8 Основные сведения о значениях и ссылках

Прочитав эту главу, вы научитесь:

- объяснять разницу между типом значений и типом ссылок;
- изменять способ передачи аргументов в качестве параметров методов с помощью ключевых слов `ref` и `out`;
- превращать значение в ссылку, используя упаковку (boxing);
- превращать ссылку обратно в значение, используя распаковку (unboxing) и приведение типов (casting).

В главе 7 «Создание классов и объектов и управление ими» было показано, как объявляются свои собственные классы и как создаются объекты с помощью использования ключевого слова `new`. Там же было показано, как с помощью конструктора инициализируется объект. В данной главе вы узнаете, чем характеристики простых типов, таких как `int`, `double` и `char`, отличаются от характеристик типов классов.

Копирование типов значений переменных и классов

Большинство элементарных типов, встроенных в C#, например `int`, `float`, `double` и `char` (но не `string` — по причинам, которые вскоре будут рассмотрены), обобщенно называются типами значений. У этих типов фиксированный размер, и когда вы объявляете переменную как тип значения, компилятор создает код, занимающий блок памяти, достаточный по размеру для хранения соответствующего значения. Например, объявление `int`-переменной заставляет компилятор выделить для хранения целочисленного значения 4 байта памяти (32 бита).

Инструкция, присваивающая значение (например, 42) `int`-переменной, приводит к тому, что значение копируется в этот блок памяти.

Такие типы классов, как `Circle`, рассмотренный в главе 7, обрабатываются по-другому. Когда объявляется `Circle`-переменная, компилятор не создает код, распределяющий блок памяти, размер которого достаточен для хранения значения типа `Circle`, а просто отводит небольшой участок памяти, где потенциально может содержаться адрес другого блока памяти (или ссылка на него), в котором содержится `Circle`-значение (адрес, указывающий место элемента в памяти). Память под реальный `Circle`-объект выделяется, только когда для создания объекта используется ключевое слово `new`. Класс является примером ссылочного типа. В ссылочных типах содержатся ссылки на блоки памяти. Для создания эффективных программ на C#, которые в полной мере используют среду Microsoft .NET Framework, нужно разобраться в том, чем типы значений отличаются от ссылочных типов.

ПРИМЕЧАНИЕ Тип string в C# фактически является классом. Дело в том, что для строки не существует стандартного размера (различные строки могут содержать разное количество символов) и динамическое выделение памяти под строку в ходе выполнения программы работает гораздо эффективнее статического выделения в ходе компиляции. Описание ссылочных типов, таких как классы, приведенное в этой главе, применимо также к типу string. Фактически ключевое слово string в C# является псевдонимом класса System.String.

Рассмотрим ситуацию объявления переменной по имени `i` с типом значения `int` и присваивания ей значения 42. Если объявить еще одну переменную по имени `copyi` с типом значения `int`, а затем присвоить переменную `i` переменной `copyi`, то `copyi` будет содержать точно такое же значение, что и переменная `i` (42). Но даже при том что `copyi` и `i` содержат одно и то же значение, это значение 42 содержат два блока памяти: один для `i`, другой для `copyi`. Если вы измените значение `i`, значение `copyi` не изменится. Давайте посмотрим, как выглядит соответствующий код:

```
int i = 42;    // объявление и инициализация i
int copyi = i; /* copyi содержит копию данных, имеющихся в i:
                  как i, так и copyi содержат значение 42 */
i++;           /* увеличение i на единицу не влияет на copyi;
                  i теперь содержит 43, а copyi — по-прежнему 42 */
```

Эффект, получаемый от объявления переменной `c` в качестве типа класса, такого как `Circle`, совершенно иной. При объявлении `c` в качестве `Circle`-переменной `c` может ссылаться на `Circle`-объект; фактическим значением, содержащимся в `c`, является адрес `Circle`-объекта в памяти. Если объявить еще одну переменную по имени `refc` (также в качестве `Circle`-переменной) и присвоить ей значение переменной `c`, в `refc` будет содержаться копия точно такого же адреса, что и в `c`.

Иными словами, будет существовать только один `Circle`-объект и теперь на него будут ссылаться обе переменные, как `refc`, так и `c`. Соответствующий пример кода выглядит следующим образом:

```
Circle c = new Circle(42);
Circle refc = c;
```

Оба примера показаны на рис. 8.1. Знак «эт» (`@`) в `Circle`-объектах обозначает ссылку, в которой содержится адрес в памяти.

