Вне зависимости от типа используемого компилятора результатом компиляции будет являться *управляе-*

*мый модуль* (managed module) — стандартный переносимый исполняемый (portable

executable, PE) файл 32-разрядной (PE32) или 64-разрядной Windows (PE32+),

который требует для своего выполнения CLR. Кстати, управляемые сборки всегда

используют преимущества функции безопасности «предотвращения выполнения

данных» (DEP, Data Execution Prevention) и технологию ASLR (Address Space

Layout Optimization), применение этих технологий повышает информационную

безопасность всей системы.

Компиляторы машинного

Заголовок PE32 или

PE32+

Стандартный заголовок PE-файла Windows, аналогичный за-

головку Common Object File Format (COFF). Файл с заголов-

ком в формате PE32 может выполняться в 32- и 64-разрядной

версиях Windows, а с заголовком PE32+ — только в 64‑раз-

рядной. Заголовок обозначает тип файла: GUI, CUI или DLL,

он также имеет временную метку, показывающую, когда файл

был собран. Для модулей, содержащих только IL-код, основ-

ной объем информации в заголовке PE32(+) игнорируется.

В модулях, содержащих машинный код, этот заголовок содер-

жит сведения о машинном коде

Заголовок CLR Содержит информацию (интерпретируемую CLR и утилита-

ми), которая превращает этот модуль в управляемый. Заголо-

вок включает нужную версию CLR, некоторые флаги, метку

метаданных MethodDef точки входа в управляемый модуль

(метод Main), а также месторасположение/размер метаданных

модуля, ресурсов, строгого имени, некоторых флагов и пр.

Метаданные Каждый управляемый модуль содержит таблицы метаданных.

Есть два основных вида таблиц — это таблицы, описывающие

типы данных и их члены, определенные в исходном коде, и та-

блицы, описывающие типы данных и их члены, на которые

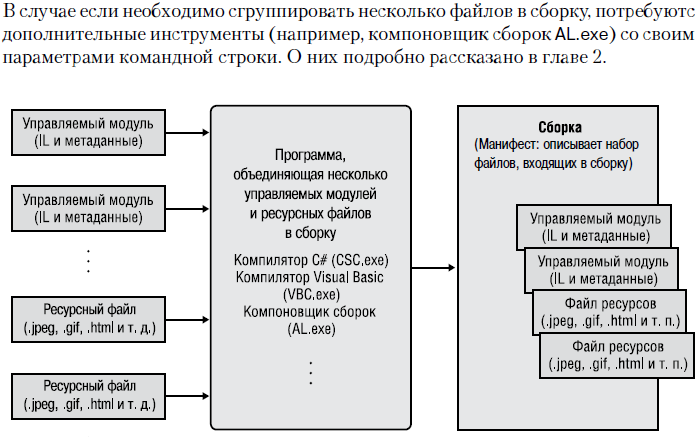
имеются ссылки в исходном коде

Код Intermediate

Language (IL)

Код, создаваемый компилятором при компиляции исходно-

го кода.



Модули сборки также содержат сведения о других сборках, на которые они

ссылаются (в том числе номера их версий). Эти данные делают сборку *самоописы-*

*ваемой* (self-describing). Другими словами, среда CLR может определить все прямые

зависимости данной сборки, необходимые для ее выполнения. Не нужно размещать

никакой дополнительной информации ни в системном реестре, ни в доменной

службе AD DS (Active Directory Domain Services). Вследствие этого развертывать

сборки гораздо проще, чем неуправляемые компоненты.

\*\*\*

Третье преимущество обобщенных интерфейсов заключается в том, что класс

может реализовать один интерфейс многократно, просто используя параметры

различного типа

public sealed class Number : IComparable<Int32>, IComparable<String>

{

private Int32 m\_val = 5;

// Этот метод реализует метод CompareTo интерфейса IComparable<Int32>

public Int32 CompareTo(Int32 n)

{

return m\_val.CompareTo(n);

}

// Этот метод реализует метод CompareTo интерфейса IComparable<String>

public Int32 CompareTo(String s)

{

return m\_val.CompareTo(Int32.Parse(s));

}

}

public static class Program

{

public static void Main()

{

Number n = new Number();

// Значение n сравнивается со значением 5 типа Int32

IComparable<Int32> cInt32 = n;

Int32 result = cInt32.CompareTo(5);

// Значение n сравнивается со значением "5" типа String

IComparable<String> cString = n;

result = cString.CompareTo("5");

}

}

\*\*\*

**Реализация нескольких интерфейсов**

**с одинаковыми сигнатурами**

**и именами методов**

public interface IWindow {

Object GetMenu();

}

public interface IRestaurant {

Object GetMenu();

}

Требуется определить тип, реализующий оба этих интерфейса. В этом случае

нужно реализовать члены типа путем явной реализации методов:

// Этот тип является производным от System.Object

// и реализует интерфейсы IWindow и IRestaurant

public sealed class MarioPizzeria : IWindow, IRestaurant {

// Реализация метода GetMenu интерфейса IWindow

Object IWindow.GetMenu() { ... }

// Реализация метода GetMenu интерфейса IRestaurant

Object IRestaurant.GetMenu() { ... }

// Метод GetMenu (необязательный),

// не имеющий отношения к интерфейсу

public Object GetMenu() { ... }

\*\*\*

GetNumericValue

{

Double d; // '\u0033' – это "цифра 3"

d = Char.GetNumericValue('3'); // Параметр '3'

// даст тот же результат

Console.WriteLine(d.ToString()); // Выводится "3"

// '\u00bc' — это "простая дробь одна четвертая ('1/4')"

d = Char.GetNumericValue('\u00bc');

Console.WriteLine(d.ToString()); // Выводится "0.25"

// 'A' — это "Латинская прописная буква A"

d = Char.GetNumericValue('A');

Console.WriteLine();

\*\*\*

**Интернирование строк**

String s1 = "Hello";

String s2 = "Hello";

Console.WriteLine(Object.ReferenceEquals(s1, s2)); // Должно быть 'False'

s1 = String.Intern(s1);

s2 = String.Intern(s2);

Console.WriteLine(Object.ReferenceEquals(s1, s2)); // 'True'

При первом вызове метода ReferenceEquals переменная s1 ссылается на объ-

ект-строку "Hello" в куче, а s2 — на другую объект-строку "Hello". Поскольку

ссылки разные, выводится значение False. Однако если выполнить этот код в CLR

версии 4.5, будет выведено значение True.

\*\*\*

Format

String s = String.Format("On {0:D}, {1} is {2:E} years old.",

new DateTime(2012, 4, 22, 14, 35, 5), "Aidan", 9);

\*\*\*

NumberStyles

Так, в следующем фрагменте при обращении к Parse генерируется исключение

System.FormatException, так как в начале разбираемой строки находится пробел:

Int32 x = Int32.Parse(" 123", NumberStyles.None, null);

Чтобы «пропустить» пробел, надо вызвать Parse с другим параметром style:

Int32 x = Int32.Parse(" 123", NumberStyles.AllowLeadingWhite, null);

Вот пример синтаксического разбора строки шестнадцатеричного числа:

Int32 x = Int32.Parse("1A", NumberStyles.HexNumber, null);

Console.WriteLine(x); // Отображает "26".

\*\*\*

Кодирование символов

String s = "Hi there.";

// Получаем объект, производный от Encoding, который "умеет" выполнять

// кодирование и декодирование с использованием UTF-8

Encoding encodingUTF8 = Encoding.UTF8;

// Выполняем кодирование строки в массив байтов

Byte[] encodedBytes = encodingUTF8.GetBytes(s);

// Показываем значение закодированных байтов

Console.WriteLine("Encoded bytes: " +

BitConverter.ToString(encodedBytes));

// Выполняем декодирование массива байтов обратно в строку

String decodedString = encodingUTF8.GetString(encodedBytes);

// Показываем декодированную строку

Console.WriteLine("Decoded string: " + decodedString);

\*\*\*

# Base-64

Byte[] bytes = new Byte[10];

new Random().NextBytes(bytes);

// Отображаем байты

Console.WriteLine(BitConverter.ToString(bytes));

// Декодируем байты в строку в кодировке base-64 и выводим эту строку

String s = Convert.ToBase64String(bytes);

Console.WriteLine(s);

// Кодируем строку в кодировке base-64 обратно в байты и выводим их

bytes = Convert.FromBase64String(s);

Console.WriteLine(BitConverter.ToString(bytes));

}

}

3B-B9-27-40-59-35-86-54-5F-F1

O7knQFk1hlRf8Q==

3B-B9-27-40-59-35-86-54-5F-F1

# Оператор new выполняет следующие действия:

1. Вычисление количества байтов, необходимых для хранения всех экземпляр-

ных полей типа и всех его базовых типов, включая System.Object (в котором

отсутствуют собственные экземплярные поля). Кроме того, в каждом объекте

кучи должны присутствовать дополнительные члены, называемые *указателем*

*на объект-тип* (type object pointer) и *индексом блока синхронизации* (sync block

index); они необходимы CLR для управления объектом. Байты этих дополни-

тельных членов добавляются к байтам, необходимым для размещения самого

объекта.

