Лекция 7. Полиморфизм и перегрузка методов и операций

Полиморфизм

Идея полиморфизма состоит в том, что объект базового класса способен использовать методы производного от него класса, который не существует на момент создания базового.

Вернёмся к примеру с геометрическими фигурами, рассмотренном в предыдущей лекции, в варианте, когда класс `GeometricShape` не был абстрактным, а его метод `Draw` рисовал точку. Пусть нам необходим метод, который выполняет отрисовку геометрической фигуры, независимо от её типа:

```
static void DrawShape(GeometricShape figure)
{
    figure.Draw();
}

...

Circle circle = new Circle()
{
    X = 100,
    Y = 100,
    R = 50
};
DrawShape(circle);
```

В этом примере создаётся круг — объект класса `Circle`, который передаётся методу `DrawShape`. Компилятор неявно преобразует ссылку на круг в ссылку на тип `GeometricShape`, но при этом, она продолжает ссылаться на объект класса `Circle`. В этом суть специализации типа и автоматического преобразования, сопровождающего её.

Однако, при вызове метода `Draw` в методе `DrawShape` вместо окружности будет нарисована точка. Дело в том, что параметр `figure` является имеет тип `GeometricShape`, и компилятор обращается к его методу `Draw`, поскольку методу `DrawShape` ничего не известно о классе `Circle`, равно как и о классе `Rectangle`. Как было отмечено ранее, информация о дополнительных и переопределённых членах производного класса недоступна при приведении типа ссылки на объект к базовому классу.

Для того чтобы метод `DrawShape` всё же имел возможность вызвать нужный метод в зависимости от типа переданной фигуры, необходимо определить метод `Draw` в классе `GeometricShape` как *виртуальный*, добавив к его описанию ключевое слово `virtual`, а во всех производных классах переопределить этот метод, указав ключевое слово `override`:

```
public class GeometricShape // класс, представляющий геометрическую фигуру
{
    public int X { get; set; } // координаты
    public int Y { get; set; } // фигуры

    public virtual void Draw()
    {
        // Выполнить отрисовку по умолчанию, например,
        // нарисовать точку с координатами X и Y
    }
}

public class Rectangle : GeometricShape // класс, представляющий прямоугольник
{
    public int Width { get; set; }  // ширина прямоугольника
    public int Height { get; set; } // высота прямоугольника

    public override void Draw()
    {
        // Нарисовать прямоугольник
    }
```

```
}

public class Circle : GeometricShape // класс, представляющий круг
{
    public int R { get; set; } // радиус круга

    public override void Draw()
    {
        // Нарисовать круг
    }
}

...

static void DrawShape(GeometricShape figure)
{
    figure.Draw();
}

...

GeometricShape point = new GeometricShape()
{
    X = 10,
    Y = 40
};
DrawShape(point); // будет вызван метод GeometricShape.Draw

Rectangle rect = new Rectangle()
{
    X = 50,
    Y = 20,
    Width = 200,
    Height = 100
};
DrawShape(rect); // будет вызван метод Rectangle.Draw

Circle circle = new Circle()
{
    X = 100,
    Y = 100,
    R = 50
};
DrawShape(circle); // будет вызван метод Circle.Draw
```

Это пример полиморфизма в действии. Метод DrawShape не интересует, какой конкретный тип фигуры представляет переданный ему объект. То, с чем он имеет дело — это GeometricShape, а Rectangle и Circle являются типом GeometricShape. Вот почему наследование иногда называют отношением «is—a» («является»).

Теперь можно передавать любой экземпляр объекта, производного от GeometricShape. После того, как построенное программное обеспечение будет упаковано в коробку и помечено наклейкой «Версия 1», некто может потом заняться версией 2 и определить новые фигуры, унаследованные от GeometricShape, причем код DrawShape не потребует никаких изменений. Ему даже не понадобится ничего знать от новой специализации. Это могут быть Trapezoid, Square (специализации Rectangle) или же Ellipse. Это не имеет значения до тех пор, пока классы фигур наследуются от GeometricShape.

При наследовании класса в методе производного класса часто возникает необходимость вызова метода либо доступа к полю, свойству или индексатору базового класса. Для этой цели предусмотрено ключевое слово base. Это ключевое слово можно применять подобно любой другой переменной экземпляра, но его можно использовать только внутри блока конструктора экземпляра, метода экземпляра или средства доступа к свой-

ству. Применение его в статических методах не допускается. Это совершенно оправдано, потому что base открывает доступ к реализациям экземпляра базового класса, подобно тому, как this разрешает доступ к экземпляру — владельцу текущего метода. Рассмотрим следующий блок кода:

```
public class A
{
    private int x;

    public A(int var)
    {
        this.x = var;
    }

    public virtual void DoSomething()
    {
        Console.WriteLine("A.DoSomething");
    }
}

public class B : A
{
    public B()
        : base(123)
    {
    }

    public override void DoSomething()
    {
        Console.WriteLine("B.DoSomething");
        base.DoSomething();
    }
}

...

B b = new B();
b.DoSomething();
```

В этом примере продемонстрированы два применения ключевого слова base, и одно из них — в конструкторе класса B. Напомним, что класс не наследует конструкторы экземпляра. Однако при инициализации объекта иногда требуется явно вызвать один из конструкторов базового класса во время инициализации производного класса. Это объясняет нотацию, применённую в конструкторе экземпляра класса B. Инициализация базового класса происходит после объявления списка параметров конструктора производного класса, но перед блоком кода конструктора производного класса.