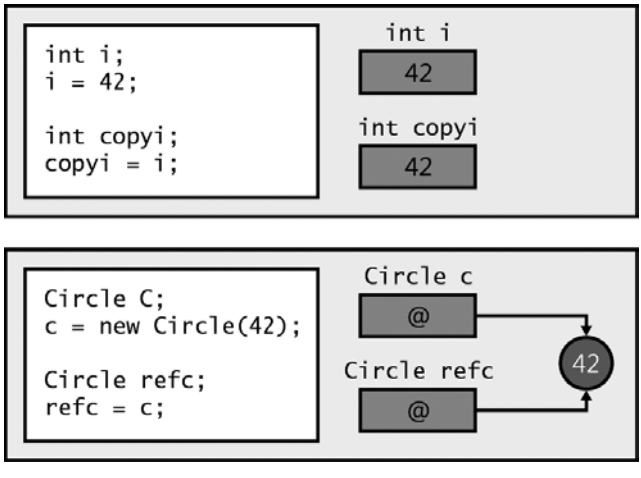

Рис. 8.1

Это различие играет весьма важную роль. В частности, оно означает, что поведение параметров метода зависит от того, к каким типам они относятся: типам значений или ссылочным типам. Это различие будет изучено при выполнении следующего упражнения.

КОПИРОВАНИЕ ССЫЛОЧНЫХ ТИПОВ И ЗАКРЫТОСТЬ ДАННЫХ

Если нужно скопировать содержимое Circle-объекта по имени c в другой Circle-объект по имени refc, то вместо простого копирования ссылки нужно сделать так, чтобы объект refc ссылался на новый экземпляр класса Circle, а затем скопировать данные из c в refc, причем отдельно скопировать каждое поле:

```
Circle refc = new Circle();
refc.radius = c.radius; // Не пытайтесь это делать
```

Но если какие-либо элементы класса Circle являются закрытыми (такими, как поле radius), то скопировать их будет невозможно. В качестве альтернативы можно сделать данные в закрытых полях доступными, выставив их в качестве

свойств, а затем воспользоваться этими свойствами для чтения данных из объекта по имени c в объект по имени refc. Как это делается, будет показано в главе 15 «Реализация свойств для доступа к полям».

Кроме этого, класс может предоставить метод Clone, возвращающий еще один экземпляр того же класса, но наполненный точно такими же данными. Метод Clone будет иметь доступ к закрытым данным внутри объекта и может копировать эти данные непосредственно в другой экземпляр того же класса. Например, метод Clone для класса Circle может быть определен следующим образом:

```
class Circle
{
    private int radius;
    // Конструкторы и другие методы опущены
    ...
    public Circle Clone()
    {
        // Создание нового Circle-объекта
        Circle clone = new Circle();
        // копирование закрытых данных из этого объекта в клон
        clone.radius = this.radius;
        // Возвращение нового Circle-объекта, содержащего скопированные
           данные
        return clone;
    }
}
```

Этот подход не вызывает затруднений, если все закрытые данные состоят из значений, но если одно или несколько полей сами по себе являются ссылочными типами (например, класс Circle может быть расширен, чтобы в нем мог содержаться Point-объект из главы 7, указывающий позицию окружности Circle на графическом изображении), этим ссылочным объектам также необходимо предоставить метод Clone, в противном случае метод Clone класса Circle просто скопирует ссылку на эти поля. Этот процесс известен как создание углубленной копии. Иной подход, при котором метод Clone просто копирует ссылки, известен как создание поверхностной копии.

Предыдущий пример кода вызывает также весьма интересный вопрос: насколько же закрыты закрытые данные? Ранее вы уже видели, что ключевое слово private делает поле или метод недоступными за пределами класса. Но это не значит, что к нему можно получить доступ из одного-единственного объекта. Если создать два объекта одного и того же класса, каждый из них может обращаться к данным, закрытым внутри кода для этого класса. Как бы странно это ни звучало, но факт остается фактом: работа таких методов, как Clone, зависит от этого свойства. Инструкция clone.radius = this.radius; работает только потому, что закрытое поле radius в объекте clone доступно из текущего экземпляра класса Circle. Следовательно, закрытость (private) означает «собственность класса», а не «собственность объекта». Но не нужно путать private со static. Если просто объявить поле закрытым (private), то каждый экземпляр класса получит свои собственные данные. Если поле объявлено статическим (static), то каждый экземпляр класса использует одни и те же данные совместно с другими экземплярами этого же класса.

Использование параметров-значений и параметров-ссылок

Откройте в Microsoft Visual Studio 2015 проект Parameters, который находится в папке \Microsoft Press\VCSBS\Chapter 8\Parameters вашей папки документов. Проект содержит три файла с кодом на C#: Pass.cs, Program.cs и WrappedInt.cs.

Выведите в окно редактора файл Pass.cs. В нем определен класс по имени `Pass`, который пока что не содержит ничего, кроме комментария `// TODO:`.

СОВЕТ Не забудьте, что для обнаружения всех имеющихся в решении комментариев TODO можно воспользоваться окном Список задач.