2. Выделение памяти для объекта с резервированием необходимого для данного

типа количества байтов в управляемой куче. Выделенные байты инициализи-

руются нулями (0).

3. Инициализация указателя на объект-тип и индекса блока синхронизации.

4. Вызов конструктора экземпляра типа с параметрами, указанными при вызове

new (в предыдущем примере это строка ConstructorParam1). Большинство

компиляторов автоматически включает в конструктор код вызова конструктора

базового класса. Каждый конструктор выполняет инициализацию определенных

в соответствующем типе полей. В частности, вызывается конструктор System.

Object, но он ничего не делает и просто возвращает управление.

Выполнив все эти операции, new возвращает ссылку (или указатель) на вновь

созданный объект. В предыдущем примере кода эта ссылка сохраняется в пере-

менной e типа Employee.

Кстати, у оператора new нет пары — оператора delete, то есть нет явного способа

освобождения памяти, занятой объектом. Уборкой мусора занимается среда CLR

(см. главу 21), которая автоматически находит объекты, ставшие ненужными или

недоступными, и освобождает занимаемую ими память.

# приведение к базовому и производному типам:

// Этот тип неявно наследует от типа System.Object

internal class Employee {

...

}

public sealed class Program {

public static void Main() {

// Приведение типа не требуется, т. к. new возвращает объект Employee,

// а Object — это базовый тип для Employee.

Object o = new Employee();

// Приведение типа обязательно, т. к. Employee — производный от Object

// В других языках (таких как Visual Basic) компилятор не потребует

// явного приведения

Employee e = (Employee) o;

}

}

# **Приведение типов в C# с помощью операторов is и as**

В C# существуют другие механизмы приведения типов. Например, оператор is

проверяет совместимость объекта с данным типом, а в качестве результата выдает

значение типа Boolean (true или false). Оператор is никогда не генерирует ис-

ключение. Взгляните на следующий код:

Object o = new Object();

Boolean b1 = (o is Object); // b1 равно true

Boolean b2 = (o is Employee); // b2 равно false

Приведение типов **127**

Для null-ссылок оператор is всегда возвращает false, так как объекта, тип

которого нужно проверить, не существует.

Обычно оператор is используется следующим образом:

if (o is Employee) {

Employee e = (Employee) o;

// Используем e внутри инструкции if

}

По сути, оператор as отличается от оператора приведения типа только тем,

что никогда не генерирует исключение. Если приведение типа невозможно, ре-

зультатом является null.

# Псевдоним

sing Microsoft; // Определяем приставку "Microsoft."

using Wintellect; // Определяем приставку "Wintellect."

// Имя WintellectWidget определяется как псевдоним для Wintellect.Widget

using WintellectWidget = Wintellect.Widget;

public sealed class Program {

public static void Main() {

WintellectWidget w = new WintellectWidget(); // Ошибки нет

}

}

# Checked

А вот пример с использованием оператора checked:

Byte b = 100; // Выдается исключение

b = checked((Byte) (b + 200)); // OverflowException

**148** Глава 5. Примитивные, ссылочные и значимые типы

Здесь b и 200 преобразуются в 32-разрядные числа и суммируются; резуль-

тат равен 300. Затем при преобразовании 300 в Byte генерируется исключение

OverflowException. Если приведение к типу Byte вывести из оператора checked,

исключения не будет:

b = (Byte) checked(b + 200); // b содержит 44; нет OverflowException

Наряду с операторами checked и unchecked в C# есть одноименные

Наряду с операторами checked и unchecked в C# есть одноименные инструк-

ции, позволяющие включить проверяемые или непроверяемые выражения внутрь

блока:

checked { // Начало проверяемого блока

Byte b = 100;

b = (Byte) (b + 200); // Это выражение проверяется на переполнение

} // Конец проверяемого блока

Кстати, внутри такого блока можно задействовать оператор += с Byte, который

немного упростит код:

checked { // Начало проверяемого блока

Byte b = 100;

b +=

# **Ссылочные и значимые типы**

При работе со ссылочными типами необходимо учитывать следующие

обстоятельства, относящиеся к производительности приложения:

. память для ссылочных типов всегда выделяется из управляемой кучи;

. каждый объект, размещаемый в куче, содержит дополнительные члены, под-

лежащие инициализации;

. незанятые полезной информацией байты объекта обнуляются (это касается

полей);

. размещение объекта в управляемой куче со временем инициирует сборку му-

сора.

// Ссылочный тип (поскольку 'class')

class SomeRef { public Int32 x; }

// Значимый тип (поскольку 'struct')

struct SomeVal { public Int32 x; }

static void ValueTypeDemo() {

SomeRef r1 = new SomeRef(); // Размещается в куче

SomeVal v1 = new SomeVal(); // Размещается в стеке

r1.x = 5; // Разыменовывание указателя

v1.x = 5; // Изменение в стеке

Console.WriteLine(r1.x); // Отображается "5"

Console.WriteLine(v1.x); // Также отображается "5"

// В левой части рис. 5.2 показан результат

// выполнения предыдущих строк

SomeRef r2 = r1; // Копируется только ссылка (указатель)

SomeVal v2 = v1; // Помещаем в стек и копируем члены

r1.x = 8; // Изменяются r1.x и r2.x

v1.x = 9; // Изменяется v1.x, но не v2.x

Console.WriteLine(r1.x); // Отображается "8"

Console.WriteLine(r2.x); // Отображается "8"

Console.WriteLine(v1.x); // Отображается "9"

Console.WriteLine(v2.x); // Отображается "5"

// В правой части рис. 5.2 показан результат

// выполнения ВСЕХ предыдущих строк

}

Единственное отличие в том, что экземпляр, создан-

ный оператором new, C# «считает» инициализированным. Поясню эту мысль на

следующем примере:

// Две следующие строки компилируются, так как C# считает,

// что поля в v1 инициализируются нулем

SomeVal v1 = new SomeVal();

Int32 a = v1.x;

// Следующие строки вызовут ошибку компиляции, поскольку C# не считает,

// что поля в v1 инициализируются нулем

SomeVal v1;

Int32 a = v1.x;

// error CS0170: Use of possibly unassigned field 'x'

// (ошибка CS0170: Используется поле 'x', которому не присвоено значение)

Объекты значимого типа существуют в двух формах (см. следующий раздел):

*неупакованной* (unboxed) и *упакованной* (boxed). Ссылочные типы бывают только

в упакованной форме.

. Значимые типы являются производными от System.ValueType. Этот тип имеет

те же методы, что и System.Object. Однако System.ValueType переопределяет

метод Equals, который возвращает true, если значения полей в обоих объектах

совпадают. Кроме того, в System.ValueType переопределен метод GetHashCode,

который создает хеш-код по алгоритму, учитывающему значения полей экзем-

пляра объекта. Из-за проблем с производительностью в реализации по умол-

чанию, определяя собственные значимые типы значений, надо переопределить

и написать свою реализацию методов Equals и GetHashCode. О методах Equals

и GetHashCode рассказано в конце этой главы.

. Поскольку в объявлении нового значимого или ссылочного типа нельзя указы-

вать значимый тип в качестве базового класса, создавать в значимом типе новые

виртуальные методы нельзя. Методы не могут быть абстрактными и неявно

являются запечатанными (то есть их нельзя переопределить).

. Переменные ссылочного типа содержат адреса объектов в куче. Когда пере-

менная ссылочного типа создается, ей по умолчанию присваивается null, то

есть в этот момент она не указывает на действительный объект. Попытка за-

действовать переменную с таким значением приведет к генерации исключения

NullReferenceException. В то же время в переменной значимого типа всегда

содержится некое значение соответствующего типа, а при инициализации

всем членам этого типа присваивается 0. Поскольку переменная значимого

типа не является указателем, при обращении к значимому типу исключение

NullReferenceException возникнуть не может. CLR поддерживает понятие

значимого типа особого вида, допускающего присваивание null (nullable types).

Этот тип обсуждается в главе 19.

. Когда переменной значимого типа присваивается другая переменная значимого

типа, выполняется копирование всех ее полей. Когда переменной ссылочно-

Ссылочные и значимые типы **155**

го типа присваивается переменная ссылочного типа, копируется только ее

адрес.

. Вследствие сказанного в предыдущем пункте несколько переменных ссылочного

типа могут ссылаться на один объект в куче, благодаря чему, работая с одной

переменной, можно изменить объект, на который ссылается другая переменная.

В то же время каждая переменная значимого типа имеет собственную копию

данных «объекта», поэтому операции с одной переменной значимого типа не

влияют на другую переменную.

. Так как неупакованные значимые типы не размещаются в куче, отведенная для

них память освобождается сразу при возвращении управления методом, в кото-

ром описан экземпляр этого типа (в отличие от ожидания уборки мусора).

**Упаковка и распаковка значимых типов**

// Объявляем значимый тип

struct Point {

public Int32 x, y;

}

public sealed class Program {

public static void Main() {

Упаковка и распаковка значимых типов **157**

ArrayList a = new ArrayList();

Point p; // Выделяется память для Point (не в куче)

for (Int32 i = 0; i < 10; i++) {

p.x = p.y = i; // Инициализация членов в нашем значимом типе

a.Add(p); // Упаковка значимого типа и добавление

// ссылки в ArrayList

}

...

