Классы — это, по сути, шаблоны, по которым можно создавать объекты. Каждый объект содержит данные и методы, манипулирующие этими данными. Класс определяет, какие данные и функциональность может иметь каждый конкретный объект (называемый экземпляром) этого класса. Например, если имеется класс, представляющий заказчика, он может определять такие поля, как CustomerlD, FirstName, LastName и Address, которые нужны ему для хранения информации о конкретном заказчике. Он также может определять функциональность, которая работает с данными, хранящимися в этих полях. Вы создаете экземпляр этого класса для представления конкретного заказчика, устанавливаете значения полей экземпляра и используете его функциональность.

**class PhoneCustomer**

**{**

**public const string DayOfSendingBill = "Monday";**

**public int CustomerlD;**

**public string FirstName;**

**public string LastName;**

**}**

Структуры отличаются от классов тем, как они сохраняются в памяти и как к ним осуществляется доступ (классы — это ссылочные типы, размещаемые в куче, структуры — типы значений, размещаемые в стеке), а также некоторыми свойствами (например, структуры не поддерживают наследование). Из соображений производительности вы будете использовать структуры для небольших типов данных. Однако в отношении синтаксиса структуры очень похожи на классы. Главное отличие состоит в том, что при их объявлении используется ключевое слово struct вместо class. Например, если вы хотите, чтобы все экземпляры PhoneCustomer размещались в стеке, а не в управляемой куче, то можете написать следующий код:

**struct PhoneCustomerStruct**

**{**

**public const string DayOfSendingBill = "Monday";**

**public int CustomerlD;**

**public string FirstName;**

**public string LastName;**

**}**

При создании как классов, так и структур используется ключевое слово new для объявления экземпляра. В результате объект создается и инициализируется. В следующем примере по умолчанию все поля обнуляются:

**PhoneCustomer myCustomer = new PhoneCustomer(); // работает с классом**

**PhoneCustomerStruct myCustomer2 = new PhoneCustomerStruct(); // работает**

**//со структурой**

В большинстве случаев классы используются чаще структур. По этой причине в главе сначала рассматриваются классы, а затем обсуждается разница между классами и структурами и специфические причины выбора структур вместо классов. Если только не указано обратное, вы можете рассчитывать на то, что код, работающий с классом, будет также работать со структурой.

**Классы**

Данные и функции, объявленные внутри класса, известны как члены класса. В официальной терминологии Microsoft делается различие между данными-членами и функциями-членами. В дополнение к членам, классы могут содержать в себе вложенные типы (такие как другие классы). Доступность членов класса может быть описана как **public, protected, internal protected, private** или **internal**.

**Данные-члены**

Данные-члены — это те члены, которые содержат данные класса — поля, константы, события. Данные-члены могут быть статическими (static). Член класса является членом экземпляра, если только он не объявлен явно как static.

Поля (field) — это любые переменные, ассоциированные с классом. Вы уже видели поля в классе PhoneCustomer в предыдущем примере. После создания экземпляра объекта PhoneCustomer к его полям можно обращаться с использованием синтаксиса **Объект.ИмяПоля**, как показано в следующем примере:

**PhoneCustomer Customerl = new PhoneCustomer();**

**Customerl.FirstName = "Simon";**

Константы могут быть ассоциированы с классом тем же способом, что и переменные. Константа объявляется с помощью ключевого слова const. Если она объявлена как public, то в этом случае становится доступной извне класса.

**class PhoneCustomer**

**{**

**public const string DayOfSendingBill = "Monday";**

**public int CustomerlD;**

**public string FirstName;**

**public string LastName;**

**}**

Для того чтобы продемонстрировать классы на конкретных примерах, разработаем постепенно класс, инкапсулирующий информацию о зданиях, в том числе о домах, складских помещениях, учреждениях и т.д. В этом классе будут храниться три элемента информации о зданиях: количество этажей, общая площадь и количество жильцов. Ниже приведен первый вариант. В нем определены три переменные экземпляра: **Floors**, **Area** и **Occupants**. Как видим, в нём вообще отсутствуют методы. Это означает, что в настоящий момент этот класс состоит только из данных.

