости от типа). Это имело бы смысл при переопределении метода с сигнатурой, которая обеспечивает более точное описание того, что фактически возвращает метод. Например, в классе Person было бы хорошо иметь возможность реализовать интерфейс ICloneable так.

```
public Person Clone()
```

Это ничего не нарушит — код, ожидающий любой старый объект, все еще будет работать прекрасно. Такая возможность называется *ковариацией типа возвращаемого значения* (return type covariance), но, к сожалению, реализацию интерфейса и переопределение методов она не поддерживает. Чтобы достичь желаемого эффекта, вместо нее, для нормальной работы интерфейсов, должна использоваться *явная реализация интерфейса* (explicit interface implementation).

```
public Person Clone()
{
    [Реализация здесь]
}
object ICloneable.Clone()
{
    return Clone(); // Вызов неинтерфейсного метода
}
```

Любой код, который вызывает метод `Clone()` в выражении со статическим типом `Person`, вызовет верхний метод; если типом выражения будет только `ICloneable`, то будет вызван нижний метод. Это сработает, но выглядит коряво. Зеркальное отображение этой ситуации также осуществляется с параметрами, когда у вас есть интерфейс или виртуальный метод, скажем, с сигнатурой `void Process(string x)`, затем, казалось бы, было логично иметь возможность реализовать или переопределить метод с менее требовательной сигнатурой, такой как `void Process(object x)`. Это называется *контрвариация типа параметра* (parameter type contravariance) и так же не поддерживается, как и ковариация типа возвращаемого значения, при той же обработке для интерфейсов и нормальной перегрузке виртуальных методов. Это, конечно, не звезда балета, но и не полный провал.

Конечно, разработчики языка C# 1 мирились со всеми этими проблемами на протяжении долгого времени, у разработчиков языка Java была подобная ситуация, но намного раньше. Хотя безопасность типов во время компиляции — это прекрасная возможность, вообще, я не могу вспомнить случая, когда люди *фактически* помещали в коллекции элемент неправильного типа. Я вполне могу обходиться и без ковариации с контрвариацией. Но есть такое понятие, как четкий и понятный код, который однозначно выражает то, что вы имеете в виду, без пояснительных текстов. Даже если ошибки не встречаются фактически, применение документированного контракта, согласно которому коллекция *должна* содержать только строки, например, может быть дорогим и ненадежным подходом, по сравнению с изменяемыми коллекциями. Это именно тот вид контракта, который действительно должна применять сама система типов.

Впоследствии мы увидим, что язык C# 2 также не безупречен, но он имеет больше преимуществ. Язык C# 4 содержит еще больше изменений, но даже в этом случае ковариация типа возвращаемого значения и контрвариация параметра отсутствуют.[11]

2.2.3. Резюме по характеристикам системы типов

В этом разделе мы рассмотрели некоторые различия между системами типов и, в частности, их характеристики, относящиеся к языку C# 1.

- Язык C# 1 является статически типизированным — компилятор знает, какие члены позволять вам использовать.
- Язык C# 1 является явным — вы должны заявить тип каждой переменной.

[11] Язык C# 4 вводит ограниченную *обобщенную* ковариацию и контрвариацию, но это не совсем то же самое.

- Язык C# 1 типизированный — вы не можете работать с одним типом, как будто это другой тип, без преобразования.
- Статическая типизация не позволяет одиночной коллекции быть строго типизированным "списком строк" или "списком целых чисел" без дублирования большого количества кода для элементов разных типов.
- Переопределение метода и реализация интерфейса не допускают ковариацию или контрвариацию.

В следующем разделе будет рассмотрен один из самых фундаментальных аспектов системы типов языка C# — различия между структурами и классами.

2.3. Типы значений и ссылочные типы

Вероятно, сложно преувеличить, насколько важна тема этого раздела. Все, что вы делаете в инфраструктуре .NET, будет относиться либо к типу значений, либо к ссылочному типу. И все же забавно, что, возможно, многие занимались разработкой в течение долгого времени, имея лишь смутное представление об этом. Прискорбно, но факт: довольно просто сделать короткое, но неправильное утверждение, которое достаточно близко к истине, чтобы быть вероятным, но достаточно неверно, чтобы ввести в заблуждение, однако придумать краткое, но точное описание относительно сложно.

Этот раздел не о полном нарушении обычного порядка обработки типов, маршалинге между доменами приложения, совместимости с базовым кодом и подобном. Здесь мы кратко рассмотрим простые темы (в применении к языку C# 1), которые крайне важны для понимания более поздних версий C#.

Начнем с выяснения фундаментальных различий между типами значений и ссылочными типами, как в реальном мире, так и в инфраструктуре .NET.