}

}

В каждой итерации цикла инициализируются поля значимого типа Point, после

чего Point помещается в ArrayList. Задумаемся, что же помещается в ArrayList:

сама структура Point, адрес структуры Point или что-то иное? За ответом обратимся

к методу Add типа ArrayList и посмотрим описание его параметра. В данном случае

прототип метода Add выглядит следующим образом:

public virtual Int32 Add(Object value);

Отсюда видно, что в параметре Add должен передаваться тип Object, то есть

ссылка (или указатель) на объект в управляемой куче.

При упаковке экземпляра значимого типа происходит следующее.

1. В управляемой куче выделяется память. Ее объем определяется длиной значи-

мого типа и двумя дополнительными членами — указателем на типовой объект

и индексом блока синхронизации. Эти члены необходимы для всех объектов

в управляемой куче.

2. Поля значимого типа копируются в память, только что выделенную в куче.

3. Возвращается адрес объекта. Этот адрес является ссылкой на объект, то есть

значимый тип превращается в ссылочный.

public static void Main() {

Int32 x = 5;

Object o = x; // Упаковка x; o указывает на упакованный объект

Int16 y = (Int16) o; // Генерируется InvalidCastException

}

Казалось бы, можно взять упакованный экземпляр Int32, на который указы-

вает o, и привести к типу Int16. Однако при распаковке объекта должно быть вы-

полнено приведение к неупакованному типу (в нашем случае — к Int32). Вот как

выглядит правильный вариант:

public static void Main() {

Int32 x = 5;

Object o = x; // Упаковка x; o указывает на упакованный объект

Int16 y = (Int16)(Int32) o; // Распаковка, а затем приведение типа

}

Взгляните на следующий пример.

using System;

public sealed class Program {

public static void Main() {

Int32 v = 5; // Создаем переменную упакованного значимого типа

#if INEFFICIENT

// При компиляции следующей строки v упакуется

// три раза, расходуя и время, и память

Console.WriteLine("{0}, {1}, {2}", v, v, v);

#else

// Следующие строки дают тот же результат,

// но выполняются намного быстрее и расходуют меньше памяти

Object o = v; // Упакуем вручную v (только единожды)

// При компиляции следующей строки код упаковки не создается

Console.WriteLine("{0}, {1}, {2}", o, o, o);

#endif

}

}

using System;

// Point - значимый тип.

internal struct Point {

private Int32 m\_x, m\_y;

public Point(Int32 x, Int32 y) {

m\_x = x;

m\_y = y;

}

public void Change(Int32 x, Int32 y) {

m\_x = x; m\_y = y;

}

public override String ToString() {

return String.Format("({0}, {1})", m\_x.ToString(), m\_y.ToString());

}

}

public sealed class Program {

public static void Main() {

Point p = new Point(1, 1);

Console.WriteLine(p);

p.Change(2, 2);

Console.WriteLine(p);

Object o = p;

*продолжение* 

**170** Глава 5. Примитивные, ссылочные и значимые типы

Console.WriteLine(o);

((Point) o).Change(3, 3);

Console.WriteLine(o);

}

}

Все просто: Main создает в стеке экземпляр p типа Point и устанавливает его поля

m\_x и m\_y равными 1. Затем p пакуется до первого обращения к методу WriteLine,

который вызывает ToString для упакованного типа Point, в результате выводится,

как и ожидалось, (1, 1). Затем p применяется для вызова метода Change, который

изменяет значения полей m\_x и m\_y объекта p в стеке на 2. При втором обращении

к WriteLine, как и предполагалось, выводится (2, 2).

Далее p упаковывается в третий раз — o ссылается на упакованный объект типа

Point. При третьем обращении к WriteLine снова выводится (2, 2), что опять

вполне ожидаемо. И наконец, я обращаюсь к методу Change для изменения полей

в упакованном объекте типа Point. Между тем Object (тип переменной o) ничего

не «знает» о методе Change, так что сначала нужно привести o к Point. При таком

приведении типа o распаковывается, и поля упакованного объекта типа Point

копируются во временный объект типа Point в стеке потока. Поля m\_x и m\_y этого

временного объекта устанавливаются равными 3, но это обращение к Change не

влияет на упакованный объект Point. При обращении к WriteLine снова выводится

(2, 2). Для многих разработчиков это оказывается неожиданным.

# как метод

# интерфейса может изменить поля в упакованном значимом типе. В C# сделать это

без интерфейсов нельзя.

// Интерфейс, определяющий метод Change

internal interface IChangeBoxedPoint {

void Change(Int32 x, Int32 y);

}

// p упаковывается, упакованный объект изменяется и освобождается

((IChangeBoxedPoint) p).Change(4, 4);

Console.WriteLine(p);

// Упакованный объект изменяется и выводится

((IChangeBoxedPoint) o).Change(5, 5);

Console.WriteLine(o);

# **Равенство и тождество объектов**

public class Object {

public virtual Boolean Equals(Object obj) {

// Сравниваемый объект не может быть равным null

if (obj == null) return false;

// Объекты разных типов не могут быть равны

if (this.GetType() != obj.GetType()) return false;

// Если типы объектов совпадают, возвращаем true при условии,

// что все их поля попарно равны.

// Так как в System.Object не определены поля,

// следует считать, что поля равны

return true;

}

}\

# **Примитивный тип данных dynamic**

internal static class DynamicDemo {

public static void Main() {

dynamic value;

for (Int32 demo = 0; demo < 2; demo++) {

value = (demo == 0) ? (dynamic) 5 : (dynamic) "A";

value = value + value;

M(value);

}

}

private static void M(Int32 n) { Console.WriteLine("M(Int32): " + n); }

private static void M(String s) { Console.WriteLine("M(String): " + s); }

}

После выполнения метода Main получается следующий результат:

M(Int32): 10

M(String): AA

Однако

компилятор разрешит выполнить приведение типа dynamic к другому типу с ис-

пользованием синтаксиса неявного приведения.

Object o1 = 123; // OK: Неявное приведение Int32 к Object (упаковка)

Int32 n1 = o1; // Ошибка: Нет неявного приведения Object к Int32

Int32 n2 = (Int32) o1; // OK: Явное приведение Object к Int32 (распаковка)

dynamic d1 = 123; // OK: Неявное приведение Int32 к dynamic (упаковка)

Int32 n3 = d; // OK: Неявное приведение dynamic к Int32 (распаковка)

Не путайте типы dynamic и var. Объявление локальной переменной как var является

синтаксическим указанием компилятору подставлять специальные данные из со-

ответствующего выражения. Ключевое слово var может использоваться только для

объявления локальных переменных внутри метода, тогда как ключевое слово dynamic

может указываться с локальными переменными, полями и аргументами. Вы не можете

привести выражение к типу var, но вы можете привести его к типу dynamic. Вы должны

явно инициализировать переменную, объявленную как var, тогда как переменную,

объявленную как dynamic, инициализировать нельзя.

**Видимость типа**

При определении типа с видимостью в рамках файла, а не другого типа его можно

сделать *открытым* (public) или *внутренним* (internal). Открытый тип доступен

любому коду любой сборки. Внутренний тип доступен только в той сборке, где он

определен. По умолчанию компилятор C# делает тип внутренним (с более огра-

ниченной видимостью). Вот несколько примеров.

using System;

// Открытый тип доступен из любой сборки

public class ThisIsAPublicType { ... }

// Внутренний тип доступен только из собственной сборки

internal class ThisIsAnInternalType { ... }

// Это внутренний тип, так как модификатор доступа не указан явно

class ThisIsAlsoAnInternalType { ... }

То есть нам необходим способ, который бы позволил группе *А* определить

свои типы как внутренние, но в то же время предоставить группе *Б* доступ к этим

типам. Для таких ситуаций в CLR и C# предусмотрен механизм *дружественных*

*сборок* (friend assemblies).

Приведем пример сборки, которая объявляет дружественными

две другие сборки со строгими именами Wintellect и Microsoft:

using System;

using System.Runtime.CompilerServices; // Для атрибута InternalsVisibleTo

// Внутренние типы этой сборки доступны из кода двух следующих сборок

// (независимо от версии или региональных стандартов)

[assembly:InternalsVisibleTo("Wintellect, PublicKey=12345678...90abcdef")]

[assembly:InternalsVisibleTo("Microsoft, PublicKey=b77a5c56...1934e089")]

internal sealed class SomeInternalType { ... }

internal sealed class AnotherInternalType { ... }

Обратиться из дружественной сборки к внутренним типам представленной здесь

сборки очень просто. Например, дружественная сборка Wintellect с открытым клю-

чом 12345678…90abcdef может обратиться к внутреннему типу SomeInternalType

представленной сборки следующим образом:

using System;

internal sealed class Foo {

private static Object SomeMethod() {

// Эта сборка Wintellect получает доступ к внутреннему типу

// другой сборки, как если бы он был открытым

SomeInternalType sit = new SomeInternalType();

return sit;

}

}

# Static class

Компилятор налагает на статический класс ряд ограничений.