Второе применение ключевого слова base содержится в реализации B.DoSomething. Было принято решение, что при реализации метода DoSomething в классе B необходимо позаимствовать реализацию DoSomething из класса A. Реализацию A.DoSomething можно вызвать непосредственно из реализации B.DoSomething, снабдив вызов префиксом — ключевым словом base.

Когда ключевое слово base встречается внутри члена экземпляра при вызове виртуального метода, оно трактуется специальным образом. Обычно вызов виртуального метода на экземпляре означает вызов его наиболее «унаследованной» версии — в данном случае B.DoSomeThing. Однако когда он вызывается через ключевое слово base, то вызывается наиболее «унаследованная» реализация метода базового класса. Поскольку A — базовый класс, а A.DoSomething — наиболее унаследованная версия метода DoSomething класса A, значит, base.DoSomething вызывает A.DoSomething. Подобным образом можно реализовать переопределённый метод, заимствуя реализацию базового класса.

Виртуальный метод объявляется с использованием в месте его объявления модификатора virtual или abstract. Оба модификатора вводят метод в пространство объявления как таковой, что может быть переопределён в производных классах. Отличие между этими двумя модификаторами в том, что абстрактные методы

обязательно должны быть переопределены, в то время как виртуальные методы — нет. Классы, которые содержат абстрактные методы, сами должны быть помечены как абстрактные. Виртуальные методы, в отличие от абстрактных, обязаны иметь ассоциированную с ними реализацию. Виртуальные методы наряду с их интерфейсами — единственные средства реализации полиморфизма в C#.

Чтобы переопределить метод в производном классе, этот метод должен быть снабжен модификатором `override`. Если этого не сделать, то компилятор предупредит о необходимости указания в объявлении производного метода либо модификатора `new`, либо модификатора `override`. По умолчанию компилятор подразумевает использование модификатора `new`, что, вероятно, даст эффект, противоположный тому, что можно было ожидать. В целом то, что C# требует пометки переопределяющего метода, следует считать положительным моментом, так как это улучшает читабельность кода.

Взгляните на следующий код:

```
public class A
{
    public virtual void SomeMethod()
    {
        Console.WriteLine("A.SomeMethod");
    }
}

public class B : A
{
    public void SomeMethod()
    {
        Console.WriteLine("B.SomeMethod");
    }
}

...

B b = new B();
A a = b;
a.SomeMethod();
```

Приведённый код компилируется, но при этом выдается следующее предупреждение о том, что метод класса `B` скрывает одноимённый метод класса `A`, и предложит программисту уточнить своё намерение, добавив ключевое слово `override` или `new`.

При выполнении кода вызывается метод `A.SomeMethod`. Ключевое слово `new` разбивает виртуальную цепочку в данной точке иерархии. Когда виртуальный метод вызывается через ссылку на объект, то конкретный вызываемый метод определяется по таблице методов во время выполнения. Если метод виртуальный, то исполняющая система движется по иерархии в поисках наиболее удалённой производной версии метода, и затем вызывает её. Однако если она во время поиска встречает метод, помеченный модификатором `new`, то возвращается к методу из предыдущего класса в иерархии и использует его. Таким образом, вызывается именно `A.SomeMethod`. Если бы метод `B.SomeMethod` был помечен ключевым словом `override`, то был бы вызван именно он. Поскольку в C# модификатор `new` применяется по умолчанию, когда не указан никакой другой, компилятор выдает предупреждение, чтобы привлечь внимание. И, наконец, модификатор `new` ортогонален по смыслу модификатору `virtual` — в том плане, что метод, помеченный как `new`, также может быть или не быть виртуальным. В предыдущем примере для метода `B.SomeMethod` не был указан модификатор `virtual`, так что не может быть такого, чтобы класс `C`, производный от `B`, переопределил `B.SomeMethod`, поскольку он не является виртуальным. Таким образом, ключевое слово `new` не только разрушает виртуальную цепочку, но также переопределяет то, получат ли данный класс и классы-наследники `B` виртуальный метод `SomeMethod`.

Другая сложность, о которой следует упомянуть в отношении переопределения методов — как и когда вызывать версию метода базового класса. В C# вызвать версию базового класса можно с использованием идентификатора `base`:

```
public class A
{
    public virtual void SomeMethod()
    {
        Console.WriteLine("A.SomeMethod");
    }
}

public class B : A
{
    public override void SomeMethod()
    {
        Console.WriteLine("B.SomeMethod");
        base.SomeMethod();
    }
}

...

B b = new B();
A a = b;
a.SomeMethod();
```

Как и можно было ожидать, вывод приведенного кода напечатает `A.SomeMethod` в строке, следующей после вывода `B.SomeMethod`. Является ли такой порядок событий правильным? Не должно ли быть все наоборот? Не должен ли метод `B.SomeMethod` вызвать версию базового класса перед тем, как выполнить свою работу? Дело в том, что для ответа на этот вопрос недостаточно информации. Здесь имеется проблема с наследованием и переопределением виртуальных методов. Как узнать, когда следует вызывать метод базового класса, и нужно ли это делать? Ответ заключается в том, что метод должен быть соответствующим образом документирован, чтобы можно было принять правильное решение. Таким образом, наследование с виртуальными методами повышает рабочую нагрузку за счёт обязательного документирования, поскольку потребителей класса потребуется снабдить информацией, выходящей за рамки простого общедоступного интерфейса.