2.3.1. Значения и ссылки в реальном мире

Предположим, что вы читаете кое-что фантастическое и хотите, чтобы ваш друг читал то же. Предположим также, что это документ в открытом домене и вас не обвинят в нарушении авторских прав. Что же сделать, чтобы ваш друг смог прочитать то же, что и вы? Это зависит от того, что вы читаете.

Сначала рассмотрим случай, когда у вас на руках есть реальный бумажный документ. Чтобы предоставить своему другу копию, вы должны ксерокопировать все страницы, а затем передать их ему. Теперь у него есть собственный полный экземпляр документа. В этой ситуации имеем дело с *типом значений* (value type). Вся информация находится непосредственно в ваших руках, вам не нужно идти куда-нибудь еще, чтобы получить ее. Ваш экземпляр, после того как вы сделали копию, полностью независим от вашего друга. Вы можете писать примечания на своих страницах, а его страницы не будут изменены.

Сравните это с ситуацией, когда вы читаете веб-страницу. На сей раз вам достаточно передать своему другу адрес URL веб-страницы. Это режим *ссылочного типа*, а URL выступает в качестве ссылки. Чтобы фактически прочитать документ, вы должны перейти по ссылке, введя URL в своем браузере и запросив загрузку страницы. С другой стороны, если веб-страница изменяется по некоторым причинам (предположим, что это страница Википедии и вы добавили на нее свои примечания), то и вы, и ваш друг увидите эти изменения при следующей загрузке страницы.

В реальном мире отображается принципиальное отличие между типами значений и ссылочными типами в языке C# и инфраструктуре .NET. Большинство типов инфраструктуры .NET является ссылочными типами, и вы, вероятно, создадите *гораздо* больше

ссылочных типов, чем типов значений. Наиболее распространенный пример: классы (объявляемые с использованием ключевого слова `class`) являются ссылочными типами, а структуры (объявляемые с использованием ключевого слова `struct`) — типы значений. Рассмотрим и другие примеры.

- Типы массивов — ссылочные типы, даже если тип элемента — тип значений (таким образом, `int[]` — это ссылочный тип, хотя `int` — тип значений).
- Перечисления (объявляемые с использованием ключевого слова `enum`) являются типами значений.
- Типы делегата (объявляемые с использованием ключевого слова `delegate`) являются ссылочными типами.
- Типы интерфейса (объявляемые с использованием ключевого слова `interface`) — ссылочные типы, но они могут быть реализованы типами значений.

Теперь, когда мы знаем, что такое ссылочные типы и типы значений, рассмотрим некоторые важнейшие детали.

2.3.2. Основные принципы ссылочных типов и типов значений

Ключевая концепция, которую следует уяснить, когда дело доходит до типов значений и ссылочных типов, — это значение конкретного выражения. Чтобы не усложнять изложение, я буду использовать переменные как наиболее распространенный пример выражений, но то же самое относится и к свойствам, вызовам методов, индексаторам и другим выражениям.

Как я упоминал в разделе 2.2.1, с большинством выражений связан статический тип. Результат выражения типа значений — это простое значение. Например, результат выражения 2+3 — это 5. Результат выражения *ссылочного типа* является ссылкой. Это не *объект*, на который указывает ссылка. Так, значение выражения `String.Empty` — не *пустая* строка, а ссылка на пустую строку. В повседневном общении и даже в документации мы обычно не замечаем это различие. Например, мы могли бы написать, что выражение `String.Concat` возвращает "строку, конкатенирующую все параметры". Использование правильной терминологии оказалось бы здесь долгим и многословным, и нет никакой проблемы, пока все понимают, что возвращается только *ссылка*.

Чтобы продемонстрировать это, рассмотрим тип `Point`, который хранит два целочисленных значения `x` и `y`. У него может быть конструктор, получающий два значения. Сам тип `Point` может быть реализован как структура или как класс. Результат выполнения следующих строк кода демонстрирует рис. 2.3.

```
Point p1 = new Point(10, 20);
Point p2 = p1;
```

На рис. 2.3, *слева* показано использование значения, когда `Point` — ссылочный тип (класс), а на рис. 2.3, *справа* представлена ситуация, когда `Point` — тип значений (структура).

В обоих случаях у переменных `p1` и `p2` после присвоения будет одно и то же значение. Но в случае, когда тип `Point` — ссылочный тип, его значение будет ссылкой: переменные `p1` и `p2` ссылаются на тот же объект. Когда `Point` — тип значений, структура `p1` содержит все данные точки, значения `y` и `x`. Присвоение значения структуры `p1` структуре `p2` копирует все эти данные.