**using System;**

**using System.Collections.Generic;**

**using System.Linq;**

**using System.Text;**

**namespace BuildingProbe**

**{**

**class Building**

**{**

**public int Floors; // количество этажей**

**public int Area; // общая площадь здания**

**public int Occupants; // количество жильцов**

**}**

**//В этом классе объявляется объект типа Building,**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**Building house = new Building(); // создать объект типа Building**

**int areaPP; // площадь на одного человека**

**// Присвоить значения полям в объекте house,**

**house.Occupants = 4;**

**house.Area = 250;**

**house.Floors = 2;**

**// Вычислить площадь на одного человека.**

**areaPP = house.Area / house.Occupants;**

**Console.WriteLine("Дом имеет:\n " +**

**house.Floors + " этажа\n " +**

**house.Occupants + " жильца\n " +**

**house.Area +**

**" кв. метров общей площади, из них\n " +**

**areaPP + " приходится на одного человека");**

**Console.ReadLine();**

**}**

**}**

**}**

Эта программа состоит из двух классов: **Building** и **Program**. В классе **Program** сначала создается экземпляр **house** класса **Building** с помощью метода **Main ()**, а затем в коде метода **Main ()** осуществляется доступ к переменным экземпляра **house** для присваивания им значений и последующего использования этих значений. Следует особо подчеркнуть, что **Building** и **BuildingDemo** — это два совершенно отдельных класса. Единственная взаимосвязь между ними состоит в том, что в одном из них создается экземпляр другого. Но, несмотря на то, что это раздельные классы, у кода из класса **Program** имеется доступ к членам класса **Building**, поскольку они объявлены как открытые *(public)*. Если бы при их объявлении не был указан спецификатор доступа *public*, то доступ к ним ограничивался бы пределами **Building**, а следовательно, их нельзя было бы использовать в классе **Program**.

События — это члены класса, позволяющие объекту уведомлять вызывающий код о том, что случилось нечто достойное упоминания, например, изменение свойства класса либо некоторое взаимодействие с пользователем. Клиент может иметь код, известный как обработчик событий, реагирующий на них.

**Функции-члены**

Функции-члены — это члены, которые обеспечивают некоторую функциональность для манипулирования данными класса. Они включают методы, свойства, конструкторы, финализаторы, операции и индексаторы.

* Методы (method) — это функции, ассоциированные с определенным классом. Как и данные-члены, по умолчанию они являются членами экземпляра. Они могут быть объявлены статическими с помощью модификатора static.
* Свойства (property) — это наборы функций, которые могут быть доступны клиенту таким же способом, как общедоступные поля класса. В С# предусмотрен специальный синтаксис для реализации чтения и записи свойств для классов, поэтому писать собственные методы с именами, начинающимися на Set и Get, не понадобится. Поскольку не существует какого-то отдельного синтаксиса для свойств, который отличал бы их от нормальных функций, создается иллюзия объектов как реальных сущностей, предоставляемых клиентскому коду.
* Конструкторы (constructor) — это специальные функции, вызываемые автоматически при инициализации объекта. Их имена совпадают с именами классов, которым они принадлежат, и они не имеют типа возврата. Конструкторы полезны для инициализации полей класса.
* Финализаторы (finalizer) похожи на конструкторы, но вызываются, когда среда CLR определяет, что объект больше не нужен. Они имеют то же имя, что и класс, но с предшествующим символом тильды (~). Предсказать точно, когда будет вызван финализатор, невозможно.
* Операции (operator) — это простейшие действия вроде + или -. Когда вы складываете два целых числа, то, строго говоря, применяете операцию + к целым. Однако С# позволяет указать, как существующие операции будут работать с пользовательскими классами (так называемая перегрузка операций).
* Индексаторы (indexer) позволяют индексировать объекты таким же способом, как массив или коллекцию.