. Класс должен быть прямым потомком System.Object — наследование любому

другому базовому классу лишено смысла, поскольку наследование применимо

только к объектам, а создать экземпляр статического класса невозможно.

. Класс не должен реализовывать никаких интерфейсов, поскольку методы ин-

терфейса можно вызывать только через экземпляры класса.

. В классе можно определять только статические члены (поля, методы, свойства

и события). Любые экземплярные члены вызовут ошибку компиляции.

. Класс нельзя использовать в качестве поля, параметра метода или локальной

переменной, поскольку это подразумевает существование переменной, ссы-

лающейся на экземпляр, что запрещено. Обнаружив подобное обращение со

статическим классом, компилятор вернет сообщение об ошибке.

# Partial

Есть три основные причины, по которым

исходный код разбивается на несколько файлов.

. **Управление версиями.** Представьте, что определение типа содержит большой

объем исходного кода. Если этот тип будут одновременно редактировать два

программиста, по завершении работы им придется каким-то образом объединять

свои результаты, что весьма неудобно. Ключевое слово partial позволяет раз-

бить исходный код типа на несколько файлов, чтобы один и тот же тип могли

одновременно редактировать несколько программистов.

# Callvirt

Иногда компилятор вместо callvirt использует для вызова виртуального

метода команду call. Такое поведение выглядит странно, но следующий пример

показывает, почему это действительно бывает необходимо.

internal class SomeClass {

// ToString - виртуальный метод базового класса Object

public override String ToString() {

// Компилятор использует команду call для невиртуального вызова

// метода ToString класса Object

// Если бы компилятор вместо call использовал callvirt, этот

// метод продолжал бы рекурсивно вызывать сам себя до переполнения стека

return base.ToString();

}

}

При вызове виртуального метода base.ToString компилятор C# вставляет

команду call, чтобы метод ToString базового типа вызывался невиртуально. Это

необходимо, ведь если ToString вызвать виртуально, вызов будет выполняться

рекурсивно до переполнения стека потока — что, разумеется, нежелательно.

Компиляторы стремятся использовать команду call при вызове методов

# Const

using System;

public sealed class SomeLibraryType {

// ПРИМЕЧАНИЕ: C# не позволяет использовать для констант модификатор

// static, поскольку всегда подразумевается, что константы являются

// статическими

public const Int32 MaxEntriesInList = 50;

}

Затем построим сборку приложения из следующего кода:

using System;

public sealed class Program {

public static void Main() {

Console.WriteLine("Max entries supported in list: "

+ SomeLibraryType.MaxEntriesInList);

}

}

ReadOnly

readonly Запись в поле разрешается только из кода конструктора

Неизменность поля ссылочного типа означает неизменность ссылки, которую этот

тип содержит, а вовсе не объекта, на которую указывает ссылка, например:

public sealed class АТуре {

// InvalidChars всегда ссылается на один объект массива

public static readonly Char[] InvalidChars = new Char[] { 'А', 'В', 'C'};

}

public sealed class AnotherType {

public static void M() {

// Следующие строки кода вполне корректны, компилируются

// и успешно изменяют символы в массиве InvalidChars

АТуре.InvalidChars[0] = 'X';

АТуре.InvalidChars[1] = 'Y';

АТуре.InvalidChars[2] = 'Z';

// Следующая строка некорректна и не скомпилируется,

// так как ссылка InvalidChars изменяться не может

АТуре.InvalidChars = new Char[] { 'X', 'Y', 'Z' };

}

}

# Конструктор

Для статических классов (запечатанных и абстрактных) компилятор не

создает конструктор по умолчанию.

Struct constructor

internal struct Point {

public Int32 m\_x, m\_y;

public Point() {

m\_x = m\_y = 5;

}

}

internal sealed class Rectangle {

public Point m\_topLeft, m\_bottomRight;

public Rectangle() {

}

}

При попытке ском-

пилировать его компилятор C# генерирует сообщение об ошибке (ошибка CS0568:

структура не может содержать явные конструкторы без параметров):

error CS0568: Structs cannot contain explicit parameterless constructors

C# преднамеренно запрещает определять конструкторы без параметров у зна-

чимых типов, чтобы не вводить разработчиков в заблуждение относительно того,

какой конструктор вызывается. Если конструктор определить нельзя, компилятор

Конструкторы экземпляров и структуры (значимые типы) **221**

никогда не будет автоматически генерировать код, вызывающий такой конструктор.

В отсутствие конструктора без параметров поля значимого типа всегда инициали-

зируются нулями/null.

internal struct SomeValType

{

// В значимый тип нельзя подставлять инициализацию экземплярных полей

private Int32 m\_x = 5;

}

чимого типа, как это сделано здесь:

// C# позволяет значимым типам иметь конструкторы с параметрами

public SomeValType(Int32 x) {

// Выглядит необычно, но компилируется прекрасно,

// и все поля инициализируются значениями 0 или null

this = new SomeValType();

m\_x = x; // Присваивает m\_x значение x

// Обратите внимание, что поле m\_y было инициализировано нулем

}

# Static Constructor

Вот как определяются ссылочные и значимые типы с конструкторами

в программах на C#:

internal sealed class SomeRefType {

static SomeRefType() {

// Исполняется при первом обращении к ссылочному типу SomeRefType

}

Конструкторы типов **223**

}

internal struct SomeValType {

// C# на самом деле допускает определять для значимых типов

// конструкторы без параметров

static SomeValType() {

// Исполняется при первом обращении к значимому типу SomeValType

}

}

Обратите внимание, что конструкторы типов определяют так же, как кон-

структоры экземпляров без параметров за исключением того, что их помечают как

статические. Кроме того, конструкторы типов всегда должны быть закрытыми (C#

делает их закрытыми автоматически). Однако если явно пометить в исходном тексте

программы конструктор типа как закрытый (или как-то иначе), компилятор C#

выведет сообщение об ошибке: (ошибка CS0515: 'SomeValType.Some-ValType()':

в статических конструкторах нельзя использовать модификаторы доступа):

error CS0515: 'SomeValType.SomeValType()': access modifiers are not allowed on

static constructors

Конструкторы типов всегда должны быть закрытыми, чтобы код разработчика

не смог их вызвать, напротив, в то же время среда CLR всегда способна вызвать

конструктор типа.

**Внимание**

Хотя конструктор типа можно определить в значимом типе, этого никогда не следует

делать, так как иногда CLR не вызывает статический конструктор значимого типа. Например:

internal struct SomeValType {

static SomeValTypeQ {

Console.WriteLine("This never gets displayed");

}

public Int32 m\_x;

}

# Operator+

public sealed class Complex {

public static Complex operator+(Complex c1, Complex c2) { ... }

}

Таким образом, показанный ранее тип Complex

можно было бы определить и так:

public sealed class Complex {

public static Complex operator+(Complex c1, Complex c2) { ... }

Explicit and Implicit

public sealed class Rational {

// Создает Rational из Int32

public Rational(Int32 num) { ... }

// Создает Rational из Single

public Rational(Single num) { ... }

// Преобразует Rational в Int32

public Int32 ToInt32() { ... }

// Преобразует Rational в Single

public Single ToSingle() { ... }

// Неявно создает Rational из Int32 и возвращает полученный объект

public static implicit operator Rational(Int32 num) {

return new Rational(num); }

// Неявно создает Rational из Single и возвращает полученный объект

public static implicit operator Rational(Single num) {

return new Rational(num); }

// Явно возвращает объект типа Int32, полученный из Rational

public static explicit operator Int32(Rational r) {

return r.ToInt32(); }

// Явно возвращает объект типа Single, полученный из Rational

public static explicit operator Single(Rational r) {

return r.ToSingle();

}

}

писать (на C#):

public sealed class Program {

public static void Main() {

Rational r1 = 5; // Неявное приведение Int32 к Rational

Rational r2 = 2.5F; // Неявное приведение Single к Rational

Int32 x = (Int32) r1; // Явное приведение Rational к Int32

Single s = (Single) r2; // Явное приведение Rational к Single

}

}

# Расширяющие методы

А сейчас я попробую объяснить, что именно делают методы расширения. Они

позволяют вам определить статический метод, который вызывается посредством

синтаксиса экземплярного метода. Иначе говоря, мы можем определить собствен-

ный метод IndexOf — и три проблемы, упомянутые выше, исчезнут. Для того чтобы

превратить метод IndexOf в метод расширения, мы просто добавим ключевое слово

this перед первым аргументом:

public static class StringBuilderExtensions {

public static Int32 IndexOf(this StringBuilder sb, Char value) {

for (Int32 index = 0; index < sb.Length; index++)

if (sb[index] == value) return index;

return -1;

}

}

Методы расширения также можно определять и для типов-делегатов, напри-

мер:

public static void InvokeAndCatch<TException>(this Action<Object> d, Object o)

where TException : Exception {

try { d(o); }

catch (TException) { }

}

Пример вызова:

Action<Object> action = o => Console.WriteLine(o.GetType());

// Выдает NullReferenceException

action.InvokeAndCatch<NullReferenceException>(null);

// Поглощает NullReferenceException

Кроме того, можно добавлять методы

# IEnumerable<T>

Методы расширения **239**

public static void Main() {

// Показывает каждый символ в каждой строке консоли

"Grant".ShowItems();

// Показывает каждую строку в каждой строке консоли

new[] { "Jeff", "Kristin" }.ShowItems();

// Показывает каждый Int32 в каждой строчке консоли.

new List<Int32>() { 1, 2, 3 }.ShowItems();

# Частичные методы

Для решения проблемы переопределения поведения можно задействовать ча-

стичные методы языка C#. В следующем коде для достижения той же семантики,

что и в предыдущем коде, используются частичные методы:

// Сгенерированный при помощи инструмента программный код

internal sealed partial class Base {

private String m\_name;

// Это объявление с определением частичного метода вызывается

// перед изменением поля m\_name

partial void OnNameChanging(String value);

public String Name {

get { return m\_name; }

set {

// Информирование класса о потенциальном изменении

OnNameChanging(value.ToUpper());

m\_name = value; // Изменение поля

}

}

}

// Написанный программистом код, содержащийся в другом файле

internal sealed partial class Base {

// Это объявление с реализацией частичного метода вызывается перед тем,

// как будет изменено поле m\_name

partial void OnNameChanging(String value) {

if (String.IsNullOrEmpty(value))

throw new ArgumentNullException("value");

}

}

# **Необязательные и именованные параметры**

private static void M(Int32 x = 9, String s = "A",

DateTime dt = default(DateTime), Guid guid = new Guid())

{

Console.WriteLine("x={0}, s={1}, dt={2}, guid={3}", x, s, dt, guid);

}

public static void Main()

{

M();

// 2. Аналогично: M(8, "X", default(DateTime), new Guid());

M(8, "X");

// 3. Аналогично: M(5, "A", DateTime.Now, Guid.NewGuid());

M(5, guid: Guid.NewGuid(), dt: DateTime.Now);

// 4. Аналогично: M(0, "1", default(DateTime), new Guid());

M(s\_n++, s\_n++.ToString());

// 5. Аналогично: String t1 = "2"; Int32 t2 = 3;

// M(t2, t1, default(DateTime), new Guid());

M(s: (s\_n++).ToString(), x: s\_n++);

}

. Значения по умолчанию указываются для параметров методов, конструкторов

методов и параметрических свойств (индексаторов C#). Также их можно ука-

зывать для параметров, являющихся частью определения делегатов. В резуль-

тате при вызове этого типа делегата аргументы можно опускать, используя их

значения по умолчанию.

. Параметры со значениями по умолчанию должны следовать за всеми остальными

параметрами. Другими словами, если указан параметр со значением по умолчанию,

значения по умолчанию должны иметь и все параметры, расположенные справа от

него. Например, если при определении метода M удалить значение по умолчанию

("A") для параметра s, компилятор выдаст сообщение об ошибке. Существует

только одно исключение из правил — параметр массива, помеченный ключевым

словом params (о котором мы подробно поговорим чуть позже). Он должен рас-

полагаться после всех прочих параметров, в том числе имеющих значение по

умолчанию. При этом сам массив значения по умолчанию иметь не может.

. Во время компиляции значения по умолчанию должны оставаться неизменными.

То есть задавать значения по умолчанию можно для параметров примитивных

типов, перечисленных в табл. 5.1 главы 5. Сюда относятся также перечисли-

мые типы и ссылочные типы, допускающие присвоение значения null. Для

параметров произвольного значимого типа значение по умолчанию задается

как экземпляр этого типа с полями, содержащими нули. Можно использовать

как ключевое слово default, так и ключевое слово new, в обоих случаях генери-

Необязательные и именованные параметры **247**

руется одинаковый IL-код. С примерами обоих вариантов синтаксиса мы уже

встречались в методе M при задании значений по умолчанию для параметров dt

и guid соответственно.

. Запрещается переименовывать параметрические переменные, так как это влечет

за собой необходимость редактирования вызывающего кода, который передает

аргументы по имени параметра. Скажем, если в объявлении метода M переиме-

новать переменную dt в dateTime, то третий вызов метода станет причиной по-

явления следующего сообщения компилятора (ошибка CS1739: в подходящей

перегруженной версии 'M' отсутствует параметр с именем 'dt'):

"error CS1739: The best overload for 'M' does not have a parameter

named 'dt'

. При вызове метода извне модуля изменение значения параметров по умолчанию

является потенциально опасным. Вызывающая сторона использует значение

по умолчанию в процессе работы. Если изменить его и не перекомпилировать

код, содержащий вызов, в вызываемый метод будет передано прежнее значение.

В качестве индикатора поведения можно использовать значение по умолчанию

0 или null. В результате исчезает необходимость повторной компиляции кода

вызывающей стороны.

Для параметров, помеченных ключевыми словами ref или out, значения по

умолчанию не задаются.

Существуют также дополнительные правила вызова методов с использованием

необязательных или именованных параметров:

. Аргументы можно передавать в произвольном порядке; но именованные аргу-

менты должны находиться в конце списка.

. Передача аргумента по имени возможна для параметров, не имеющих значения

по умолчанию, но при этом компилятору должны быть переданы все аргументы,

необходимые для компиляции (c указанием их позиции или имени).

. В C# между запятыми не могут отсутствовать аргументы. Иначе говоря, запись

M(1, ,DateTime.Now) недопустима, так как ведет к нечитабельному коду. Что-

бы опустить аргумент для параметра со значением по умолчанию, передавайте

аргументы по именам параметров.

. Вот как передать аргумент по имени параметра, требующего ключевого слова

ref/out:

**248** Глава 9. Параметры

// Объявление метода:

private static void M(ref Int32 x) { ... }

// Вызов метода:

Int32 a = 5;

M(x: ref a);

# Dynamic vs var

Не путайте ключевые слова dynamic и var. Объявление локальной переменной с клю-

чевым слово var является не более чем синтаксическим сокращением, заставляющим

компилятор определить тип данных по выражению. Данное ключевое слово служит

только для объявления локальных переменных внутри метода, в то время как клю-

чевое слово dynamic используется для локальных переменных, полей и аргументов. Невозможно привести выражение к типу var, но такая операция вполне допустима

для типа dynamic. Переменные, объявленные с ключевым словом var, должны ини-

циализироваться явно, что не обязательно для переменных типа dynamic. Более

подробную информацию о динамическом типе вы найдете в главе 5.

# Ref and Out

CLR также позволяет передавать параметры по ссылке, а не по значению. В C#

это делается с помощью ключевых слов out и ref. Оба заставляют компилятор

генерировать метаданные, описывающие параметр как переданный по ссылке.

Компилятор использует эти метаданные для генерирования кода, передающего

вместо самого параметра его адрес.

С точки зрения CLR, ключевые слова

public sealed class Program {

public static void Main() {

Int32 x; // Инициализация x

GetVal(out x); // Инициализация x не обязательна

Console.WriteLine(x); // Выводится 10

}

private static void GetVal(out Int32 v) {

v = 10; // Этот метод должен инициализировать переменную V

}

}

А теперь рассмотрим аналогичный пример с ключевым словом ref:

public sealed class Program {

public static void Main() {

Int32 x = 5; // Инициализация x

AddVal(ref x); // x требуется инициализировать

Console.WriteLine(x); // Выводится 15

}

private static void AddVal(ref Int32 v) {

v += 10; // Этот метод может использовать инициализированный параметр v

}

}

ref. Например, следующий код на C# вполне допустим

и прекрасно компилируется:

public sealed class Point {

static void Add(Point p) { ... }

static void Add(ref Point p) { ... }

}

# Обобщения

Однако, как оказалось, эти методы можно заставить работать при помощи обоб-

щений. Вот так следует исправить показанный ранее метод Swap:

public static void Swap<T>(ref T a, ref T b) {

T t = b;

b = a;

a = t;

}

После этого следующий код (идентичный ранее показанному) будет без проблем

компилироваться и выполняться:

public static void SomeMethod() {

String s1 = "Jeffrey";

String s2 = "Richter";

Swap(ref s1, ref s2);

Console.WriteLine(s1); // Выводит "Richter"

Console.WriteLine(s2); // Выводит "Jeffrey"

}

# Params

Должен быть последним в списке параметров метода

Метод, принимающий переменное число аргументов, объявляют так:

static Int32 Add(params Int32[] values) {

// ПРИМЕЧАНИЕ: при необходимости этот массив

// можно передать другим методам

Int32 sum = 0;

if (values != null) {

for (Int32 x = 0; x < values.Length; x++)

sum += values[x];

}

return sum;

}

public static void Main() {

// Выводит "15"

Console.WriteLine(Add(new Int32[] { 1, 2, 3, 4, 5 } ));

}

Не вызывает сомнений утверждение, что этот массив легко инициализировать

произвольным числом элементов и передать для обработки методу Add. Показан-

ный здесь код немного неуклюж, хотя он корректно компилируется и работает.