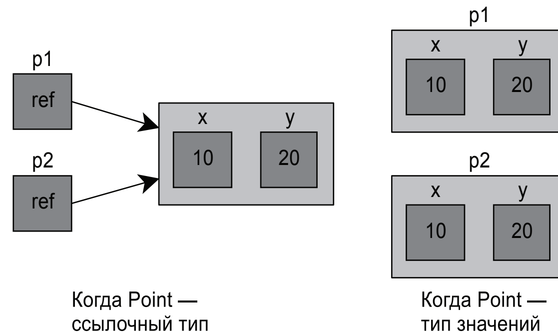

Рис. 2.3. Сравнение типа значений и ссылочного типа в процессе присвоения

Значения переменных хранятся там, где они объявляются. Значения локальных переменных всегда хранятся в стеке[12], а значения переменных экземпляров — там, где хранится сам экземпляр. Экземпляры ссылочного типа (объекты) всегда хранятся в распределяемой памяти, как статические переменные.

Еще одно различие между двумя видами типов в том, что типы значений не могут быть производными от других типов. Значения не нуждаются ни в какой дополнительной информации о типе; это *фактически* значения. Сравните их со ссылочными типами, где каждый объект содержит в начале совокупность данных, идентифицирующих фактический тип объекта, наряду с некоторой другой информацией. Вы никак не можете изменить тип объекта — когда выполняете простое приведение, среда выполнения получает только ссылку, проверяет, ссылается ли она на допустимый объект правильного типа, и возвращает исходную ссылку, если он допустим, или передает исключение — в противном случае. Самой ссылке тип объекта безразличен, поэтому одно и то же значение ссылки применимо для нескольких переменных различных типов. Рассмотрим, например, следующий код.

```
Stream stream = new MemoryStream();
MemoryStream memoryStream = (MemoryStream) stream;
```

Первая строка создает новый объект `MemoryStream` и назначает переменную `stream` ссылкой на этот новый объект. Вторая строка проверяет, ссылается ли значение переменной `stream` на объект типа `MemoryStream` (или типа, производного от него), и устанавливает для переменной `memoryStream` такое же значение.

Как только вы уясните эти простые положения, можете применять их при размышлении о некоторых мифах, которые зачастую складывают о типах значений и ссылочных типах.

2.3.3. Разрушение мифов

Существует много различных мифов, которые, я уверен, почти всегда передаются без преступного намерения и злого умысла, но это тем не менее вредно. В этом разделе рассмотрим самые распространенные мифы и объясним истинную ситуацию.

Миф 1. "Структуры — это облегченные классы"

Этот миф имеет множество форм. Одни полагают, что типы значений не могут или не должны иметь методов или других существенных действий, они должны использоваться

[12] Это верно только для языка C# 1. Позже мы увидим, что в более поздних версиях при определенных ситуациях локальные переменные могут располагаться в распределяемой памяти.

как простые типы передачи для данных, только с открытыми полями или простыми свойствами. Тип `DateTime` — хороший контраргумент этому. Он должен быть типом значений, поскольку является фундаментальной единицей, такой как число или символ, и выполнять вычисления с его значением. С другой стороны, типы передачи данных зачастую бывают ссылочными. Так или иначе, решение должно быть принято на основании желаемой семантики типа значений или ссылочного типа, а не простоты типа.

Другие полагают, что типы значений "легче" ссылочных типов, с точки зрения производительности. Правда в том, что *в некоторых случаях* типы значений более производительны, они не требуют сбора "мусора", если они не упакованы, не имеют дополнительных затрат на идентификацию типа и не требуют обращения к значению, например. Но в других случаях ссылочные типы более производительны — для передачи параметров, присвоения значения переменным, возвращения значения и подобных операций требуется скопировать только 4 или 8 байт (в зависимости от использования 32- или 64-битовой среды CLR), вместо копирования *всех* данных. Вообразите, если бы тип `ArrayList` был "чистым" типом значений и передавал используемому методу путем копирования все свои данные! Практически во всех случаях производительность определяется не этим видом решения. Узкие места почти никогда не появляются там, где вы предполагаете, и прежде чем принять проектное решение на основании производительности, вы должны проанализировать все возможности.

Стоит также обратить внимание на то, что комбинация этих двух верований также не работает: не имеет значения, сколько методов имеет тип (является ли он классом или структурой), на объем памяти, занимаемый под экземпляр, это не влияет. (Есть разница с точки зрения памяти, занимаемой для самого кода, но она выделяется лишь однажды, а не для каждого экземпляра.)

Миф 2. "Ссылочные типы находятся в распределяемой памяти, а типы значений — в стеке"

Как правило, только ленивый не повторяет этого. Первая часть правильна: экземпляр ссылочного типа всегда создается в распределяемой памяти. Со второй частью есть проблемы. Как я уже упоминал, значение переменной располагается там, где оно объявляется, поэтому если у вас есть класс с переменными экземпляра типа `int`, то значения этих переменных для любого объекта данного класса всегда будут там, где и остальная часть данных объекта, в распределяемой памяти. В стеке находятся только локальные переменные (переменные, объявленные в пределах методов) и параметры методов. В языке C# 2 и выше даже некоторые локальные переменные не находятся в стеке, как мы увидим при рассмотрении анонимных методов в главе 5.