По причинам, установленным ранее, классы по умолчанию должны быть герметизированы (`sealed`), а наследование может быть разрешено только в хорошо продуманных случаях. Тот факт, что наследование сопровождается виртуальными методами, настолько неожиданно сложен, что лучше явно отключать эту возможность, чем открывать её для злоупотребления. Таким образом, при проектировании классов следует отдавать предпочтение созданию герметизированных, ненаследуемых классов и тщательно документировать общедоступный интерфейс. Потребители, которым понадобится расширить функциональность, смогут это сделать, но не через наследование, а через включение. *Расширение включением* (*containement*) в сопровождении изощрённых определений интерфейсов намного мощнее, чем наследование классов.

В редких случаях требуется выполнить наследование от класса с виртуальными методами и завершить виртуальную цепочку своим переопределением. Другими словами, необходимо запретить производным классам дальнейшее переопределение виртуального метода. Чтобы сделать это, метод помечается как герметизированный с помощью модификатора `sealed`. Такая пометка означает, что ни один производный класс не сможет переопределить данный метод. Однако, как говорилось в предыдущем разделе, можно предоставить метод с той же сигнатурой, если он помечен модификатором `new`. Фактически можно было бы пометить новый метод как `virtual`, тем самым начав новую виртуальную цепочку в иерархии. Это не то же самое, что герметизация всего класса, которая вообще запрещает дальнейшее наследование от этого класса. Если производный класс помечен как `sealed`, то снабжение его методов модификатором `sealed` является излишним.

Перегрузка методов

Перегрузка в C# — приём времени компиляции, при котором в точке вызова компилятор выбирает метод из набора одноимённых методов с разной сигнатурой. Для выбора наиболее подходящего метода компилятор использует список аргументов. Типы аргументов, а также модификаторы параметров играют роль в перегрузке методов, поскольку все они составляют часть сигнатуры метода. Тип возврата метода не является частью сигнатуры. Поэтому внутри класса нельзя иметь перегруженные методы, отличающиеся только типом возврата. И,

наконец, если компилятор доходит до точки, где обнаруживается неоднозначность в выборе версии перегруженного метода, компиляция останавливается с выдачей сообщения об ошибке.

Перегрузка не может послужить причиной возникновения исключений времени выполнения, поскольку весь алгоритм выбора применяется во время компиляции. Когда компилятор не может найти точно подходящий метод на основе переданных параметров, он начинает подбор наиболее подходящего соответствия на основе возможности неявного преобразования экземпляров в списке параметров. Поэтому если метод с одним параметром принимает объект типа A, а ему передается объект типа B, унаследованный от A, то в отсутствии версии этого метода, которая бы принимала тип B, компилятор неявно преобразует переданный экземпляр к ссылке типа A, чтобы удовлетворить такой вызов. В зависимости от ситуации и величины набора перегруженных методов процесс выбора может оказаться довольно сложным. Обнаружилось, что лучше минимизировать количество неоднозначных перегрузок, при которых для удовлетворительного решения компилятору нужно выполнять неявные преобразования. Слишком много неявных преобразований могут затруднить понимание кода, вызывая необходимость применения отладчика, чтобы разобраться, что же именно происходит. Это усложняет задачу персонала сопровождения, которому придется следовать за вами и разбираться во всех действиях. Это не значит, что неявные преобразования — плохая вещь при разрешении перегрузки, но лучше использовать их благоразумно и сдержанно, чтобы минимизировать неприятные сюрпризы в будущем.

Пусть у нас есть класс, который содержит информацию об игроке спортивной команды:

```
public class TeamMember
{
    public string FullName { get; private set; }    // имя игрока
    public string Title { get; private set; }       // надпись на футболке
    public string Team { get; private set; }        // название команды
    public bool IsFullTime { get; private set; }    // является ли игрок постоянным
    public TeamMember Manager { get; private set; } // менеджер игрока

    public TeamMember()  // конструктор без параметров
    {
        FullName = null;
        Title = null;
        Team = null;
        IsFullTime = false;
        Manager = null;
    }

    public TeamMember(string fullName, string title, string team, bool isFullTime,
        TeamMember manager) // конструктор с параметрами
    {
        FullName = fullName;
        Title = title;
        Team = team;
        IsFullTime = isFullTime;
        Manager = manager;
    }

    public void AddTeamMember(TeamMember teamMember)
    {
        // добавление игрока в команду
    }

    public void AddTeamMember(string fullName, string title, string team, bool isFullTime,
        TeamMember manager)
    {
        AddTeamMember(new TeamMember(fullName, title, team, isFullTime, manager));
    }
}
```

Класс TeamMember содержит две версии конструктора и две версии метода AddTeamMember, который добавляет игрока в команду. Перегруженные версии методов различаются набором параметров, но выполняют

похожие действия; в данном примере одна из перегруженных версий метода AddTeamMember вызывает другую версию этого же метода. Программист может использовать любую версию перегруженного метода в зависимости от того, какой набор параметров более предпочтителен. В момент вызова метода с несколькими перегрузками компилятор по набору параметров и их типам определяет версию метода с подходящей сигнатурой и передаёт ему управление.

