ГЛАВА 6

Перегрузка операций

Язык C# позаимствовал возможность перегрузки операций из C++. Точно так же, как перегружаются методы, вы можете перегружать операции, подобные +, –, * и т.д. В дополнение к перегрузке арифметических операций вы можете также создавать собственные операции для преобразования от одного типа к другому. Вы можете перегружать и другие операции, позволяя использовать объекты в булевских выражениях проверки.

Можете — не значит должны

Перегрузка операций может сделать использование некоторых классов и структур более естественным. Однако перегрузка операций, выполненная неаккуратно, может значительно ухудшить читабельность кода и снизить возможность его понимания. Вы должны быть внимательными, рассматривая семантику операций с типами. Не вводите того, что трудно будет расшифровать. Всегда ориентируйтесь на создание более читабельного кода — и не только для той счастливой души, которая будет хлопать глазами, изучая ваш код, но также и для себя. Случалось ли вам когда-нибудь смотреть на код и думать: "Кто в здравом уме мог написать такую чушь?", а потом обнаруживать, что это были вы? Со мной такое было...

Другая причина не перегружать операции заключается в том, что не все языки .NET поддерживают перегруженные операции, потому что они не являются частью CLS. От языков, ориентированных на CLI, не требуется поддержка перегруженных операций. Так, например Visual Basic 2005 был первой .NET-версией языка для поддержки перегрузки операций. Поэтому важно, чтобы ваши перегруженные операции были синтаксическими сокращениями для функциональности, представленной другими методами, выполняющими те же операции, и которые можно было бы вызвать из других CLS-совместимых языков. Фактически, я рекомендую проектировать типы так, как если бы перегруженные операции не существовали. Тогда позже при желании вы сможете добавить их, просто вызывая уже существующие методы того же семантического назначения.

Типы и форматы перегруженных операций

Вы определяете перегруженные операции как общедоступные статические методы в классах, для расширения которых они предназначены. В зависимости от типа перегружаемой операции, такие методы могут принимать один или два параметра, и всегда возвращают значение. Для всех операций за исключением операций преобразования типом одного из параметров должен быть тип включаю-

щего метод класса. Например, не имеет смысла перегружать операцию + в классе Complex, если он будет складывать вместе два значения double. К тому же, как вы вскоре убедитесь, это и невозможно.

Типичная операция + для класса Complex может выглядеть так:

```
public static Complex operator+( Complex lhs, Complex rhs )
```

Даже несмотря на то, что этот метод складывает два экземпляра Complex вместе, чтобы произвести третий экземпляр Complex, ничто не запрещает сделать один из параметров типа double, чтобы, таким образом, прибавлять double к экземпляру Complex. Как именно вы будете прибавлять double к экземпляру Complex — ваше дело. Вообще синтаксис перегрузки операций следует приведенному шаблону, где + заменяется нужной операцией, и конечно, некоторые операции принимают только один параметр.

На заметку! При сравнении операций C# с операциями C++ обратите внимание, что объявление операции C# больше похоже на прием объявления дружественных функций для реализации операций в C++, поскольку операции C# не являются методами экземпляра.

Существуют всего три группы перегружаемых операций. Унарные операции принимают только один параметр. К знакомым вам унарным операциям относятся ++ и --. Бинарные операции, как следует из их названия, принимают два параметра и включают знакомые математические операции, такие как +, –, / и *, а также операции сравнения. И, наконец, операции преобразования определяют пользовательские преобразования типов. Они могут иметь либо операнд, либо возвращаемое значение того же типа, что и класс или структура, в которой они объявлены.