**Создание объектов**

В предыдущем примере для объявления объекта типа **Building** использовалась следующая строка кода:

**Building house = new Building();**

Эта строка объявления выполняет три функции. Во-первых, объявляется переменная **house**, относящаяся к типу класса **Building**. Сама эта переменная не является объектом, а лишь переменной, которая может ссылаться на объект. Во-вторых, создается конкретная, физическая, копия объекта. Это делается с помощью оператора **new**. И наконец, переменной **house** присваивается ссылка на данный объект. Таким образом, после выполнения анализируемой строки объявленная переменная **house** ссылается на объект типа Building. Оператор **new** динамически (т.е. во время выполнения) распределяет память для объекта и возвращает ссылку на него, которая затем сохраняется в переменной.

Следовательно, в С# для объектов всех классов должна быть динамически распределена память. Как и следовало ожидать, объявление переменной **house** можно отделить от создания объекта, на который она ссылается, следующим образом:

**Building house; // объявить ссылку на объект**

**house = new Building(); // распределить память для объекта типа Building**

В первой строке объявляется переменная **house** в виде ссылки на объект типа **Building**. Следовательно, **house** — это переменная, которая может ссылаться на объект, хотя сама она не является объектом. А во второй строке создается новый объект типа **Building**, и ссылка на него присваивается переменной house. В итоге переменная **house** оказывается связанной с данным объектом. То обстоятельство, что объекты классов доступны по ссылке, объясняет, почему классы называются ссылочными типами. Главное отличие типов значений от ссылочных типов заключается в том, что именно содержит переменная каждого из этих типов. Так, переменная типа значения содержит конкретное значение. Например, во фрагменте кода

**int x;**

**х = 10;**

переменная '*х* содержит значение 10, поскольку она относится к типу int, который является типом значения. Но в строке

**Building house = new Building();**

переменная **house** содержит не сам объект, а лишь ссылку на него.

**Методы**

Следует отметить, что официальная терминология С# делает различие между функциями и методами. Согласно этой терминологии, понятие "функция-член" включает не только методы, но также другие члены, не являющиеся данными, класса или структуры. Сюда входят индексаторы, операции, конструкторы, деструкторы, а также — возможно, несколько неожиданно — свойства. Они контрастируют с данными-членами: полями, константами и событиями.

**Объявление методов**

В С# определение метода состоит из любых модификаторов (таких как спецификация доступности), типа возвращаемого значения, за которым следует имя метода, затем списка аргументов в круглых скобках и далее — тела метода в фигурных скобках:

**[модификаторы] тип\_возврата ИмяМетода([параметры])**

**{**

**// Тело метода**

**}**

Каждый параметр состоит из имени типа параметра и имени, по которому к нему можно обратиться в теле метода. Вдобавок, если метод возвращает значение, то для указания точки выхода должен использоваться оператор возврата вместе с возвращаемым значением. Например:

**public bool IsSquare(Rectangle rect)**

**{**

**return (rect.Height == rect.Width);**

**}**

В этом коде применяется один из базовых классов .NET, System.Drawing.Rectangle, представляющий прямоугольники. Если метод не возвращает ничего, то в качестве типа возврата указывается void, поскольку вообще опустить тип возврата невозможно. Если же он не принимает аргументов, то все равно после имени метода должны присутствовать пустые круглые скобки. При этом включать в тело метода оператор возврата не обязательно — метод возвращает управление автоматически по достижении закрывающей фигурной скобки. Следует отметить, что метод может содержать любое необходимое количество операторов возврата:

**public bool IsPositive(int value)**

**{**

**if (value < 0)**

**return false;**

**return true;**

**}**

Используя возвращаемое значение, можно усовершенствовать рассматривавшийся ранее пример с классом **Building**. Вместо того чтобы вычислять величину площади на одного человека в методе **Main()** , лучше возвратить ее из этого метода **AreaPerPerson()** класса **Building**. Среди прочих преимуществ такого подхода следует особо отметить возможность использовать возвращаемое значение для выполнения других вычислений. Приведенный ниже пример представляет собой улучшенный вариант рассматривавшейся ранее программы с методом **AreaPerPerson ()**, возвращающим величину площади на одного человека.