Вообще, перегрузки методов не обязательно должны выполнять одинаковые операции, однако рекомендуется строить класс таким образом, чтобы при замене вызова метода его перегруженной версией (с необходимыми преобразованиями параметров) программа работала точно так же. В любом случае, решение о функциональности перегрузок методов принимается при разработке класса и должно быть надлежащим образом задокументировано.

Параметры методов

Параметры методов по умолчанию объявляют идентификатор переменной, который действителен на протяжении и в контексте самого метода. Здесь нет константных параметров, и параметрам методов можно присваивать значения. Если только параметры не объявлены определённым образом — как `ref` или `out` — такое присваивание остаётся локальным в пределах метода.

В C# аргументы передаются по значению. Однако для ссылок копируемое значение представляет собой саму ссылку, а не объект, на который она ссылается. Изменения в состоянии ссылаемого объекта внутри метода становятся видимыми коду, вызвавшему этот метод.

В действительности все параметры, передаваемые в методы, являются *аргументами-значениями*, если предполагается, что они простые, *недекорированные параметры* методов. Под недекорированными понимается отсутствие специальных ключевых слов, таких как `out`, `ref` и `params`, которые могут быть присоединены к ним. Однако они могут иметь присоединённые атрибуты, как почти все что угодно в системе типов CLR. Подобно всем параметрам, идентификатор находится в контексте блока метода, который следует за списком параметров (т.е. внутри фигурных скобок), а метод получает копию переданной переменной в момент его вызова. Однако, если переданная переменная является структурой, или типом значения, то метод получает копию этого значения. Любые изменения, проведенные локально в этой переменной, вызывающему коду не видны. Если же передаваемая переменная является ссылкой на объект в куче, как любая переменная для экземпляра класса, то метод получает копию ссылки. То есть любые изменения, проведенные в объекте через такую ссылку, становятся видимыми коду, вызвавшему метод.

Передача *параметров по ссылке* отмечается с помощью модификатора `ref` перед типом параметра в списке параметров метода. Когда переменная передаётся по ссылке, новая копия этой переменной не создаётся, и эту переменную из вызывающего метода напрямую затрагивают все действия внутри метода. Как обычно бывает в CLR, это означает две слегка отличающиеся вещи, в зависимости от того, является переменная экземпляром типа значения (структуры) или же объекта (класса).

Когда экземпляр значения передаётся по ссылке, копия значения из вызывающего кода не создаётся. Когда экземпляр объекта (ссылки) передаётся по ссылке, никакой копии переменной не создаётся. Фактически переменная ведет себя подобно указателю C++ на ссылочную переменную, что в C++ выглядит как указатель на указатель. Вдобавок верификатор проверяет, чтобы переменная, на которую ссылается параметр `ref`, имела определённое присвоенное значение перед вызовом метода. Рассмотрим некоторые примеры применения полной нотации `ref`-параметров:

```
public struct MyStruct
{
    public int val;
}

...

static void PassByValue(MyStruct myValue)
{
    myValue.val = 50;
}
```

```
static void PassByRef(ref MyStruct myValue)
{
    myValue.val = 42;
}

...

MyStruct myValue = new MyStruct();
myValue.val = 10;

PassByValue(myValue);
Console.WriteLine("Результат PassByValue: myValue.val = {0}", myValue.val);

PassByRef(ref myValue);
Console.WriteLine("Результат PassByRef: myValue.val = {0}", myValue.val);
```

В примере содержатся два метода: `PassByValue` и `PassByRef`. Оба метода модифицируют поле экземпляра типа значения, переданного им. Однако, как доказывает приведенный ниже вывод, метод `PassByValue` модифицирует локальную копию, в то время как метод `PassByRef` модифицирует экземпляр из вызывающего кода:

```
Результат PassByValue: myValue.val = 10
Результат PassByRef: myValue.val = 42
```

Кроме того, обратите внимание, что в точке вызова метода `PassByRef` требуется ключевое слово `ref`. Это необходимо, потому что метод может быть перегружен на основе присутствия или отсутствия ключевого слова `ref`. Другими словами, другой метод `PassByRef` мог бы также принимать `MyStruct` по значению, а не по ссылке. Плюс к этому обязательное применение ключевого слова `ref` в точке вызова облегчает чтение кода.

Теперь рассмотрим пример, в котором вместо типа значения используется объект:

```
static void PassByRef(ref object myObject)
{
    // Присвоить новый экземпляр этой переменной
    myObject = new object();
}

...

object myObject = new object();
Console.WriteLine("myObject.GetHashCode() == {0}", myObject.GetHashCode());

PassByRef(ref myObject);
Console.WriteLine("myObject.GetHashCode() == {0}", myObject.GetHashCode());
```

В этом случае переданная по ссылке переменная является объектом. Но как было сказано, вместо того, чтобы передать в метод копию ссылки, тем самым создавая новую ссылку на тот же объект, здесь передаётся сама исходная ссылка. В предыдущем методе `PassByRef` переданная ссылка переустанавливается на новый экземпляр объекта. Исходный объект остается без ссылок, и потому готов к тому, чтобы его обработал сборщик мусора. Для получения доказательств, что новая переменная `myObject` ссылается на два разных экземпляра в точке вызова и в точке, следующей за вызовом, результаты вызовов `myObject.GetHashCode` отправляются на консоль. Полученный в результате вывод будет таким:

```
myObject.GetHashCode() == 46104728
myObject.GetHashCode() == 12289376
```