Несмотря на то что операции являются статическими и общедоступными, и как таковые, наследуются производными классами, методы операций должны иметь как минимум, один параметр в их объявлении, совпадающий с *включающим* типом, что делает невозможным точное соответствие методов операций производных типов сигнатуре методов операций базового класса. Например, следующее объявление неправильно:

```
public class Apple
{
    public static Apple operator+( Apple rhs, Apple lhs ) {
        // Метод не делает ничего, существует только для примера.
        return rhs;
    }
}
public class GreenApple : Apple
{
    // НЕВЕРНО!!! Не скомпилируется.
    public static Apple operator+( Apple rhs, Apple lhs ) {
        // Метод не делает ничего, существует только для примера.
        return rhs;
    }
}
```

Если вы попытаетесь скомпилировать предыдущий код, то получите следующую ошибку компиляции:

```
error CS0563: One of the parameters of a binary operator must be the
containing type
ошибка CS0563: Один из параметров бинарной операции должен быть
включающего типа
```

Операции не должны изменять свои операнды

Вы уже знаете, что методы операций являются статическими. Поэтому настоятельно рекомендуется (читай: требуется), чтобы вы не изменяли переданные им параметры. Вместо этого вы должны создавать новый экземпляр возвращаемого типа и возвращать его, как результат операции. Неизменяемые структуры и классы вроде `System.String` — блестящие кандидаты для реализации пользовательских операций. Такое поведение естественно для булевских операций, которые обычно возвращают тип, отличающийся от типов переданных в параметрах.

На заметку! "Минуточку!" — скажут некоторые из вас, принадлежащие к сообществу C++ — "а как же тогда реализовать постфиксные и префиксные операции `++` и `--` без изменения операнда?". Все операции C# являются статическими, и к ним относятся также и постфиксные, и префиксные операции, в то время как в C++ они представлены методами экземпляров, модифицирующими экземпляр объекта через указатель `this`. Красота подхода C# заключается в том, что вам не нужно беспокоиться о реализации двух различных версий операции `++`, чтобы поддерживать префиксную и постфиксную его формы, как в C++. Компилятор выполняет задачу создания временных копий объекта для обработки отличия в поведении между префиксом и постфиксом. Это — еще одна причина того, что операции могут возвращать новые экземпляры, никогда не модифицируя состояния самих операндов. Если вы не будете следовать такой практике, то обречете себя на некоторые серьезные неприятности при отладке.

Имеет ли значение порядок параметров?

Предположим, вы создаете `struct` для представления простых комплексных чисел — скажем, структуру `Complex` — и вам нужно складывать вместе экземпляры `Complex`. Было бы также удобно иметь возможность прибавлять простые значения `double` к экземпляру `Complex`. Добавление такой функциональности — не проблема, поскольку вы можете перегрузить метод операции `+` так, чтобы один параметр был `Complex`, а другой — `double`. Это объявление могло бы выглядеть примерно так:

```
static public Complex operator+( Complex lhs, double rhs )
```

Имея такую операцию, объявленную и определенную в структуре `Complex`, вы можете писать код вроде следующего:

```
Complex cpx1 = new Complex( 1.0, 2.0 );
Complex cpx2 = cpx1 + 20.0;
```

Это избавляет вас от необходимости создавать дополнительный экземпляр `Complex`, состоящий только из реальной части, равной `20.0`, чтобы добавить ее к `cpx1`. Однако представьте, что вы хотите иметь возможность менять местами операнды и делать что-нибудь вроде следующего:

```
Complex cpx2 = 20.0 + cpx1;
```

Если вы хотите поддерживать разный порядок операндов разных типов, то для этого должны предусмотреть разные перегрузки для операции. Если вы перегру-

жаете бинарную операцию, используя разные типы параметров, то можете создавать зеркальные перегрузки — т.е. другой метод операции, который просто меняет местами параметры.