Действительно ли эти концепции уместны теперь

Сейчас является довольно спорным, что, когда вы пишете управляемый код, должны позволить среде выполнения позаботиться о том, как лучше всего использовать память. Действительно, спецификация языка не дает гарантий того, где и что будет располагаться; будущая среда выполнения может оказаться в состоянии создавать некоторые объекты в стеке, если ей будет известно, что это можно сделать, или компилятор C# может создать код, который почти не использует стек вообще.

Следующий миф — скорее, проблема терминологии.

Миф 3. "По умолчанию объекты C# передаются по ссылке"

Это, вероятно, наиболее широко распространенный миф. Люди, которые утверждают это, зачастую (хоть и не всегда) знают, как фактически ведет себя язык C#, но не знают, что действительно означает термин "передача по ссылке" (pass by reference). К сожалению, это сомнительно для людей, которые *действительно* знают, что это означает. Фор-

мальное определение *передачи по ссылке* относительно сложно, оно подразумевает такую терминологию, как *l-значение* (l-value) и подобное, но важно то, что если вы передаете переменную по ссылке, то вызванный метод может изменить *значение переменной вызывающей стороны* при изменении значения ее параметра. Теперь вспомните, что содержимым переменной ссылочного типа является *ссылка*, а не сам объект непосредственно. Вы можете изменить *содержимое* объекта, на который указывает параметр, без передачи самого параметра по ссылке. Например, следующий метод изменяет содержимое объекта `StringBuilder`, но выражение вызывающей стороны все еще остается тем же объектом, что и прежде.

```
void AppendHello(StringBuilder builder)
{
    builder.Append("hello");
}
```

При вызове этого метода значение параметра (ссылка на объект `StringBuilder`) передается по значению. Например, если я должен изменить в методе значение переменной `builder` при помощи оператора `builder = null`, то *это* изменение, вопреки мифу, не будет замечено вызывающей стороной.

Интересно заметить, что не только часть "по ссылке" выражения является мифом, но и часть "объекты передаются". Сами объекты никуда не передаются, ни по ссылке, ни по значению. Когда используется ссылочный тип, либо переменная передается по ссылке, либо значение аргумента (ссылка) передается по значению. Кроме всего прочего, это отвечает на вопрос о том, что происходит, когда в качестве аргумента используется значение `null`, — если бы передавались объекты, возникла бы проблема, поскольку не будет объекта для передачи! Вместо этого ссылка `null` передается как значение, точно так же как любая другая ссылка.

Если это краткое объяснение оставило у вас вопросы, вы можете обратиться к статье на моем веб-сайте C# (`http://mng.bz/otVt`), где тема раскрыта более подробно.

Эти мифы не единственные. Упаковка и распаковка вносят свою долю недоразумений, которые я пытаюсь устранить.

2.3.4. Упаковка и распаковка

Иногда тип значений не подходит. Нужна ссылка. Тому есть немало причин, и к счастью, язык C#, а также инфраструктура .NET предоставляют механизм, называемый *упаковкой* (boxing), который позволяет создать объект из переменной, имеющей тип значений, и использовать ссылку на этот новый объект. Прежде чем перейти непосредственно к примеру, рассмотрим два важных факта.

- Значение переменной ссылочного типа всегда является ссылкой.
- Значение переменной типа значений всегда является значением этого типа.

С учетом этих двух фактов, следующие три строки кода, казалось бы, не имеют смысла.

```
int i = 5;
object o = i;
int j = (int) o;
```

Имеется две переменные: `i` — переменная типа значений и `o` — переменная ссылочного типа. Какой смысл присваивать значение переменной `i` переменной `o`? Значение переменной `o` должно быть ссылкой, а число 5 ссылкой не является, это целочисленное значение. То, что фактически происходит, и является упаковкой — среда выполнения создает объект (в распределяемой памяти; это вполне нормальный объект), который со-

держит значение (5). Теперь переменная `o` содержит ссылку на этот новый объект. Значение в объекте — это *копия* исходного значения, последующее изменение значения переменной `i` не изменит упакованного значения.

Третья строка выполняет обратную операцию — *распаковку* (unboxing). Мы должны указать компилятору, какой тип распаковывать из объекта, и если мы используем неправильный тип (если, например, упакован тип `uint` или `long` либо значение не упаковано вообще), передается исключение `InvalidCastException`. При распаковке также создается копия упакованного значения: после присвоения нет никакой дальнейшей связи между переменной `j` и объектом.