Разработчики, конечно, предпочли бы вызывать метод Add так:

public static void Main() {

// Выводит "15"

Console.WriteLine(Add(1, 2, 3, 4, 5));

}

как написать метод, принима-

ющий произвольное количество параметров любого типа? Ответ прост: достаточно

отредактировать прототип метода, заставив его вместо Int32[] принимать Object[].

Следующий метод выводит значения Type всех переданных ему объектов:

public sealed class Program {

public static void Main() {

DisplayTypes(new Object(), new Random(), "Jeff", 5);

}

private static void DisplayTypes(params Object[] objects) {

if (objects != null) {

foreach (Object o in objects)

Console.WriteLine(o.GetType());

}

}

}

# **Типы параметров**

# **и возвращаемых значений**

// Рекомендуется в этом методе использовать параметр слабого типа

public void ManipulateItems<T>(IEnumerable<T> collection) { ... }

// Не рекомендуется в этом методе использовать параметр сильного типа

public void ManipulateItems<T>(List<T> collection) { ... }

Причина, конечно же, в том, что первый метод можно вызывать, передав в него

массив, объект List<T>, объект String и т. п., то есть любой объект, тип которого

реализует интерфейс IEnumerable<T>. Второй метод принимает только объекты

List<T>, с массивами или объектами String он работать уже не может. Ясно, что

первый метод предпочтительнее, так как он гибче и может использоваться в более

разнообразных ситуациях.

Естественно, при создании метода, получающего список (а не просто любой

перечислимый объект), нужно объявлять тип параметра как IList<T>, в то время

как типа List<T> лучше избегать. Именно такой подход позволит вызывающему

коду передавать массивы и другие объекты, тип которых реализует IList<T>.

Потому, к примеру, при

реализации метода, обрабатывающего байты из потока, пишем следующее:

// Рекомендуется в этом методе использовать параметр мягкого типа

public void ProcessBytes(Stream someStream) { ... }

// Не рекомендуется в этом методе использовать параметр сильного типа

public void ProcessBytes(FileStream fileStream) { ... }

В то же время, объявляя тип возвращаемого методом объекта, желательно

выбирать самый сильный из доступных вариантов (пытаясь не ограничиваться

конкретным типом). Например, лучше объявлять метод, возвращающий объект

FileStream, а не Stream:

// Рекомендуется в этом методе использовать

// сильный тип возвращаемого объекта

public FileStream OpenFile() { ... }

// Не рекомендуется в этом методе использовать

// слабый тип возвращаемого объекта

public Stream OpenFile() { ... }

# Свойства

public void SetAge(Int32 value) {

if (value < 0)

throw new ArgumentOutOfRangeException("value", value.ToString(),

"The value must be greater than or equal to 0");

m\_Age = value;

}

public sealed class Employee {

private String m\_Name;

private Int32 m\_Age;

public String Name {

get { return(m\_Name); }

set { m\_Name = value; } // Ключевое слово value

} // идентифицирует новое значение

public Int32 Age {

get { return(m\_Age); }

set {

if (value < 0) // Ключевое слово value всегда

// идентифицирует новое значение

throw new ArgumentOutOfRangeException("value", value.ToString(),

"The value must be greater than or equal to 0");

m\_Age = value;

}

}

}

# Режим вычисления свойств

отключить режим

вычислений для свойств, указанных в окне просмотра отладчика. Для этого выбе-

рите команду ToolsOptions, в списке открывшегося окна Options раскройте ветвь

DebuggingGeneral и сбросьте флажок Enable Property Evaluation And Other Implicit

Function Calls (рис. 10.1). Обратите внимание, что даже отключив таким образом

вычисления для свойства, все равно можно будет добавить свойство в окно про-

смотра отладчика и вручную запустить вычисления, щелкнув мышью на значке

вычислений в колонке Value окна просмотра отладчика Visual Studio.

# Инициализация колекции объекта

public sealed class Classroom {

private List<String> m\_students = new List<String>();

public List<String> Students { get { return m\_students; } }

public Classroom() {}

}

Следующий код создает объект Classroom и инициализирует коллекцию Students:

public static void M() {

Classroom classroom = new Classroom {

Students = { "Jeff", "Kristin", "Aidan", "Grant" }

};

// Вывести имена 4 студентов, находящихся в классе

foreach (var student in classroom.Students)

Console.WriteLine(student);

}

var table = new Dictionary<String, Int32> {

{ "Jeffrey", 1 }, { "Kristin", 2 }, { "Aidan", 3 }, { "Grant", 4 }

};

# Анонимные типы (Typle type)

// Определение типа, создание сущности и инициализация свойств

var o1 = new { Name = "Jeff", Year = 1964 };

// Вывод свойств на консоль

Console.WriteLine("Name={0}, Year={1}", o1.Name, o1.Year); // Выводит:

// Name=Jeff, Year=1964

Здесь создается анонимный тип, потому что не был определен тип имени после

слова new, таким образом, компилятор автоматически создает имя типа

Когда вы пишете этот код, компилятор определяет тип каждого выражения,

создает закрытые поля этих типов, для каждого типа поля создает открытые свой-

ства только для чтения и для всех этих выражений создает конструктор.

var r = new[] {new {Name = "Roma", Age = 20}, new {Name = "Azaza", Age = 21} };

# ExpandoObject

Пример кода с использованием объекта ExpandoObject:

dynamic e = new System.Dynamic.ExpandoObject();

e.x = 6; // Добавление свойства 'x' типа Int32

// со значением 6

Свойства с параметрами **279**

e.y = "Jeff"; // Добавление свойства 'y' строкового типа

// со значением "Jeff"

e.z = null; // Добавление свойста 'z' объекта

// со значением null

// Просмотр всех свойств и других значений

foreach (var v in (IDictionary<String, Object>)e)

Console.WriteLine("Key={0}, V={1}", v.Key, v.Value);

// Удаление свойства 'x' и его значения

var d = (IDictionary<String, Object>)e;

d.Remove("x");

# Обобщение

*Обобщения* (generics) —

еще один механизм, поддерживаемый средой CLR и языками программирования

для другой разновидности многократного использования кода — а именно много-

кратного использования алгоритмов.

CLR поддерживает создание

как обобщенных ссылочных, так и обобщенных значимых типов

Необходимо отметить, что CLR размещает статические поля типа в самом

объекте-типе (см. главу 4). Следовательно, каждый закрытый тип имеет свои ста-

тические поля. Иначе говоря, статические поля, определенные в объекте List<T>,

не будут совместно использоваться объектами List<DateTime> и List<String>,

потому что у каждого объекта закрытого типа есть свои статические поля.

internal class Node {

protected Node m\_next;

public Node(Node next) {

m\_next = next;

}

}

internal sealed class TypedNode<T> : Node {

public T m\_data;

public TypedNode(T data) : this(data, null) {

}

public TypedNode(T data, Node next) : base(next) {

m\_data = data;

}

public override String ToString() {

Инфраструктура обобщений **313**

return m\_data.ToString() +

((m\_next != null) ? m\_next.ToString() : String.Empty);

}

}

Теперь можно написать код для создания связного списка с разными типами

данных у разных узлов. Код будет выглядеть примерно так:

private static void DifferentDataLinkedList() {

Node head = new TypedNode<Char>('.');

head = new TypedNode<DateTime>(DateTime.Now, head);

head = new TypedNode<String>("Today is ", head);

Console.WriteLine(head.ToString());

}

internal sealed class DateTimeList : List<DateTime> {

// Здесь никакой код добавлять не нужно!

}

using DateTimeList = System.Collections.Generic.List<System.DateTime>;

internal sealed class SomeType {

private static void SomeMethod () {

// Компилятор определяет, что dtl имеет тип

// System.Collections.Generic.List<System.DateTime>

var dtl = List<DateTime>();

...

}

# **Разрастание кода**

Во-первых, если метод вызывается для конкретного

аргумента типа и позже он вызывается опять с тем же аргументом типа, CLR ком-

пилирует код для такого сочетания «метод + тип» только один раз. Поэтому, если

List<DateTime> используется в двух совершенно разных сборках (загруженных

в один домен приложений), CLR компилирует методы для List<DateTime> всего

один раз. Это существенно сокращает степень разрастания кода.

Обобщенные делегаты

Например, обобщенный делегат определяется следующим образом:

public delegate TReturn CallMe<TReturn, TKey, TValue>(

TKey key, TValue value);

Компилятор превращает его в класс, который на логическом уровне выглядит так:

public sealed class CallMe<TReturn, TKey, TValue> : MulticastDelegate {

public CallMe(Object object, IntPtr method);

public virtual TReturn Invoke(TKey key, TValue value);

public virtual IAsyncResult BeginInvoke(TKey key, TValue value,

AsyncCallback callback, Object object);

public virtual TReturn EndInvoke(IAsyncResult result);

}

# **Контравариантные и ковариантные аргументы-типы**

# **в делегатах и интерфейсах**

Каждый из параметров-типов обобщенного делегата должен быть помечен как

ковариантный или контравариантный. Это позволяет вам осуществлять приве-

дение типа переменной обобщенного делегата к *тому же типу делегата* с другим

параметром-типом. Параметры-типы могут быть:

. **Инвариантными.** Параметр-тип не может изменяться. Пока в этой главе при-

водились только инвариантные параметры-типы.