Перегрузка операции сложения

Давайте рассмотрим краткий пример структуры `Complex`, которая не претендует на звание исчерпывающей реализации, а просто демонстрирует перегрузку операций. На протяжении всей главы я буду отталкиваться от этого примера, и добавлять к нему дополнительные операции:

```
using System;
public struct Complex
{
    public Complex( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    static public Complex Add( Complex lhs,
                               Complex rhs ) {
        return new Complex( lhs.real + rhs.real,
                            lhs.imaginary + rhs.imaginary );
    }
    static public Complex Add( Complex lhs,
                               double rhs ) {
        return new Complex( rhs + lhs.real,
                            lhs.imaginary );
    }
    public override string ToString() {
        return String.Format( "({0}, {1})",
                              real,
                              imaginary );
    }
    static public Complex operator+( Complex lhs,
                                     Complex rhs ) {
        return Add( lhs, rhs );
    }
    static public Complex operator+( double lhs,
                                     Complex rhs ) {
        return Add( rhs, lhs );
    }
    static public Complex operator+( Complex lhs,
                                     double rhs ) {
        return Add( lhs, rhs );
    }
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main() {
        Complex cpx1 = new Complex( 1.0, 3.0 );
        Complex cpx2 = new Complex( 1.0, 2.0 );
```

```
            Complex cpx3 = cpx1 + cpx2;
            Complex cpx4 = 20.0 + cpx1;
            Complex cpx5 = cpx1 + 25.0;
            Console.WriteLine( "cpx1 == {0}", cpx1 );
            Console.WriteLine( "cpx2 == {0}", cpx2 );
            Console.WriteLine( "cpx3 == {0}", cpx3 );
            Console.WriteLine( "cpx4 == {0}", cpx4 );
            Console.WriteLine( "cpx5 == {0}", cpx5 );
        }
    }
```

Обратите внимание, что как рекомендуется, перегруженные методы операции вызывают методы, выполняющие саму операцию. Фактически это очень упрощает поддержку обоих последовательностей операндов операции + для типа `Complex`.

Совет. Если вы абсолютно уверены, что ваш тип будет использоваться только в среде C# или с языком, поддерживающим перегруженные операции, то можете пренебречь этим правилом и просто написать перегруженные операции.

Операции, допускающие перегрузку

Давайте вкратце перечислим операции, которые вы можете перегружать. Унарные операции, бинарные операции и операции преобразования — три главных типа операций. Невозможно перечислить здесь все операции преобразования, поскольку их набор неограничен. Вдобавок вы можете использовать одну тернарную операцию — знакомую вам `?:` — для условных операторов, но не можете перегружать ее напрямую. Далее, в разделе "Булевские операции" я расскажу, как можно "поиграть" с тернарной операцией. В табл. 6.1 перечислены все перегружаемые операции, за исключением операций преобразования.

Таблица 6.1. Унарные и бинарные операции

Унарные операции	Бинарные операции
+	+
–	–
!	*
~	/
++	%
--	&
true и false	|
	^
	<<
	>>
	== и !=
	> и <
	>= и <=