Вот и вся премудрость упаковки и распаковки. Осталось только узнать, когда происходят упаковка и распаковка. Распаковка обычно очевидна, поскольку приведение присутствует в коде. Упаковка может быть немного сложнее. Мы видели простую версию, но упаковка может также происходить при вызове методов `ToString`, `Equals` или `GetHashCode` для значения типа, который не переопределяет их,[13] или если вы используете значение как интерфейс выражения, присвоив его переменной, типом которой является тип интерфейса, или передав его как значение для параметра с типом интерфейса. Например, оператор `IComparable x = 5;` приведет к упаковке числа 5.

Упаковка и распаковка могут привести к потере производительности. Одна операция упаковки и распаковки погоды не сделает, но если вы выполняете их сотнями тысяч, то, кроме потерь, на сами операции вы также получите *множество* объектов, которое создадут массу работы сборщику "мусора". Как правило, такая потеря производительности не проблематична, но о ней стоит знать, если это критически важно для результата.

2.3.5. Резюме по типам значений и ссылочным типам

В этом разделе мы рассмотрели различия между типами значений и ссылочными типами, а также развенчали некоторые мифы о них. Вот ключевые пункты.

- Значение выражения ссылочного типа (например, переменная) является ссылкой, а не объектом.
- Ссылки подобны URL — это небольшие фрагменты данных, которые позволяют обращаться к реальной информации.
- Значение выражения, имеющего типа значений — это фактические данные.
- Иногда типы значений эффективнее ссылочных типов, а иногда наоборот.
- Объекты ссылочного типа всегда находятся в распределяемой памяти, а значения типа значений могут быть либо в стеке, либо в распределяемой памяти, в зависимости от контекста.
- Когда ссылочный тип используется как параметр метода, по умолчанию аргумент передается по *значению*, но само значение — ссылка.
- Значения переменных, имеющих типа значений, упаковываются, когда необходим режим ссылочного типа; распаковка — это обратный процесс.

Теперь, когда рассмотрены все элементы языка C# 1, пришло время выяснить, как каждая из возможностей совершенствуется в более поздних версиях.

---

[13] Упаковка будет происходить *всегда*, когда вы вызовете метод `GetType()` для переменной типа значений, поскольку он не может быть переопределен. Вы должны уже знать точный тип, если имеете дело с распакованной формой, таким образом, вы можете использовать вместо него только оператор `typeof`.

2.4. Вне C# 1: новые возможности на солидной базе

Все три темы, затронутые в этой главе, жизненно важны для всех версий языка C#. Почти все новые средства затрагивают, по крайней мере, одну из них. Прежде чем закончить главу, давайте рассмотрим, как новые средства связаны со старыми. Я не собираюсь вникать в подробности (издатель не хотел бы, чтобы один раздел занял 600 страниц), но полезно иметь представление о том, что происходит, прежде чем мы доберемся до основных элементов. Мы рассмотрим их в том же порядке, как и ранее, начиная с делегатов.

2.4.1. Средства, связанные с делегатами

В языке C# 2 улучшены делегаты всех видов, а их обработка в языке C# 3 получила дальнейшее развитие. Большинство средств — это не нововведение CLR, а хитрые уловки компилятора, улучшающие работу делегатов в пределах языка. Изменения затрагивают не только синтаксис, который мы *можем* использовать, но и внешний вид идиоматического кода C#. Со временем язык C# получил более функциональный подход.

Когда дело доходит до создания экземпляра делегата, синтаксис языка C# 1 выглядит довольно неуклюже. С одной стороны, даже если необходимо сделать что-то простое, вы вынуждены писать отдельный метод, специализированный на этой задаче, чтобы создать экземпляр делегата. Язык C# 2 устраняет этот недостаток при помощи анонимных методов и вводит упрощенный синтаксис для случаев, когда вы все еще хотите использовать нормальный метод для предоставления действия делегату. Вы можете также создать экземпляры делегата, используя методы с *совместимыми* сигнатурами, — сигнатура метода больше не должна точно совпадать с объявлением делегата.

Все эти усовершенствования демонстрирует следующий листинг.