Параметры `out` (*выходные*) почти идентичны параметрам `ref`, но с двумя существенными отличиями. Во-первых, вместо ключевого слова `ref` используется ключевое слово `out`, и оно также должно применяться в точке вызова, как это было в случае `ref`. Во-вторых, переменная, на которую ссылается переменная `out`, не

обязана иметь присвоенное значение перед вызовом метода, как в случае ref-параметра. Причина в том, что методу не позволено каким-либо полезным способом использовать переменную до тех пор, пока ей не будет присвоено значение. Например, следующий код совершенно корректен:

```
static void PassAsOutParam(out object obj)
{
    obj = new Object();
}

...

object obj;
PassAsOutParam(out obj);
```

Обратите внимание, что переменной obj в методе Main перед вызовом PassAsOutParam значение напрямую не присваивается. И это правильно, потому что она помечена как out-параметр. Метод PassAsOutParam не обращается к этой переменной, пока не присвоит ей значение. Если вы попробуете заменить два вхождения out на ref в приведенном выше коде, то получите ошибку компиляции.

C# позволяет легко передавать переменный список параметров. Для этого просто объявите последний параметр в списке параметров с типом массива, предварив его ключевым словом params. Теперь, если метод вызывается с переменным количеством параметров, то эти параметры передаются ему в форме массива, по которому легко выполнить итерацию, и тип этого массива может базироваться на любом допустимом типе. Ниже приведен краткий пример:

```
static void VarArgs(int vail, params int[] vals)
{
    Console.WriteLine("vail: {0}", vail);
    foreach (int i in vals)
    {
        Console.WriteLine("vals[]: {0}", i);
    }
    Console.WriteLine();
}

...

VarArgs(42);
VarArgs(42, 43, 44);
VarArgs(44, 56, 23, 234, 45, 123);
```

В каждом случае VarArgs вызывается успешно, но в каждом случае массив, переданный в vals, отличается. Как видите, передача переменного числа параметров в C# замечательно проста. Можете построить эффективный метод Add для контейнерного типа, применяя массивы параметров, когда для добавления переменного количества элементов необходим только один вызов.

Необязательные параметры

Необязательные аргументы в C# работают подобно своим аналогам в C++. В объявлении метода для параметра можно указать значение по умолчанию. Любой параметр, не имеющий значение по умолчанию, рассматривается как обязательный. Из этого следует, что все параметры со значениями по умолчанию должны размещаться в конце списка параметров метода. Рассмотрим следующий пример:

```
class TeamMember
{
    public string FullName { get; private set; }
    public string Title { get; private set; }
    public string Team { get; private set; }
    public bool IsFullTime { get; private set; }
    public TeamMember Manager { get; private set; }
```

```
    public TeamMember(string fullName, string title = "Unknown", string team = "Unknown",
        bool isFullTime = false, TeamMember manager = null)
    {
        FullName = fullName;
        Title = title;
        Team = team;
        IsFullTime = isFullTime;
        Manager = manager;
    }
}

...

TeamMember tm = new TeamMember("Milton Waddams");
```

В этом примере в объявлении конструктора используются необязательные параметры, и при инициализации экземпляра `TeamMember` участвуют ассоциированные необязательные аргументы. Обратите внимание, что все значения параметров по умолчанию являются константами. В качестве значений параметров по умолчанию ничего кроме констант, существующих во время компиляции, указывать не разрешено. В случае нарушения этого правила компилятор выдаст сообщение.

Именованные аргументы — новое средство, которое появилось в C# 4.0. В действительности оно дополняет необязательные аргументы. Предположим, что необходимо создать экземпляр `TeamMember`, приняв все аргументы конструктора по умолчанию за исключением `isFullTime`. Для этого можно использовать именованный аргумент:

```
TeamMember tm = new TeamMember("Peter Gibbons", isFullTime: true);
```

Обратите внимание на способ передачи аргумента `isFullTime` конструктору. Применять именованные аргументы легко. Необходимо просто указать имя аргумента, за которым следует двоеточие и значение, которое должно быть передано для этого аргумента. Остальные аргументы при вызове конструктора сохраняют свои значения по умолчанию. В приведённом выше вызове конструктора именованные аргументы фактически можно было бы использовать и для передачи значений обязательных аргументов. При этом можно было бы изменить порядок аргументов в списке вызова:

```
TeamMember tm = new TeamMember(isFullTime: true, fullName: "Peter Gibbons");
```

Именованные аргументы оказывают незначительное влияние на перегрузку методов. Позиционный список аргументов конструируется за счёт комбинирования заданных позиционных аргументов вместе с именованными аргументами и всеми применимыми аргументами по умолчанию. Как только позиционный список аргументов сконструирован, разрешение перегрузок выполняется нормально. Единственное место, где именованные аргументы представляют сложность — это виртуальные переопределения. Рассмотрим следующий надуманный пример:

```
class A
{
    public virtual void DoSomething(int x, int y)
    {
        Console.WriteLine("{0}, {1}", x, y);
    }
}

class B : A
{
    public override void DoSomething(int y, int x)
    {
        base.DoSomething(y, x);
    }
}
```

```
...

B b = new B();
b.DoSomething(x: 1, y: 2);
A a = b;
a.DoSomething(x: 1, y: 2);
```

Это совершенно экстремальный пример, и поступать подобным образом в реальной разработке ни в коем случае нельзя! Запуск на выполнение этого кода даёт следующий вывод:

```
2, 1
1, 2
```

При переопределении виртуального метода в типе-наследнике должно выполняться единственное требование — соответствие типов параметров и их позиций объявлению базового метода. Как было показано выше, действительные имена аргументов могут измениться. Проблема в том, что компилятор должен иметь надёжную методику отображения именованного аргумента на позиционный аргумент виртуального метода. Он делает это, используя объявление метода статического типа переменной-ссылки.