Операции сравнения

Бинарные операции сравнения == и !=, < и >, а также >= и <= должны быть реализованы парами. Конечно, это имеет совершенно прямой смысл, поскольку я сомневаюсь, чтобы в каком-то случае вы захотели разрешить пользователям применять операцию ==, но не !=. Кроме того, если ваш тип позволяет упорядочивание через реализацию интерфейса `IComparable` или его обобщенного аналога `IComparable<T>`, то имеет смысл реализовать все операции сравнения. Их реализация тривиальна, если следовать каноническому руководству, приведенному в главах 4 и 13, соответственно переопределяя `Equals` и `GetHashCode` и реализуя `IComparable` (и, необязательно, `IComparable<T>`). Учитывая это, перегрузка операций просто требует, чтобы вы вызывали эти реализации. Рассмотрим модифицированную форму примера `Complex`, следующую этому шаблону для реализации всех операций сравнения:

```
using System;
public struct Complex : IComparable,
           IEquatable<Complex>,
           IComparable<Complex>
{
    public Complex( double real, double img ) {
        this.real = real;
        this.img = img;
    }
    // Перегрузка System.Object
    public override bool Equals( object other ) {
        bool result = false;
        if( other is Complex ) {
           result = Equals( (Complex) other );
        }
        return result;
    }
    // Версия, безопасная к типам
    public bool Equals( Complex that ) {
        return (this.real == that.real &&
                  this.img == that.img);
    }
    // Должен быть перегружен, если перегружен Object.Equals()
    public override int GetHashCode() {
        return (int) this.Magnitude;
    }
    // Версия, безопасная к типам
    public int CompareTo( Complex that ) {
        int result;
        if( Equals( that ) ) {
           result = 0;
        } else if( this.Magnitude > that.Magnitude ) {
           result = 1;
        } else {
           result = -1;
        }
        return result;
    }
```

```
    // Реализация IComparable
    int IComparable.CompareTo( object other ) {
        if( !(other is Complex) ) {
           throw new ArgumentException( "Неверное сравнение" );
        }
        return CompareTo( (Complex) other );
    }
    // Перегрузка System.Object
    public override string ToString() {
        return String.Format( "({0}, {1})",
                              real,
                              img );
    }
    public double Magnitude {
        get {
           return Math.Sqrt( Math.Pow(this.real, 2) +
                  Math.Pow(this.img, 2) );
        }
    }
    // Перегруженные операции
    public static bool operator==( Complex lhs, Complex rhs ) {
        return lhs.Equals( rhs );
    }
    public static bool operator!=( Complex lhs, Complex rhs ) {
        return !lhs.Equals( rhs );
    }
    public static bool operator<( Complex lhs, Complex rhs ) {
        return lhs.CompareTo( rhs ) < 0;
    }
    public static bool operator>( Complex lhs, Complex rhs ) {
        return lhs.CompareTo( rhs ) > 0;
    }
    public static bool operator<=( Complex lhs, Complex rhs ) {
        return lhs.CompareTo( rhs ) <= 0;
    }
    public static bool operator>=( Complex lhs, Complex rhs ) {
        return lhs.CompareTo( rhs ) >= 0;
    }
    // Прочие методы пропущены для ясности
    private double real;
    private double img;
}
public class EntryPoint
{
    static void Main() {
        Complex cpx1 = new Complex( 1.0, 3.0 );
        Complex cpx2 = new Complex( 1.0, 2.0 );
        Console.WriteLine( "cpx1 = {0}, cpx1.Magnitude = {1}",
        cpx1, cpx1.Magnitude );
        Console.WriteLine( "cpx2 = {0}, cpx2.Magnitude = {1}\n",
        cpx2, cpx2.Magnitude );
        Console.WriteLine( "cpx1 == cpx2 ? {0}", cpx1 == cpx2 );
```

```
            Console.WriteLine( "cpx1 != cpx2 ? {0}", cpx1 != cpx2 );
            Console.WriteLine( "cpx1 < cpx2 ? {0}", cpx1 < cpx2 );
            Console.WriteLine( "cpx1 > cpx2 ? {0}", cpx1 > cpx2 );
            Console.WriteLine( "cpx1 <= cpx2 ? {0}", cpx1 <= cpx2 );
            Console.WriteLine( "cpx1 >= cpx2 ? {0}", cpx1 >= cpx2 );
        }
    }
```

Обратите внимание, что методы операций просто вызывают методы, реализующие `Equals` и `CompareTo`. Кроме того, я следую рекомендации о предоставлении версий, безопасных к типам, двух методов через реализацию `IComparable<Complex>` и `IEquatable<Complex>`, поскольку `Complex` — тип значения, и я хочу по возможности избежать упаковки[1]. Дополнительно я явно реализовал метод `IComparable.CompareTo`, чтобы предоставить компилятору большой безопасный к типам молоток, чтобы затруднить пользователям возможность нечаянно вызвать неверную версию. Всякий раз, когда у вас есть возможность использовать систему контроля типов компилятора, чтобы выявить ошибки на этапе компиляции, а не на этапе выполнения, вы должны воспользоваться ею. Если бы я не реализовал явно `Icomparable.CompareTo`, то компилятор спокойно пропустит оператор, где я пытаюсь сравнить экземпляр `Apple` с экземпляром `Complex`. Конечно, вы ожидаете исключения `InvalidCastException` во время выполнения, если попытаетесь сделать нечто настолько глупое, так что всегда предпочитайте ошибки времени компиляции ошибкам времени выполнения.

Операции преобразования

Операции преобразования, как следует из их названия, являются операциями, преобразующими объекты одного типа в объекты другого типа. Операции преобразования могут допускать неявные преобразования наряду с явными. Неявные преобразования выполняются при простом присваивании, в то время как явные требуют знакомого синтаксиса приведений, где целевой тип указывается в скобках, предшествующих экземпляру, из которого присваивается значение.