Листинг 2.4. Усовершенствования в создании экземпляра делегата (C# 2)

```
static void HandleDemoEvent(object sender, EventArgs e)
{
    Console.WriteLine ("Handled by HandleDemoEvent");
}
...
EventHandler handler;
handler = new EventHandler(HandleDemoEvent) // #1 Определяет тип делегата
                                            // и метод
handler(null, EventArgs.Empty);

handler = HandleDemoEvent;                  // #2 Неявное преобразование
handler(null, EventArgs.Empty);             // в экземпляр делегата

handler = delegate(object sender, EventArgs e) // #3 Определяет действие
{                                              // при помощи
    Console.WriteLine ("Handled anonymously"); // анонимного
};                                             // метода
handler(null, EventArgs.Empty);

handler = delegate                                   // #4 Использование
{                                                    // сокращения
    Console.WriteLine ("Handled anonymously again"); // анонимного
};                                                   // метода
handler(null, EventArgs.Empty);

MouseEventHandler mouseHandler = HandleDemoEvent;     // #5 Использование
mouseHandler(null, new MouseEventArgs(MouseButtons.None, // контрвариации
                                  0, 0, 0, 0));       // делегата
```

Первой частью основного кода #1 является код C# 1, сохраненный только для сравнения. Все остальные делегаты используются новыми средствами языка C# 2. Преобразования группы методов #2 делают код подписки на события более читабельны; такие строки, как `saveButton.Click += SaveDocument;`, понятны без дополнительных объяснений. Синтаксис анонимного метода #3 громоздкий, но действительно позволяет действию оставаться в точке создания, вместо того чтобы располагаться в другом методе, который нужно найти, чтобы посмотреть и понять, что происходит. При использовании анонимных методов #4 доступно сокращение, но эта форма может использоваться только тогда, когда вы не нуждаетесь в параметрах. У анонимных методов есть также другие преимущества, но о них позже.

Последний экземпляр делегата создается (строки #5) как экземпляр `MouseEventHandler`, а не просто `EventHandler`, но метод `HandleDemoEvent` все еще может использоваться благодаря *контрвариации*, которая определяет совместимость параметра. *Ковариация* определяет совместимость типа возвращаемого значения. Более подробная информация по этой теме приведена в главе 5. Обработчики событий — это, вероятно, наибольшее преимущество. Все типы делегатов от Microsoft, используемые в событиях, следуют одинаковому соглашению, что имеет определенный смысл. В языке C# 1 не имело значения, выглядели ли два разных обработчика событий одинаково, у вас должен был быть метод с *точно* соответствующей сигнатурой, чтобы создать экземпляр делегата. В языке C# 2 вы можете использовать тот же метод для обработки множества разных видов событий, особенно если цель метода — совершенно независимое событие, такое как регистрация.

Язык C# 3 предоставляет специальный синтаксис для создания экземпляров типов делегата с использованием *лямбда-выражений*. Чтобы продемонстрировать их, мы будем использовать новый тип делегата. Обобщенные типы делегатов стали доступны в результате того, что CLR получила обобщения в .NET 2.0, они используются во многих вызовах функций API в обобщенных коллекциях. Но инфраструктура .NET 3.5 делает следующий шаг, представляя группу обобщенных типов делегата по имени `Func`, получающих множество параметров определенных типов и возвращающих значение другого определенного типа. Следующий листинг представляет пример использования типа делегата `Func`, а также лямбда-выражений.

Листинг 2.5. Лямбда-выражения, подобные улучшенным анонимным методам

```
Func<int,int,string> func = (x, y) => (x * y).ToString();
Console.WriteLine(func(5, 20));
```

`Func<int,int,string>` — это тип делегата, получающий два целых числа и возвращающий строку. Лямбда-выражение в этом листинге определяет, что экземпляр делегата (содержащийся в `func`) должен перемножить эти два целых числа и вызвать метод `ToString()`. Синтаксис намного проще и понятнее, чем у анонимных методов, и есть еще преимущества с точки зрения объема выведения типов, выполняемого компилятором самостоятельно. Лямбда-выражения крайне важны для LINQ, и вы должны быть готовы сделать их основной частью вашего языкового инструментария. Они не ограничиваются работой с LINQ, хотя там, где в языке C# 2 используются анонимные методы, в языке C# 3 может использоваться лямбда-выражение, что почти всегда приводит к более короткому коду. Таким образом, с делегатами связаны следующие новые средства.

- Обобщения (обобщенные типы делегата) — C# 2
- Выражения для создания экземпляра делегата — C# 2
- Анонимные методы — C# 2

- Ковариация и контрвариация делегата — C# 2
- Лямбда-выражение — C# 3

Кроме того, язык C# 4 обеспечивает *обобщенную* ковариацию и контрвариацию для делегатов, как мы только что видели. Действительно, обобщенные типы формируют одно из принципиальных расширений системы типов, которое рассмотрим впоследствии.

2.4.2. Средства, связанные с системой типов

Главным нововведением языка C# 2 в области системы типов являются обобщения. Это в значительной степени решает проблемы, которые я упомянул в разделе 2.2.2, посвященном строго типизированным коллекциям, хотя общие типы полезны также во многих других ситуациях. Это изящное средство решает реальную проблему, и, несмотря на несколько шероховатостей, в целом оно работает хорошо. Мы уже видели примеры этого и полностью рассмотрим в следующей главе, поэтому я не буду излагать здесь много деталей. Это небольшая отсрочка, хотя обобщения составляют, вероятно, самую важную возможность языка C# 2 в области системы типов, и вы будете встречать их повсюду в этой книге.