В приведённом выше примере две ссылки — a и b — ссылаются на один и тот же экземпляр `B`. Однако при вызове `DoSomething` с именованными аргументами имеет значение объявление `DoSomething`, ассоциированное со статическим типом соответствующей ссылки. В первом вызове `DoSomething` через переменную `b`, поскольку её статическим типом является `B`, именованные аргументы разрешаются на основе определения `B.DoSomething`. Но в следующем вызове `DoSomething` через переменную a, из-за того, что её статическим типом является `A`, именованные аргументы разрешаются через обращение к определению `A.DoSomething`. Как видите, этого следует избегать любой ценой, поскольку оно определённо вносит некоторую излишнюю путаницу в код.

И, наконец, следует учесть еще один момент: когда в списке аргументов присутствуют выражения, они вычисляются в том порядке, в котором появляются в списке, даже если этот список отличается от порядка позиционных параметров вызываемого метода. Взгляните на следующий пример:

```
class A
{
    public virtual void DoSomething(int x, int y)
    {
        Console.WriteLine("{0}, {1}", x, y);
    }
}

class B : A
{
    public override void DoSomething(int y, int x)
    {
        base.DoSomething(y, x);
    }
}

...

static int GenerateValue1()
{
    Console.WriteLine("Вызов GenerateValue1");
    return 1;
}

static int GenerateValue2()
{
    Console.WriteLine("Вызов GenerateValue2");
    return 2;
}
```

```
...

A a = new A();
a.DoSomething(GenerateValue1(), GenerateValue2()); // Теперь используем именованные аргументы
a.DoSomething(y: GenerateValue2(), x: GenerateValue1());
```

В результате запуска на выполнение приведённого выше кода получается следующий вывод:

```
Вызов GenerateValue1
Вызов GenerateValue2
1, 2
Вызов GenerateValue2
Вызов GenerateValue1
1, 2
```

Обратите внимание, что порядок вызова методов `GenerateValue1` и `GenerateValue2` зависит от порядка, в котором они появляются в списке аргументов, независимо от того, с какими позиционными параметрами они ассоциированы. После того, как выражения вычислены — в данном случае, после вызовов `GenerateValue1` и `GenerateValue2` — аргументы помещаются в их соответствующие позиции, чтобы найти лучший метод.

До появления именованных аргументов можно было писать код, в котором имел значение порядок вычисления выражений в списках параметров. Этот подход нельзя назвать удачным, как с именованными аргументами, так и без них. Предположим, что в предыдущем примере методы закодированы с побочным эффектом — исходя из того, что `GenerateValue1` будет всегда вызываться перед `GenerateValue2`. Далее предположим, что метод `A.DoSomething` был вызван с использованием позиционных аргументов, когда текущей была версия C# 3.0. Позднее, после появления именованных аргументов, инженер службы поддержки решил изменить код и передать аргументы в противоположном порядке, применив именованные аргументы. Сразу же возникнет серьезная проблема! Поэтому следует избегать ситуации, когда приходится полагаться на определённый порядок вычисления выражений в списке аргументов.

Перегрузка операций

В языке C# есть возможность перегружать не только методы, но и операции. Перегруженные операции определяются 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 включающего метод класса. Например, не имеет смысла перегружать операцию + в классе `Complex`, если он будет складывать вместе два значения `double`. К тому же это даже невозможно. Типичная операция + для класса `Complex` может выглядеть так:

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

Несмотря на то что этот метод складывает два экземпляра `Complex`, чтобы создать третий экземпляр `Complex`, ничто не запрещает сделать один из параметров типа `double`, чтобы таким образом прибавлять `double` к экземпляру `Complex`. В общем случае синтаксис перегрузки операций следует приведённому шаблону, где + заменяется нужной операцией. Некоторые операции принимают только один параметр.

Существуют всего три группы перегружаемых операций:

- *унарные операции* принимают только один параметр. К хорошо знакомым 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;
- *операции преобразования* определяют пользовательские преобразования типов. Они должны 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, будучи таковыми, наследуются производными классами, методы операций должны иметь в своём объявлении как минимум

один параметр, совпадающий с включающим типом, что делает невозможным точное соответствие между методом операции производного типа и сигнатурой метода операции базового класса.

Как известно, методы операций являются статическими. Поэтому настоятельно рекомендуется не изменять передаваемые им параметры. Вместо этого необходимо создавать новый экземпляр возвращаемого типа и возвращать его в качестве результата операции.

Неизменяемые структуры и классы вроде `System.String` лучше всего подходят для реализации пользовательских операций. Такое поведение естественно для булевских операций, которые обычно возвращают тип, отличающийся от переданных типов.