На неявные операции накладывается одно важное ограничение. Стандарт C# требует, чтобы неявные операции не генерировали исключений, чтобы они всегда гарантированно завершались успешно, без потери информации. Если вы не можете соблюсти это требование, то ваше преобразование должно быть явным. Например, при преобразовании одного типа в другой всегда имеется возможность утери информации, если целевой тип не настолько выразительный, как исходный. Рассмотрим преобразование `long` в `short`. Ясно, что информация может быть утеряна, если значение `long` окажется больше максимально допустимого для типа `short`. Даже несмотря на то, что по умолчанию исключение в случае усечения не генерируется, иногда стоит сгенерировать исключение во время выполнения. Такое преобразование должно быть явным и требовать от пользователя использования синтаксиса приведения. Теперь представьте, что вы идете в противоположном направлении, преобразуя `short` в `long`. Такое преобразование всегда пройдет успешно, поэтому оно может быть неявным.

[1] Я подробно описываю эти рекомендации в главе 5, в разделе, озаглавленном “Явная реализация интерфейса с типами значений”.

На заметку! Выполнение явных преобразований от типа с большим диапазоном хранимых значений к типу с меньшим диапазоном может дать ошибку усечения, если исходное значение окажется слишком большим, чтобы быть представленным в маленьком типе. Например, если вы выполните явное приведение `long` к `short`, то можете вызвать ситуацию переполнения. По умолчанию ваш компилированный код будет молча выполнять усечение. Если вы откомпилируете код с опцией компилятора `/checked+`, то при попытке явного преобразования `long` к `short` будет генерироваться исключение `System.OverflowException`. Я рекомендую всегда выполнять сборку с включенной опцией `/checked+`.

Посмотрим, какие операции преобразования вы должны предусмотреть для `Complex`. Я могу представить, по крайней мере, один определенный случай — преобразование из `double` в `Complex`. Несомненно, такое преобразование должно быть неявным. Другой вариант — `Complex` в `double` — требует явного преобразования. (Поскольку приведение `Complex` к `double` все равно не имеет смысла и показано здесь только для примера, вы можете возвращать общую величину (magnitude), а не просто реальную часть комплексного числа, выполняя приведение к `double`.) Рассмотрим пример реализации операций приведения:

```
using System;
public struct Complex
{
    public Complex( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    // Переопределение System.Object
    public override string ToString() {
        return String.Format( "({0}, {1})", real, imaginary );
    }
    public double Magnitude {
        get {
           return Math.Sqrt( Math.Pow(this.real, 2) +
                  Math.Pow(this.imaginary, 2) );
        }
    }
    public static implicit operator Complex( double d ) {
        return new Complex( d, 0 );
    }
    public static explicit operator double( Complex c ) {
        return c.Magnitude;
    }
    // Прочие методы пропущены для ясности.
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main() {
        Complex cpx1 = new Complex( 1.0, 3.0 );
        Complex cpx2 = 2.0;       // Использовать неявную операцию.
        double d = (double) cpx1; // Использовать явную операцию.
        Console.WriteLine( "cpx1 = {0}", cpx1 );
        Console.WriteLine( "cpx2 = {0}", cpx2 );
        Console.WriteLine( "d = {0}", d );
    }
}
```

Синтаксис в методе `Main` использует операции преобразования. Однако будьте осторожны при реализации операций преобразования, чтобы не создать для пользователей никаких сюрпризов или путаницы с вашими неявными преобразованиями. Сложно внести путаницу с явными операциями, когда пользователи вашего типа вынуждены применять синтаксис приведения для того, чтобы заставить их работать. В конце концов, чем можно удивить пользователя, если он должен указать тип для преобразования в круглых скобках? Но с другой стороны, невнимательное или неправильное использование неявного преобразования может стать причиной большой путаницы. Если вы напишете множество операций неявного преобразования, не имеющих семантического смысла, я гарантирую, что ваши пользователи в один прекрасный день будут весьма удивлены, когда компилятор решит выполнить преобразование, когда они вовсе этого не ожидали. Например, компилятор может выполнять неявное преобразование, пытаясь подогнать аргумент при вызове метода. Даже если операции преобразования имеют семантический смысл, они могут таить в себе ряд сюрпризов, поскольку у компилятора есть возможность молча преобразовывать экземпляры одного типа в другой, когда он сочтет это необходимым.