. **Контравариантными.** Параметр-тип может быть преобразован от класса к классу,

производному от него. В языке C# контравариантный тип обозначается ключе-

вым словом in. Контравариантный параметр-тип может появляться только во

входной позиции, например, в качестве аргументов метода.

. **Ковариантными.** Аргумент-тип может быть преобразован от класса к одному

из его базовых классов. В языке С# ковариантный тип обозначается ключевым

словом out. Ковариантный параметр обобщенного типа может появляться только

в выходной позиции, например, в качестве возвращаемого значения метода.

public delegate TResult Func<in T, out TResult>(T arg);

Здесь параметр-тип T помечен словом in, делающим его контравариантным,

а параметр-тип TResult помечен словом out, делающим его ковариантным.

Пусть объявлена следующая переменная:

Func<Object, ArgumentException> fn1 = null;

Ее можно привести к типу Func с другими параметрами-типами:

Func<String, Exception> fn2 = fn1; // Явного приведения типа не требуется

Exception e = fn2("");

Это говорит о том, что fn1 ссылается на функцию, которая получает Object

и возвращает ArgumentException. Переменная fn2 пытается сослаться на метод,

который получает String и возвращает Exception. Так как мы можем передать

String методу, которому требуется тип Object (тип String является производным

от Object), а результат метода, возвращающего ArgumentException, может интер-

претироваться как Exception (тип ArgumentException является производным от

Exception), представленный здесь программный код откомпилируется, а на этапе

компиляции будет сохранена безопасность типов.

# **Обобщенные методы**

internal sealed class GenericType<T> {

private T m\_value;

public GenericType(T value) { m\_value = value; }

public TOutput Converter<TOutput>() {

TOutput result = (TOutput) Convert.ChangeType(m\_value, typeof(TOutput));

*продолжение* 

**320** Глава 12. Обобщения

return result;

}

}

# **Обобщенные методы и выведение типов**

private static void CallingSwapUsingInference() {

Int32 n1 = 1, n2 = 2;

Swap(ref n1, ref n2); // Вызывает Swap<Int32>

String s1 = "Aidan";

Object s2 = "Grant";

Swap(ref s1, ref s2); // Ошибка, невозможно вывести тип

}

Тип может определять несколько методов таким образом, что один из них будет

принимать конкретный тип данных, а другой — обобщенный параметр-тип, как

в этом примере:

private static void Display(String s) {

Console.WriteLine(s);

}

private static void Display<T>(T o) {

Display(o.ToString()); // Вызывает Display(String)

}

Метод Display можно вызвать несколькими способами:

Display("Jeff"); // Вызывает Display(String)

Display(123); // Вызывает Display<T>(T)

Display<String>("Aidan"); // Вызывает Display<T>(T)

В первом случае компилятор может вызвать либо метод Display, принимающий

String, либо обобщенный метод Display (заменяя T типом String). Но компилятор

C# всегда выбирает явное, а не обобщенное соответствие, поэтому генерирует вызов

**322** Глава 12. Обобщения

необобщенного метода Display, получающего String. Во втором случае компилятор

не может вызвать необобщенный метод Display, получающий String, поэтому он

вызывает обобщенный метод Display. Кстати, очень удачно, что компилятор всегда

выбирает более явное соответствие. Ведь если бы компилятор выбрал обобщенный

метод Display, тот вызвал бы метод ToString, возвращающий String, что привело

бы к бесконечной рекурсии.

# Generic ограничение

Ограничение сужает перечень типов, которые можно передать в обобщенном

аргументе, и расширяет возможности по работе с этими типами. Вот новый вариант

метода Min, который задает ограничение (выделено полужирным шрифтом):

public static T Min<T>(T o1, T o2) **where T** : **IComparable<T>** {

if (o1.CompareTo(o2) < 0) return o1;

return o2;

}

Маркер where в C# сообщает компилятору, что указанный в T тип должен реа-

лизовывать обобщенный интерфейс IComparable того же типа (T). Благодаря этому

ограничению компилятор разрешает методу вызвать метод CompareTo, потому что

последний определен в интерфейсе IComparable<T>.

# Запрет на изменения ограничений

internal class Base {

public virtual void M<T1, T2>()

where T1 : struct

where T2 : class {

}

}

internal sealed class Derived : Base {

public override void M<T3, T4>()

where T3 : EventArgs // Ошибка

where T4 : class // Ошибка

{ }

}

При компиляции этого кода появится сообщение об ошибке (ошибка CS0460:

ограничения для методов интерфейсов с переопределением и явной реализацией

наследуются от базового метода и поэтому не могут быть заданы явно):

Error CS0460: Constraints for override and explicit interface implementation

methods are inherited from the base method, so they cannot be

specified directly

Если из метода M<T3, T4> класса Derived убрать две строки where, код успешно

компилируется. Заметьте: разрешается переименовывать параметры типа (в этом

примере имя T1 изменено на T3, а Т2 — на Т4), но изменять (и даже задавать) огра-

ничения нельзя.

# Типы ограничений

Теперь поговорим о различных типах ограничений, которые компилятор и CLR

позволяют применять к параметрам типа. К параметру-типу могут применяться

следующие ограничения: *основное* (primary), *дополнительное* (secondary) и/или

*ограничение конструктора* (constructor constraint). Речь о них идет в следующих

трех разделах.

The struct constraint promises the compiler that a specified type argument will be a value type.

Any value type, including enumerations, satisfies this constraint. However, the compiler and the CLR

treat any System.Nullable<T> value type as a special type, and nullable types do not satisfy this

constraint. The reason is because the Nullable<T> type constrains its type parameter to struct,

and the CLR wants to prohibit a recursive type such as Nullable<Nullable<T>>. Nullable types are

discussed in Chapter 19, “Nullable Value Types.”

Here is an example class that constrains its type parameter by using the struct constraint.

internal sealed class PrimaryConstraintOfStruct<T> where T : struct {

public static T Factory() {

// Allowed because all value types implicitly

// have a public, parameterless constructor

return new T();

}

}

# Secondary Constraints

A type parameter can specify zero or more secondary constraints where a secondary constraint represents

an interface type. When specifying an interface type constraint, you are promising the compiler

that a specified type argument will be a type that implements the interface. And because you

can specify multiple interface constraints, the type argument must specify a type that implements all

of the interface constraints (and all of the primary constraints too, if specified). Chapter 13 discusses

interface constraints in detail.

private static List<TBase> ConvertIList<T, TBase>(IList<T> list)

where T : TBase {

List<TBase> baseList = new List<TBase>(list.Count);

for (Int32 index = 0; index < list.Count; index++) {

baseList.Add(list[index]);

}

return baseList;

}

Constructor Constraints

Here is an example class

that constrains its type parameter by using the constructor constraint.

internal sealed class ConstructorConstraint<T> where T : new() {

public static T Factory() {

// Allowed because all value types implicitly

// have a public, parameterless constructor and because

// the constraint requires that any specified reference

// type also have a public, parameterless constructor

return new T();

}

}

Casting a Generic Type Variable

Casting a generic type variable to another type is illegal unless you are casting to a type compatible

with a constraint.

private static void CastingAGenericTypeVariable1<T>(T obj) {

Int32 x = (Int32) obj; // Error

String s = (String) obj; // Error

}

The compiler issues an error on both lines above because T could be any type, and there is no

guarantee that the casts will succeed. You can modify this code to get it to compile by casting to

Object first.

private static void CastingAGenericTypeVariable2<T>(T obj) {

Int32 x = (Int32) (Object) obj; // No error

String s = (String) (Object) obj; // No error

}

Although this code will now compile, it is still possible for the CLR to throw an InvalidCastException

at run time.

Setting a Generic Type Variable to a Default Value

Setting a generic type variable to null is illegal unless the generic type is constrained to a reference

type.

private static void SettingAGenericTypeVariableToNull<T>() {

T temp = null; // CS0403 – Cannot convert null to type parameter 'T' because it could

// be a non-nullable value type. Consider using 'default(T)' instead

}

Because T is unconstrained, it could be a value type, and setting a variable of a value type to null

is not possible. If T were constrained to a reference type, setting temp to null would compile and run

just fine.

Microsoft’s C# team felt that it would be useful to give developers the ability to set a variable to a

default value. So the C# compiler allows you to use the default keyword to accomplish this.