Предположим, что создаётся структура для представления простых комплексных чисел, скажем, `Complex`, и требуется складывать вместе экземпляры `Complex`. Было бы также удобно иметь возможность прибавлять простые значения `double` к экземпляру `Complex`. Добавление такой функциональности — не проблема, поскольку можно перегрузить метод операции + так, чтобы один параметр был `Complex`, а другой — `double`. Это объявление может выглядеть примерно так:

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

При наличии такой операции, объявленной и определенной в структуре `Complex`, появляется возможность писать код вроде следующего:

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

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

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

Поддержка разного порядка следования для операндов различных типов достигается с помощью отдельных перегрузок для операции. Если перегружается бинарная операция, использующая разные типы параметров, то можно создавать *зеркальную* перегрузку — т.е. другой метод операции, в котором параметры просто меняются местами.

Перегрузка операции сложения

Рассмотрим краткий пример структуры `Complex`, которая демонстрирует перегрузку операций:

```
public struct Complex
{
    private double real;
    private double imaginary;

    public Complex(double real, double imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public static Complex Add(Complex lhs, Complex rhs)
    {
        return new Complex(lhs.real + rhs.real, lhs.imaginary + rhs.imaginary);
    }

    public static Complex Add(Complex lhs, double rhs)
    {
        return new Complex(rhs + lhs.real, lhs.imaginary);
    }

    public static string ToString()
```

```
    {
        return String.Format("({0}, {1})", real, imaginary);
    }

    public static Complex operator +(Complex lhs, Complex rhs)
    {
        return Add(lhs, rhs);
    }

    public static Complex operator +(double lhs, Complex rhs)
    {
        return Add(rhs, lhs);
    }

    public static Complex operator +(Complex lhs, double rhs)
    {
        return Add(lhs, rhs);
    }
}

...

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 и рекомендуется, перегруженные методы операции вызывают методы, выполняющие саму операцию. Фактически это существенно упрощает поддержку обоих последовательностей операндов `operator +` для типа Complex.

Перегрузка операций сравнения

Бинарные операции сравнения == и !=, < и >, а также >= и <= должны быть реализованы парами. Это вполне естественно, поскольку вряд ли возникнет ситуация, когда пользователям разрешено применять, скажем, операцию ==, но не разрешено !=. Перегрузка операций сравнения требует также переопределения методов Equals и GetHashCode:

```
public struct Complex
{
    private double real;
    private double imaginary;
    // модуль комплексного числа, используется для определения хеш-кода числа
    public double Magnitude
    {
        get { return Math.Sqrt(Math.Pow(this.real, 2) + Math.Pow(this.imaginary, 2)); }
    }

    public Complex(double real, double imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public override bool Equals(object other)
    {
        bool result = false;
        if (other is Complex)
```

```
            result = Equals((Complex)other);
        return result;
    }

    public bool Equals(Complex that)
    {
        return (this.real == that.real && this.imaginary == that.imaginary);
    }
    // метод GetHashCode возвращает хэш-код, который является целым числом
    // значение хэш-кода должно быть одинаковым для одинаковых комплексных чисел
    public override int GetHashCode()
    {
        return (int)this.Magnitude;
    }
    // метод GetHashCode возвращает 0, если текущее число и that одинаковы,
    // 1, если текущее число больше чем that, и -1, если that больше чем текущее число
    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;
    }

    public override string ToString()
    {
        return String.Format("({0}, {1})", real, imaginary);
    }
    // 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;
    }
}

...

Complex cpx1 = new Complex(1.0, 3.0);
```

```
Complex cpx2 = new Complex(1.0, 2.0);
Console.WriteLine("cpxl = {0}, cpx1.Magnitude = {1}", cpx1, cpx1.Magnitude);
Console.WriteLine("cpx2 = {0}, cpx2.Magnitude = {l}\n", cpx2, cpx2.Magnitude);
Console.WriteLine("cpxl == cpx2 ? {0}", cpx1 == cpx2);
Console.WriteLine("cpxl != cpx2 ? {0}", cpx1 != cpx2);
Console.WriteLine("cpxl < cpx2 ? {0}", cpx1 < cpx2);
Console.WriteLine("cpxl > cpx2 ? {0}", cpx1 > cpx2);
Console.WriteLine("cpxl <= cpx2 ? {0}", cpx1 <= cpx2);
Console.WriteLine("cpxl >= cpx2 ? {0}", cpx1 >= cpx2);
```

Обратите внимание, что методы операций просто вызывают методы, реализующие `Equals` и `CompareTo`.

Операции преобразования

Операции преобразования, как следует из их названия, являются операциями, преобразующими объекты одного типа в объекты другого типа. Операции преобразования могут допускать как неявные, так и явные преобразования. Неявные преобразования выполняются при простом присваивании, в то время как явные требуют синтаксиса приведения, 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 и всегда гарантированно завершались успешно, без потери информации. Если соблюсти это требование не удается, то 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` (`short.MaxValue`). Такое преобразование должно быть явным и требовать от пользователя применения синтаксиса приведения. Теперь предположим, что выполняется противоположное преобразование, т.е. `short` в `long`. Такое преобразование всегда проходит успешно, поэтому оно может быть неявным.

Давайте посмотрим, какие операции преобразования должны быть предусмотрены для `Complex`. Довольно несложно 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` всё равно не имеет смысла и показано здесь только для примера, в качестве результата этого приведения можно возвращать значение по модулю, а не просто вещественную часть комплексного числа. Рассмотрим пример реализации операций приведения:

```
public struct Complex
{
    private double real;
    private double imaginary;

    public double Magnitude
    {
        get { return Math.Sqrt(Math.Pow(this.real, 2) + Math.Pow(this.imaginary, 2)); }
    }

    public Complex(double real, double imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public override string ToString()
    {
        return String.Format("({0}, {1})", real, imaginary);
    }

    public static implicit operator Complex(double d)
    {
        return new Complex(d, 0);
```

```
    }

    public static explicit operator double(Complex c)
    {
        return c.Magnitude;
    }
    // остальные методы опущены для ясности
}

...