C# требует, чтобы вы явно писали операции неявного преобразования для определяемых вами типов[2]. Таким образом, вы не сможете нечаянно создать операцию неявного преобразования, не осознавая того, что вы делаете (как это возможно в C++). Однако для того, чтобы предоставить такие преобразования, вы должны всегда подчиняться правилам перегрузки методов. Рассмотрим случай, когда `Complex` предоставляет другую операцию явного преобразования экземпляра `Fraction` также в экземпляр `double`. Это потребует добавления в `Complex` методов со следующими сигнатурами:

```
public static explicit operator double( Complex d )
public static explicit operator Fraction( Complex f )
```

Эти два метода принимают тот же тип `Complex` и возвращают другой тип. Однако правила перегрузки четко указывают, что тип возврата не учитывается в сигнатуре метода. Если следовать этому правилу, такие два метода вносят неоднозначность, приводящую к ошибке компиляции. Но на самом деле они не вносят неоднозначности, поскольку существует специальное правило, разрешающее рассматривать тип возврата операций преобразования как часть сигнатуры. Кстати, ключевые слова `implicit` и `explicit` не входят в сигнатуру методов операций преобразования. Поэтому невозможно иметь одновременно и явную, и неявную версии одной и той же операции преобразования. Естественно, по крайней мере, один из типов в сигнатуре операции преобразования должен быть включающим типом. Не допускается типу `Complex` реализовать операцию преобразования из типа `Apples` в тип `Oranges`.

Булевские операции

Для некоторых типов имеет смысл участие в булевских выражениях проверки, таких как внутри скобок блока `if` или внутри тернарной операции `?:`. Чтобы это работало, у вас есть две альтернативы. Первая заключается в том, что вы мо-

[2] Да, я понимаю последствия моих высказываний и возможную путаницу, вызванную применением слов *явный* и *неявный*. Я явно надеюсь не запутать вас неявно...

жете реализовать две операции преобразования, известные как `operator true` и `operator false`. Вы должны реализовать эти две операции в паре, чтобы позволить комплексному числу (типу `Complex`) участвовать в булевских выражениях проверки. Рассмотрим следующую модификацию типа `Complex`, чтобы можно было использовать его в выражениях, где значение (0, 0) означает `false`, а все остальное — `true`:

```
using System;
public struct Complex
{
    public Complex( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    // Переопределение System.Object
    public override string ToString() {
        return String.Format( "({0}, {1})",
                              real,
                              imaginary );
    }
    public double Magnitude {
        get {
           return Math.Sqrt( Math.Pow(this.real, 2) +
                             Math.Pow(this.imaginary, 2) );
        }
    }
    public static bool operator true( Complex c ) {
        return (c.real != 0) || (c.imaginary != 0);
    }
    public static bool operator false( Complex c ) {
        return (c.real == 0) && (c.imaginary == 0);
    }
    // Прочие методы пропущены для ясности.
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main() {
        Complex cpx1 = new Complex( 1.0, 3.0 );
        if( cpx1 ) {
           Console.WriteLine( "cpx1 равно true" );
        } else {
           Console.WriteLine( "cpx1 равно false" );
        }
        Complex cpx2 = new Complex( 0, 0 );
        Console.WriteLine( "cpx2 равно {0}", cpx2 ? "true" : "false" );
    }
}
```

Вы можете видеть здесь две операции для применения к типу `Complex` проверок на предмет равенства `true` и `false`. Обратите внимание на синтаксис — он несколько причудлив. Он выглядит почти так же, как в операциях преобразования, за исключением того, что типом возврата является `bool`. Я не совсем понимаю, зачем это нужно, потому что вы все равно не сможете указать никакой другой тип

возврата для этих операций. Если сделать это, то компилятор немедленно сообщит, что единственный допустимый тип возврата для `operator true` и `operator false` — это `bool`. Тем не менее, тип возврата вы должны указывать. Также заметьте, что эти операции нельзя пометить как `explicit` или `implicit`, потому что они не являются операциями преобразования. Как только вы определите эти две операции в типе, вы сможете использовать экземпляры `Complex` в булевских выражениях проверки, как показано в методе `Main`.