**using System;**

**using System.Collections.Generic;**

**using System.Linq;**

**using System.Text;**

**namespace BuildingProbe**

**{**

**class Building**

**{**

**public int Floors; // количество этажей**

**public int Area; // общая площадь здания**

**public int Occupants; // количество жильцов**

**// Возвратить величину площади на одного человека,**

**public int AreaPerPerson()**

**{**

**return Area / Occupants;**

**}**

**}**

**//В этом классе объявляется объект типа Building,**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**Building house = new Building(); // создать объект типа Building**

**int areaPP; // площадь на одного человека**

**// Присвоить значения полям в объекте house,**

**house.Occupants = 4;**

**house.Area = 250;**

**house.Floors = 2;**

**// Вычислить площадь на одного человека.**

**areaPP = house.AreaPerPerson();**

**Console.WriteLine("Дом имеет:\n " +**

**house.Floors + " этажа\n " +**

**house.Occupants + " жильца\n " +**

**house.Area +**

**" кв. метров общей площади, из них\n " +**

**areaPP + " приходится на одного человека");**

**Console.ReadLine();**

**}**

**}**

**}**

**Вызов методов**

В следующем примере, **MathTest**, демонстрируется определение и создание экземпляров классов, а также определение и вызов методов. Помимо класса, содержащего метод**Main ()**, в нем определен класс по имени **MathTest**, содержащий набор методов и полей.

**using System;**

**using System.Collections.Generic;**

**using System.Linq;**

**using System.Text;**

**namespace MathProbe**

**{**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**// Попытка вызова некоторых статических функций**

**Console.WriteLine ("Pi равно " + MathTest.GetPi ());**

**int x = MathTest.GetSquareOf(5);**

**Console.WriteLine("5 в квадрате равно " + x) ;**

**// Создание объекта MathTest**

**MathTest math = new MathTest (); // это способ, которым в С# создаются**

**// экземпляры ссылочного типа**

**// Вызов нестатических методов**

**math.value = 30;**

**Console.WriteLine("Поле value переменной math содержит " + math.value);**

**Console.WriteLine("30 в квадрате равно " + math.GetSquare());**

**Console.ReadLine();**

**}**

**}**

**// Определение класса MathTest, методы которого мы вызываем**

**class MathTest**

**{**

**public int value;**

**public int GetSquare()**

**{**

**return value \* value;**

**}**

**public static int GetSquareOf(int x)**

**{**

**return x \* x;**

**}**

**public static double GetPi()**

**{**

**return 3.14159;**

**}**

**}**

**}**

Запуск на выполнение примера **MathTest** даст следующий результат:

**Pi равно 3.14159**

**5 в квадрате равно 25**

**Поле value переменной math содержит 30**

**30 в квадрате равно 900**

Как можно видеть в коде, класс **MathTest** содержит поле, которое хранит число, а также метод для вычисления квадрата этого числа. Кроме того, он включает два статических метода — один для возврата значения числа "пи" и другой — для вычисления квадрата числа, переданного в параметре.

Некоторые свойства этого класса на самом деле не могут служить примером правильного дизайна программы С#. Например, GetPi () следовало бы реализовать в виде константного поля, но остальная часть демонстрирует концепции, которые пока не рассматривались.

**Перегрузка методов**

В С# допускается совместное использование одного и того же имени двумя или более методами одного и того же класса, при условии, что их параметры объявляются по-разному. В этом случае говорят, что методы перегружаются, а сам процесс называется перегрузкой методов. Перегрузка методов относится к одному из способов реализации полиморфизма в С#.

В общем, для перегрузки метода достаточно объявить разные его варианты, а об остальном позаботится компилятор. Но при этом необходимо соблюсти следующее важное условие: тип или число параметров у каждого метода должны быть разными. Совершенно недостаточно, чтобы два метода отличались только типами возвращаемых значений. Они должны также отличаться типами или числом своих параметров. (Во всяком случае, типы возвращаемых значений дают недостаточно сведений компилятору С#, чтобы решить, какой именно метод следует использовать.) Разумеется, перегружаемые методы могут отличаться и типами возвращаемых значений. Когда вызывается перегружаемый метод, то выполняется тот его вариант, параметры которого соответствуют (по типу и числу) передаваемым аргументам.