Язык C# 2 не занимается проблемами ковариации типов возвращаемого значения и контрвариации параметров, чтобы переопределить члены или реализовать интерфейсы. Но в определенных случаях это *действительно* улучшает ситуацию при создании экземпляра делегата, как мы видели в разделе 2.4.1.

Язык C# 3 вводит в систему типов множество новых концепций, в частности *анонимные типы*, *неявно типизированные локальные переменные* и *методы расширения*. Сами анонимные типы, главным образом, существуют ради LINQ, где полезна возможность эффективно создать тип передачи данных с набором свойств только для чтения, без необходимости фактически писать код для них. Но ничего не мешает использовать их вне LINQ, просто так проще для демонстрации. Листинг 2.6 демонстрирует оба средства в действии.

Листинг 2.6. Демонстрация анонимных типов и неявной типизации

```
var jon = new { Name = "Jon", Age = 31 };
var tom = new { Name = "Tom", Age = 4 };
Console.WriteLine ("{0} is {1}", jon.Name, jon.Age);
Console.WriteLine ("{0} is {1}", tom.Name, tom.Age);
```

Первые две строки демонстрируют неявную типизацию (использование переменной `var`) и инициализаторы анонимных объектов (часть `new {...}`), которые создают экземпляры анонимных типов.

На данном этапе, прежде чем переходить к подробностям, имеет смысл обратить внимание на то, что прежде вызывало у людей напрасные волнения. Прежде всего, язык C# 3 все еще является статически типизированным. Компилятор C# объявил, что `jon` и `tom` имеют специфический тип. Как обычно, когда мы используем свойства объектов, это нормальные свойства, никакого динамического поиска. Это именно то, что мы (как авторы исходного кода) не могли указать компилятору, какой тип использовать в объявлении переменной, поскольку компилятор сам создаст тип. Свойства также являются статически типизированными, здесь свойство `Age` имеет тип `int`, а свойство `Name` — тип `string`.

Кроме того, здесь мы не создали два разных анонимных типа. Переменные `jon` и `tom` имеют тот же тип, поскольку компилятор использует имена свойств, типы и порядок, чтобы решить, что он может создать только один тип и использовать его для обоих операторов. Это делается по каждой сборке, а способность присвоить значение одной переменной другой (например, `jon = tom;` вполне допустимо в предыдущем коде) и подобные операции существенно упрощают жизнь.

Методы расширения также предназначены для LINQ, но могут быть полезны и вне его. Вспомните, когда у типа инфраструктуры не было определенного метода и вы должны были писать статический вспомогательный метод, чтобы реализовать его. Например, чтобы создать новую строку, обратную существующей, вы могли бы написать статический метод `StringUtil.Reverse`. Метод расширения, фактически, позволяет вам вызвать этот статический метод, как будто он существовал непосредственно в типе `string`. Таким образом, вы можете написать следующее.

```
string x = "dlrow olleH".Reverse();
```

Методы расширения также позволяют добавлять методы с реализациями в интерфейсы — и это то, на что полагается LINQ, позволяя вызывать все виды методов интерфейса `IEnumerable<T>`, которые ранее никогда не существовали.

У языка C# 4 есть два средства, связанных с системой типов. Относительно малозначительная возможность — ковариация и контрвариация для обобщенных делегатов и интерфейсов. Это уже было в среде CLR, начиная с выхода инфраструктуры .NET 2.0, но только с введением C# 4 (и обновления обобщенных типов в BCL) это стало возможным для использования разработчиками C#. Наиболее существенная возможность, хотя многим разработчикам она может никогда и не потребоваться, — это динамические типы C#.

Во введении я упоминал статическую типизацию, когда пытался использовать свойство `Length` массива и строки через ту же переменную. А вот в языке C# 4 это работает. Листинг 2.7 демонстрирует тот же код, за исключением объявления переменной, и он вполне допустим в языке C# 4.

Листинг 2.7. Динамическая типизация в C# 4

```
dynamic o = "hello";
Console.WriteLine(o.Length);
o = new string[] {"hi", "there"};
Console.WriteLine(o.Length);
```

Объявление переменной `o`,как имеющей статический тип `dynamic` (да, вы прочитали правильно), заставляет компилятор делать почти все с переменной `o` по-другому, оставляя все связанные решения (такие, как смысл слова `Length`) до времени выполнения.