Альтернативно вы можете реализовать преобразование к типу `bool`, чтобы получить тот же результат. Обычно вы захотите реализовать эту операцию неявно для простоты использования. Рассмотрим модифицированную форму предыдущего примера с использованием неявной операции преобразования к `bool` вместо `operator true` и `operator false`:

```
using System;
public struct Complex
{
    public Complex( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    // Переопределение System.Object
    public override string ToString() {
        return String.Format( "({0}, {1})",
                              real,
                              imaginary );
    }
    public double Magnitude {
        get {
           return Math.Sqrt( Math.Pow(this.real, 2) +
                  Math.Pow(this.imaginary, 2) );
        }
    }
    public static implicit operator bool( Complex c ) {
        return (c.real != 0) || (c.imaginary != 0);
    }
    // Прочие методы пропущены для ясности.
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main() {
        Complex cpx1 = new Complex( 1.0, 3.0 );
        if( cpx1 ) {
           Console.WriteLine( "cpx1 равно true" );
        } else {
           Console.WriteLine( "cpx1 равно false" );
        }
        Complex cpx2 = new Complex( 0, 0 );
        Console.WriteLine( "cpx2 равно {0}", cpx2 ? "true" : "false" );
    }
}
```

Конечный результат — тот же, что и в предыдущем примере. Теперь вы можете недоумевать, зачем вообще нужно реализовывать `operator true` и `operator false` вместо простой неявной операции преобразования к `bool`? Ответ связан с тем, допустимо для вашего типа преобразование в тип `bool` или нет. В последней форме, где вы реализовали неявную операцию преобразования, следующий оператор будет корректным:

```
bool f = cpx1;
```

Такое присваивание должно работать, потому что компилятор найдет операцию неявного преобразования во время компиляции и применит ее. Однако если вы смертельно устали после ночи кодирования этой строки и действительно намерены присваивать `f` значение совершенно отличной переменной, может пройти много времени прежде, чем вы найдете ошибку. Это лишь один пример того, как необдуманное применение операций неявного преобразования может привести к проблеме.

Существует следующее эмпирическое правило: предусматривайте только те операции, что действительно необходимы для выполнения работы, и не более. Если все, что вам нужно от вашего типа — в данном случае `Complex` — это возможность участия в булевских выражениях проверки, реализуйте только `operator true` и `operator false`. Не реализуйте операцию неявного преобразования в `bool`, если только вы действительно не нуждаетесь в ней. Если так случится, что она вам нужна, и вы реализовали неявное преобразование в `bool`, тогда вам не нужно реализовывать `operator true` и `operator false`, потому что они будут избыточны. Если вы представите все три, то компилятор использует операцию неявного преобразования в `bool` вместо `operator true` и `operator false`, потому что вызов первого не более эффективен, чем второго, предполагая, что вы закодировали их одинаково.

Резюме

В настоящей главе я представил некоторые полезные рекомендации по перегрузке операций, включая унарные, бинарные и операции преобразования. Перегрузка операций — одно из тех средств, что делают C# настолько мощным и выразительным языком .NET. Однако только то, что вы можете что-то делать, еще не означает, что вы должны это делать. Неправильное использование операций преобразования и неправильное определение семантики перегрузок других операций вновь и вновь становятся причиной путаницы для пользователей (а среди них — и самого разработчика типа) наряду с нежелательным поведением. Когда доходит до перегрузки операций, предусматривайте их лишь столько, сколько необходимо, и не вторгайтесь в область общей семантики разнообразных операций. Поскольку от CLS не требуется обязательная поддержка перегружаемых операций, не все языки .NET их поддерживают. Поэтому важно всегда предоставлять явно именованные методы, выполняющие ту же функциональность. Иногда такие методы уже определены в системных интерфейсах, таких как `IComparable` или `IComparable<T>`. Никогда не изолируйте функциональность только внутри перегруженных операций, если только вы не уверены на 100%, что ваш код будет использоваться языками .NET, поддерживающими перегружаемые операции.

В следующей главе я расскажу о сложностях и приемах, связанных с созданием безопасного и нейтрального по отношению к исключениям кода в .NET Framework.