Ниже приведен простой пример, демонстрирующий перегрузку методов.

**namespace ConsoleApplication1**

**{**

**class Overload**

**{**

**public void OvlDemo()**

**{**

**Console.WriteLine("Без параметров");**

**}**

**// Перегрузка метода OvlDemo с одним целочисленным параметром,**

**public void OvlDemo(int a)**

**{**

**Console.WriteLine("Один параметр: " + a);**

**}**

**// Перегрузка метода OvlDemo с двумя целочисленными параметрами,**

**public int OvlDemo(int a, int b)**

**{**

**Console.WriteLine("Два параметра: " + a + " " + b);**

**return a + b;**

**}**

**// Перегрузка метода OvlDemo с двумя параметрами типа double,**

**public double OvlDemo(double a, double b)**

**{**

**Console.WriteLine("Два параметра типа double: " + a + " "+ b);**

**return a + b;**

**}**

**}**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**Overload ob = new Overload();**

**int resI;**

**double resD;**

**// Вызвать все варианты метода OvlDemoO .**

**ob.OvlDemo() ;**

**Console.WriteLine ();**

**ob.OvlDemo (2);**

**Console.WriteLine ();**

**resI = ob.OvlDemo(4, 6);**

**Console.WriteLine("Результат вызова метода ob.OvlDemo(4, 6): " + resI);**

**Console.WriteLine ();**

**resD = ob.OvlDemo(1.1, 2.32);**

**Console.WriteLine("Результат вызова метода ob.OvlDemo(1.1, 2.32): " + resD);**

**Console.ReadLine();**

**}**

**}**

**}**

Вот к какому результату приводит выполнение приведенного выше кода.

**Без параметров**

**Один параметр: 2**

**Два параметра: 4 6**

**Результат вызова метода ob.OvlDemo (4, 6): 10**

**Два параметра типа double: 1.1 2.32**

**Результат вызова метода ob.OvlDemo (1.1, 2.32): 3.42**

**Конструкторы**

В приведенных выше примерах программ переменные экземпляра каждого объекта типа **Building** приходилось инициализировать вручную. Такой прием обычно не применяется в профессионально написанном коде С#. Кроме того, он чреват ошибками. Впрочем, существует лучший способ решить подобную задачу: воспользоваться конструктором. Конструктор инициализирует объект при его создании. У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу. Но у конструкторов нет возвращаемого типа, указываемого явно. Ниже приведена общая форма конструктора.

**доступ имя\_класса(список\_параметров) {**

**// тело конструктора**

**}**

Как правило, конструктор используется для задания первоначальных значений переменных экземпляра, определенных в классе, или же для выполнения любых других установочных процедур, которые требуются для создания полностью сформированного объекта. Кроме того, доступ обычно представляет собой модификатор доступа типа *public*, поскольку конструкторы зачастую вызываются в классе. А *список\_параметров* может быть как пустым, так и состоящим из одного или более указываемых параметров.

У всех классов имеются конструкторы, независимо от того, определите вы их или нет, поскольку в С# автоматически предоставляется конструктор, используемый по умолчанию и инициализирующий все переменные экземпляра их значениями по умолчанию. Для большинства типов данных значением по умолчанию является нулевое, для типа bool — значение false, а для ссылочных типов — пустое значение.

Но как только вы определите свой собственный конструктор, то конструктор по умолчанию больше не используется. Ниже приведен простой пример применения конструктора.