Безусловно, мы собираемся рассмотреть динамическую типизацию достаточно глубоко, но я хочу подчеркнуть сейчас, что язык C# 4 все еще остается по большей части статически типизированным. Если вы не используете тип `dynamic` (который действует как статический тип, обозначающий динамическое значение), все работает точно так же, как прежде. Большинство разработчиков C# нуждаются в динамической типизации довольно редко, и вполне могут игнорировать ее. Когда динамическая типизация нужна, она может быть без проблем применена и, конечно, позволит задействовать код, написанный на динамических языках, запустив *исполняющую среду динамического языка* (Dynamic Language Runtime — DLR). Я только не советовал бы вам с самого начала использовать язык C# как динамический. Если это то, что вам нужно, используйте язык IronPython или что-то подобное; у языков, которые с самого начала разрабатываются для поддержки динамической типизации, вероятно, будет меньше неожиданных “глюков”.

Вот краткий перечень этих средств.

- Общие типы — C# 2
- Ограниченная ковариация и контрвариация делегата — C# 2
- Анонимные типы — C# 3
- Неявная типизация — C# 3

- Методы расширения — C# 3
- Ограниченная обобщенная ковариация и контрвариация — C# 4
- Динамическая типизация — C# 4

После рассмотрения в общем такого широкого разнообразия средств системы типов, давайте обратим внимание на средства, добавленные в одну конкретную часть типов инфраструктуры .NET, — типы значений.

2.4.3. Средства, связанные с типами значений

Здесь обсудим два средства, введенных в язык C# 2. Первое относится к обобщениям, в частности к коллекциям. Наиболее частая жалоба на использование типов значений в коллекциях .NET 1.1 заключалась в том, что в связи со всеми "универсальными" API, определяемыми в терминах типа `object`, каждая операция, которая добавляла значение структуры к коллекции, приводила к его упаковке, а при возвращении — к распаковке. Хотя при отдельном вызове цена упаковки невелика, она может привести к существенной потере производительности, когда применяется к коллекциям с частыми обращениями. В связи с дополнительными затратами на поддержку объектов, увеличивается также расход памяти. Обобщенные типы устраняют проблемы и скорости, и памяти за счет использования *реального* типа, а не только универсального объекта. Например, было бы безумием читать файл и сохранять каждый его байт как элемент списка `ArrayList` в инфраструктуре .NET 1.1, а в инфраструктуре .NET 2.0 это вполне можно сделать со списком `List<byte>`.

Следующее средство решает еще одну проблему, особенно когда речь идет о базах данных: вы не можете присвоить значение `null` переменной, имеющей тип значений. Нет такой концепции, как значение `null` типа `int`, например, даже при том, что целочисленное поле *базы данных* вполне может быть пустым. С этой точки зрения может быть крайне затруднительно моделировать таблицу базы данных в пределах статически типизированного класса без тех или иных уродств. Типы, допускающие значения `null`, являются частью инфраструктуры .NET 2.0, и язык C# 2 включает дополнительный синтаксис, чтобы сделать удобным их применение. В листинге 2.8 показан небольшой пример этого синтаксиса.

Листинг 2.8. Демонстрация разнообразия преимуществ типа, допускающего значения `null`

```
int? x = null; // Объявляет, устанавливает переменную,
               // допускающую значения null
x=5;

if (x != null) // Проверка присутствия "реального" значения
{
    int y = x.Value; // Получение "реального" значения
    Console.WriteLine(y);
}
int z = x ?? 10; // Использование оператора ??
```

Этот листинг демонстрирует разнообразные возможности типов, допускающих значения `null` и сокращения, которые язык C# предусматривает для работы с ними. Мы найдем время для подробного рассмотрения каждой возможности в главе 4, но важнее всего то, что это значительно проще и понятнее, чем при любом используемом в прошлом альтернативном подходе.

На сей раз список дополнений меньше, но это важные средства с точки зрения производительности и элегантности выражения.

- Обобщения — C# 2
- Типы, допускающие значения `null`, — C# 2

2.5. Резюме

Эта глава, в основном, посвящена описанию возможностей C# 1. Цель главы состоит не в том, чтобы раскрыть эту тему полностью на нескольких страницах, просто я предпочитаю описывать более поздние средства, не волнуясь о фундаменте, на котором основано мое изложение.

Все затронутые здесь темы являются основой C# и .NET, но в обсуждениях сообщества я видел много недоразумений, связанных с ними. Хоть эта глава и не достигла большой глубины по всем пунктам, она, надеюсь, прояснит некоторый беспорядок, который мог бы возникнуть в остальной части книги и затруднить ее понимание.

Все три основные темы, кратко затронутые в этой главе, были существенно усовершенствованы, по сравнению с версией C# 1. В частности, обобщения влияют почти на каждую область, которую мы рассматривали в этой главе; это, вероятно, наиболее широко используемая возможность в языке C# 2. Теперь можем приступать к рассмотрению следующей главы.