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);
```

В коде метода `Main` используются операции преобразования. При реализации операций преобразования будьте осторожны, чтобы не создать для пользователей никаких сюрпризов или путаницы с неявными преобразованиями. Внести путаницу с явными операциями, когда пользователи типа вынуждены применять синтаксис приведения для того, чтобы заставить их работать, довольно-таки сложно. Невнимательное или неправильное использование неявного преобразования может стать причиной большой путаницы. Если написать множество операций неявного преобразования, не имеющих семантического смысла, 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 сочтёт это необходимым.

Рассмотрим случай, когда `Complex` предоставляет другую операцию явного преобразования в экземпляр `Fraction`, а 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 тип `Complex` и возвращают другой тип. Однако правила перегрузки чётко указывают, что тип возврата в сигнатуре метода не учитывается. 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 операции преобразования как часть сигнатуры. Между прочим, ключевые слова `implicit` и `explicit` не входят в сигнатуру методов операций преобразования, поэтому иметь одновременно и явную, и неявную версии одной и той же операции преобразования невозможно. Естественно, по крайней мере, один из типов в сигнатуре операции преобразования должен быть включающим типом.

Булевские операции

Некоторые типы могут принимать участие в булевских выражениях проверки, таких как встречающиеся внутри скобок блока `if` или внутри тернарной операции `?:`. Чтобы это работало, имеются две альтернативы. Первая заключается в том, что можно реализовать две операции преобразования, известные как `operator true` и `operator false`. Они должны быть реализованы в паре, чтобы позволить комплексному числу (типу `Complex`) участвовать в булевских выражениях проверки. Рассмотрим следующую модификацию типа `Complex`, которая позволяет использовать его в выражениях; здесь значение (0, 0) означает `false`, а все остальное — `true`:

```
public struct Complex
{
```

```
    private double real;
    private double imaginary;

    public double Magnitude
    {
        get { return Math.Sqrt(Math.Pow(this.real, 2) + Math.Pow(this.imaginary, 2)); }
    }

    public Complex(double real, double imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public override string ToString()
    {
        return String.Format("({0}, {1})", real, imaginary);
    }

    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);
    }
    // остальные методы опущены для ясности
}

...

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");
```

Здесь реализованы две операции для применения к типу `Complex` проверок на предмет равенства `true` и `false`. Обратите внимание на несколько причудливый синтаксис. Он выглядит почти так же, как в обычных операциях, за исключением того, что типом возврата является `bool`. Зачем это нужно — не совсем понятно, поскольку указать какой-то другой тип возврата для этих операций всё равно не удастся. Если всё-таки попробовать это сделать, то компилятор немедленно сообщит, что единственным допустимым типом возврата для `operator true` и `operator false` является `bool`. Тем не менее, тип возврата должен быть указан. Также обратите внимание, что эти операции нельзя пометить как `explicit` или `implicit`, потому что они не являются операциями преобразования. После определения этих двух операций в типе можно использовать экземпляры `Complex` в булевских выражениях проверки, как показано в примере.

В качестве альтернативы можно реализовать преобразование к типу `bool` и получить тот же результат. Обычно для простоты использования эта операция реализуется неявно. Рассмотрим модифицированную форму предыдущего примера с использованием неявной операции преобразования к `bool` вместо `operator true` и `operator false`:

```
public struct Complex
{
    private double real;
    private double imaginary;
```

```
    public double Magnitude
    {
        get { return Math.Sqrt(Math.Pow(this.real, 2) + Math.Pow(this.imaginary, 2)); }
    }

    public Complex(double real, double imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public override string ToString()
    {
        return String.Format("({0}, {1})", real, imaginary);
    }

    public static implicit operator bool(Complex c)
    {
        return (c.real != 0) || (c.imaginary != 0);
    }

    // остальные методы опущены для ясности
}

...

Complex cpx1 = new Complex(1.0, 3.0);
if (cpx1)
    Console.WriteLine("cpxl равно true");
else
    Console.WriteLine("cpxl равно false");

Complex cpx2 = new Complex(0, 0);
Console.WriteLine("cpx2 равно {0}", cpx2 ? "true" : "false");
```

Конечный результат выглядит так же, как в предыдущем примере. Теперь может возникнуть вопрос, а зачем вообще реализовывать `operator true` и `operator false` вместо простой неявной операции преобразования к `bool`? Ответ зависит от того, допустимо ли для заданного типа преобразование в тип `bool`.

Существует следующее эмпирическое правило: предусматривайте только те операции, которые действительно необходимы для выполнения работы, и не более. Если всё, что нужно от 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 действительно не нуждаетесь в ней. Если окажется, что она всё-таки нужна и должна быть реализована, тогда не понадобится реализовывать `operator true` и `operator false`, потому что они будут избыточны. Если представлены все три операции, то компилятор будет использовать операцию неявного преобразования в `bool` вместо `operator true` и `operator false`, поскольку вызов первой не является более эффективным, чем остальных, предполагая, что закодированы они одинаково.