Вместо комментария `// TODO:` добавьте к классу `Pass` открытый статический метод по имени `Value`. Этот метод должен получить один `int`-параметр (с типом значения) по имени `param` и иметь возвращаемый тип `void`. Тело метода `Value`, выделенное в следующем примере кода жирным шрифтом, будет просто присваивать значение 42 переменной `param`:

```
namespace Parameters
{
    class Pass
    {
        public static void Value(int param)
        {
            param = 42;
        }
    }
}
```

ПРИМЕЧАНИЕ Чтобы упражнение не усложнилось, этот метод определен с использованием ключевого слова static. Метод Value можно вызвать непосредственно в отношении класса Pass без предварительного создания нового Pass-объекта. Принципы, проиллюстрированные в этом упражнении, применяются точно так же и к методам экземпляров.

Выведите в окно редактора файл Program.cs, а затем найдите метод `doWork` класса `Program`. Метод `doWork` вызывается методом `Main`, когда программа начинает работу. Как говорилось в главе 7, вызов метода заключен в блок `try`, за которым следует обработчик исключения.

Добавьте к методу `doWork` четыре инструкции, выполняющие следующие задачи.

- Объявление локальной `int`-переменной по имени `i` и ее инициализация нулевым значением.
- Запись значения переменной `i` в консоль с помощью метода `Console.WriteLine`.

- Вызов `Pass.Value` с передачей `i` в качестве аргумента.
- Повторная запись значения `i` в консоль.

Благодаря вызовам `Console.WriteLine` до и после вызова `Pass.Value` вы сможете увидеть, действительно ли вызов `Pass.Value` изменяет значение переменной `i`. Окончательно метод `doWork` должен приобрести следующий вид:

```
static void doWork()
{
    int i = 0;
    Console.WriteLine(i);
    Pass.Value(i);
    Console.WriteLine(i);
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запуск программы. Убедитесь, что значение 0 записано в консоль дважды. Инструкция присваивания внутри метода `Pass.Value`, обновляющая параметр и устанавливающая для него значение 42, использует копию переданного аргумента, а на исходный аргумент `i` это абсолютно не влияет.

Нажмите Ввод и закройте приложение.

А теперь посмотрите, что произойдет, когда передается `int`-параметр, заключенный в классе.

Выведите в окно редактора файл WrappedInt.cs. В нем содержится класс `WrappedInt`, в котором нет ничего, кроме комментария `// TODO:`.

Добавьте к классу `WrappedInt` выделенное в следующем примере кода жирным шрифтом открытое поле экземпляра с именем `Number`, имеющее тип `int`:

```
namespace Parameters
{
    class WrappedInt
    {
        public int Number;
    }
}
```

Выведите в окно редактора файл Pass.cs. Добавьте к классу `Pass` открытый статический метод по имени `Reference`. Этот метод должен принимать единственный `WrappedInt`-параметр по имени `param` и иметь возвращаемый тип `void`. В теле метода `Reference` значение 42 должно присваиваться `param.Number`:

```
public static void Reference(WrappedInt param)
{
    param.Number = 42;
}
```

Выведите в окно редактора файл Program.cs. Закомментируйте существующий код в методе doWork и добавьте четыре инструкции, выполняющие следующие задачи.

- Объявление локальной WrappedInt-переменной по имени wi и инициализация ее новым WrappedInt-объектом путем вызова пассивного конструктора.
- Запись значения wi.Number в консоль.
- Вызов метода Pass.Reference с передачей wi в качестве аргумента.
- Повторная запись значения wi.Number в консоль.

Как и прежде, при вызове метода Console.WriteLine вы сможете увидеть, изменяет ли значение wi.Number вызов Pass.Reference. Теперь метод doWork должен приобрести следующий вид (новые инструкции выделены жирным шрифтом):

```
static void doWork()
{
    // int i = 0;
    // Console.WriteLine(i);
    // Pass.Value(i);
    // Console.WriteLine(i);

    WrappedInt wi = new WrappedInt();
    Console.WriteLine(wi.Number);
    Pass.Reference(wi);
    Console.WriteLine(wi.Number);
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запуск приложения. На этот раз два значения, отображаемые в окне консоли, соответствуют значению wi.Number до и после вызова метода Pass.Reference. Вы должны увидеть, что на экране отображаются значения 0 и 42.

Нажмите Ввод, чтобы закрыть приложение и вернуться в среду Visual Studio 2015.

Давайте разберемся с тем, что показывает предыдущее упражнение. Значением wi.Number при инициализации, осуществляемой создаваемым компилятором пассивным конструктором, становится нуль. В переменной wi содержится ссылка на только что созданный WrappedInt-объект, содержащий int-значение. Затем переменная wi копируется в качестве аргумента в метод Pass.Reference. Поскольку WrappedInt является классом (ссылочный тип), и wi и param ссылаются на один и тот же WrappedInt-объект. Любые изменения, вносимые в содержимое объекта посредством переменной param в методе Pass.Reference, видны вследствие использования переменной wi, когда метод завершает работу. На следующей схеме показано, что происходит, когда WrappedInt-объект передается методу Pass.Reference в качестве аргумента (рис. 8.2).