Comparing a Generic Type Variable with null

Comparing a generic type variable to null by using the == or != operator is legal regardless of

whether the generic type is constrained.

private static void ComparingAGenericTypeVariableWithNull<T>(T obj) {

if (obj == null) { /\* Never executes for a value type \*/ }

}

Because T is unconstrained, it could be a reference type or a value type

Comparing Two Generic Type Variables with Each Other

Comparing two variables of the same generic type is illegal if the generic type parameter is not

known to be a reference type.

private static void ComparingTwoGenericTypeVariables<T>(T o1, T o2) {

if (o1 == o2) { } // Error

}

Using Generic Type Variables as Operands

Finally, it should be noted that there are a lot of issues about using operators with generic type

operands. In Chapter 5, I talked about C# and how it handles its primitive types: Byte, Int16, Int32,

Int64, Decimal, and so on. In particular, I mentioned that C# knows how to interpret operators (such

as +, -, \*, and /) when applied to the primitive types. Well, these operators can’t be applied to

variables of a generic type because the compiler doesn’t know the type at compile time. This means

that you can’t use any of these operators with variables of a generic type. So it is impossible to write

a mathematical algorithm that works on an arbitrary numeric data type. Here is an example of a

generic method that I’d like to write.

private static T Sum<T>(T num) where T : struct {

T sum = default(T) ;

for (T n = default(T) ; n < num ; n++)

sum += n;

return sum;

}

I’ve done everything possible to try to get this method to compile. I’ve constrained T to struct,

and I’m using default(T) to initialize sum and n to 0. But when I compile this code, I get the following

three errors.

■■ error CS0019: Operator '<' cannot be applied to operands of type 'T'

and 'T'

■■ error CS0023: Operator '++' cannot be applied to operand of type 'T'

■■ error CS0019: Operator '+=' cannot be applied to operands of type 'T'

and 'T'

Interfaces

As mentioned in the previous section, an interface is a named set of method signatures. Note that

interfaces can also define events, parameterless properties, and parameterful properties (indexers in

C#) because all of these are just syntax shorthands that map to methods anyway, as shown in previous

chapters. However, an interface cannot define any constructor methods. In addition, an interface

is not allowed to define any instance fields.

Although the CLR does allow an interface to define static methods, static fields, constants, and

static constructors, a Common Language Specification (CLS)–compliant interface must not have any

of these static members because some programming languages aren’t able to define or access them.

In fact, C# prevents an interface from defining any of these static members.

public sealed class Point : IComparable<Point> {

private Int32 m\_x, m\_y;

public Point(Int32 x, Int32 y) {

m\_x = x;

m\_y = y;

}

// This method implements IComparable<T>.CompareTo() for Point

public Int32 CompareTo(Point other) {

return Math.Sign(Math.Sqrt(m\_x \* m\_x + m\_y \* m\_y)

- Math.Sqrt(other.m\_x \* other.m\_x + other.m\_y \* other.m\_y));

}

public override String ToString() {

return String.Format("({0}, {1})", m\_x, m\_y);

}

}

public static class Program {

public static void Main() {

Point[] points = new Point[] {

new Point(3, 3),

new Point(1, 2),

};

// Here is a call to Point's IComparable<T> CompareTo method

if (points[0].CompareTo(points[1]) > 0) {

Point tempPoint = points[0];

points[0] = points[1];

points[1] = tempPoint;

}

Console.WriteLine("Points from closest to (0, 0) to farthest:");

foreach (Point p in points)

Console.WriteLine(p);

}

}

The C# compiler requires that a method that implements an interface be marked as public. The

CLR requires that interface methods be marked as virtual. If you do not explicitly mark the method

as virtual in your source code, the compiler marks the method as virtual and sealed; this prevents a

derived class from overriding the interface method. If you explicitly mark the method as virtual, the

compiler marks the method as virtual (and leaves it unsealed); this allows a derived class to override

the interface method.

# **Implicit and Explicit Interface Method Implementations**

# **(What’s Happening Behind the Scenes)**

public sealed class Program {

public static void Main() {

SimpleType st = new SimpleType();

// This calls the public Dispose method implementation

st.Dispose();

// This calls IDisposable's Dispose method implementation

IDisposable d = st;

d.Dispose();

}

}

In the first call to Dispose, the Dispose method defined by SimpleType is called. Then I define

a variable, d, which is of the IDisposable interface type. I initialize the d variable to refer to

the SimpleType object. Now when I call d.Dispose(), I am calling the IDisposable interface’s

Dispose method. Because C# requires the public Dispose method to also be the implementation for

IDisposable’s Dispose method, the same code will execute, and, in this example, you can’t see any

observable difference.

Now, let me rewrite the preceding SimpleType so that you can see an observable difference.

internal sealed class SimpleType : IDisposable {

public void Dispose() { Console.WriteLine("public Dispose"); }

void IDisposable.Dispose() { Console.WriteLine("IDisposable Dispose"); }

}

Without changing the Main method shown earlier, if we just recompile and rerun the program, the

output will be the following.

public Dispose

IDisposable Dispose

**Generic Interfaces**

private void SomeMethod1() {

Int32 x = 1, y = 2;

IComparable c = x;

// CompareTo expects an Object; passing y (an Int32) is OK

c.CompareTo(y); // y is boxed here

// CompareTo expects an Object; passing "2" (a String) compiles

// but an ArgumentException is thrown at runtime

c.CompareTo("2");

}

private void SomeMethod2() {

Int32 x = 1, y = 2;

IComparable<Int32> c = x;

// CompareTo expects an Int32; passing y (an Int32) is OK

c.CompareTo(y); // y is not boxed here

// CompareTo expects an Int32; passing "2" (a String) results

// in a compiler error indicating that String cannot be cast to an Int32

c.CompareTo("2"); // Error

}

Note The FCL defines non-generic and generic versions of the IComparable, ICollection,

IList, and IDictionary interfaces, as well as some others. If you are defining a type, and

you want to implement any of these interfaces, you should typically implement the generic

versions of these interfaces. The non-generic versions are in the FCL for backward compatibility

to work with code written before the .NET Framework supported generics. The nongeneric

versions also provide users a way of manipulating the data in a more general, less

type-safe fashion.

using System;

// This class implements the generic IComparable<T> interface twice

public sealed class Number: IComparable<Int32>, IComparable<String> {

private Int32 m\_val = 5;

// This method implements IComparable<Int32>'s CompareTo

public Int32 CompareTo(Int32 n) {

return m\_val.CompareTo(n);

}

// This method implements IComparable<String>'s CompareTo

public Int32 CompareTo(String s) {

return m\_val.CompareTo(Int32.Parse(s));

}

}

public static class Program {

public static void Main() {

Number n = new Number();

// Here, I compare the value in n with an Int32 (5)

IComparable<Int32> cInt32 = n;

Int32 result = cInt32.CompareTo(5);

// Here, I compare the value in n with a String ("5")

IComparable<String> cString = n;

result = cString.CompareTo("5");

}

}

**Implementing Multiple Interfaces That Have the Same Method**

**Name and Signature**

Occasionally, you might find yourself defining a type that implements multiple interfaces that define

methods with the same name and signature. For example, imagine that there are two interfaces defined

as follows.

public interface IWindow {

Object GetMenu();

}

public interface IRestaurant {

Object GetMenu();

}

Let’s say that you want to define a type that implements both of these interfaces. You’d have to

implement the type’s members by using explicit interface method implementations as follows.

// This type is derived from System.Object and

// implements the IWindow and IRestaurant interfaces.

public sealed class MarioPizzeria : IWindow, IRestaurant {

// This is the implementation for IWindow's GetMenu method.

Object IWindow.GetMenu() { ... }

// This is the implementation for IRestaurant's GetMenu method.

Object IRestaurant.GetMenu() { ... }

// This (optional method) is a GetMenu method that has nothing

// to do with an interface.

public Object GetMenu() { ... }

}

MarioPizzeria mp = new MarioPizzeria();

// This line calls MarioPizzeria's public GetMenu method

mp.GetMenu();

// These lines call MarioPizzeria's IWindow.GetMenu method

IWindow window = mp;

window.GetMenu();

// These lines call MarioPizzeria's IRestaurant.GetMenu method

IRestaurant restaurant = mp;

restaurant.GetMenu();

**Improving Compile-Time Type Safety with Explicit Interface**

**Method Implementations**

internal struct SomeValueType : IComparable {

private Int32 m\_x;

public SomeValueType(Int32 x) { m\_x = x; }

public Int32 CompareTo(Object other) {

return(m\_x - ((SomeValueType) other).m\_x);

}

}

Using SomeValueType, I can now write the following code.

public static void Main() {

SomeValueType v = new SomeValueType(0);

Object o = new Object();

Int32 n = v.CompareTo(v); // Undesired boxing

n = v.CompareTo(o); // InvalidCastException

}

There are two characteristics of this code that are not ideal.

■■ **Undesired boxing** When v is passed as an argument to the CompareTo method, it must be

boxed because CompareTo expects an Object.

■■ **The lack of type safety** This code compiles, but an InvalidCastException is thrown

inside the CompareTo method when it attempts to cast o to SomeValueType.

Both of these issues can be fixed by using EIMIs. Here’s a modified version of SomeValueType that

has an EIMI added to it.

internal struct SomeValueType : IComparable {

private Int32 m\_x;

public SomeValueType(Int32 x) { m\_x = x; }

public Int32 CompareTo(SomeValueType other) {

return(m\_x - other.m\_x);

}

// NOTE: No public/private used on the next line

Int32 IComparable.CompareTo(Object other) {

return CompareTo((SomeValueType) other);

}

}