**using System;**

**using System.Collections.Generic;**

**using System.Linq;**

**using System.Text;**

**namespace ConstructorProbe**

**{**

**class MyClass**

**{**

**public int x;**

**public MyClass() //конструктор**

**{**

**x = 10;**

**}**

**}**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**MyClass tl = new MyClass();**

**MyClass t2 = new MyClass();**

**Console.WriteLine(tl.x + " " + t2.x);**

**Console.ReadLine();**

**}**

**}**

**}**

**Деструкторы**

Как было показано выше, при использовании оператора new свободная память для создаваемых объектов динамически распределяется из доступной буферной области оперативной памяти. Разумеется, оперативная память не бесконечна, и поэтому свободно доступная память рано или поздно исчерпывается. Это может привести к неудачному выполнению оператора new из-за нехватки свободной памяти для создания требуемого объекта. Именно по этой причине одной из главных функций любой схемы динамического распределения памяти является освобождение свободной памяти от неиспользуемых объектов, чтобы сделать ее доступной для последующего перераспределения. Во многих языках программирования освобождение распределенной ранее памяти осуществляется вручную. Например, в C++ для этой цели служит оператор delete. Но в С# применяется другой, более надежный подход: "сборка мусора".

Система "сборки мусора" в С# освобождает память от лишних объектов автоматически, действуя незаметно и без всякого вмешательства со стороны программиста. "Сборка мусора" происходит следующим образом. Если ссылки на объект отсутствуют, то такой объект считается ненужным, и занимаемая им память в итоге освобождается и накапливается. Эта утилизированная память может быть затем распределена для других объектов.

"Сборка мусора" происходит лишь время от времени по ходу выполнения программы. Она не состоится только потому, что существует один или более объектов, которые больше не используются. Следовательно, нельзя заранее знать или предположить, когда именно произойдет "сборка мусора".

В языке С# имеется возможность определить метод, который будет вызываться непосредственно перед окончательным уничтожением объекта системой "сборки мусора". Такой метод называется деструктором и может использоваться в ряде особых случаев, чтобы гарантировать четкое окончание срока действия объекта. Например, деструктор может быть использован для гарантированного освобождения системного ресурса, задействованного освобождаемым объектом. Следует, однако, сразу же подчеркнуть, что деструкторы — весьма специфические средства, применяемые только в редких, особых случаях. И, как правило, они не нужны. Но здесь они рассматриваются вкратце ради полноты представления о возможностях языка С#. Ниже приведена общая форма деструктора:

**~имя\_класса() {**

**// код деструктора**

**}**

где имя\_класса означает имя конкретного класса.

Следовательно, деструктор объявляется аналогично конструктору, за исключением того, что перед его именем указывается знак "тильда" (~). Обратите внимание на то, что у деструктора отсутствуют возвращаемый тип и передаваемые ему аргументы.

Для того чтобы добавить деструктор в класс, достаточно включить его в класс в качестве члена. Он вызывается всякий раз, когда предполагается утилизировать объект его класса. В деструкторе можно указать те действия, которые следует выполнить перед тем, как уничтожать объект. Следует, однако, иметь в виду, что деструктор вызывается непосредственно перед "сборкой мусора". Он не вызывается, например, в тот момент, когда переменная, содержащая ссылку на объект, оказывается за пределами области действия этого объекта. (В этом отношении деструкторы в С# отличаются от деструкторов в C++, где они вызываются в тот момент, когда объект оказывается за пределами области своего действия.) Это означает, что заранее нельзя знать, когда именно следует вызывать деструктор. Кроме того, программа может завершиться до того, как произойдет "сборка мусора", а следовательно, деструктор может быть вообще не вызван.

Ниже приведен пример программы, демонстрирующий применение деструктора. В этой программе создается и уничтожается большое число объектов. В какой-то момент по ходу данного процесса активизируется "сборка мусора" и вызываются деструкторы для уничтожения ненужных объектов.

**namespace ConsoleApplication1**

**{**

**class Destruct {**

**public int x;**

**public Destruct(int i) {**

**x = i;**

**}**

**// Вызывается при утилизации объекта.**

**~Destruct() {**

**Console.WriteLine("Уничтожить " + x);**

**}**

**// Создает объект и тут же уничтожает его.**

**public void Generator(int i) {**

**Destruct о = new Destruct (i);**

**}**

**}**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**int count;**

**Destruct ob = new Destruct (0);**

**/\* А теперь создать большое число объектов.**

**В какой-то момент произойдет "сборка мусора".**

**Примечание: для того чтобы активизировать**

**"сборку мусора", возможно, придется увеличить**

**число создаваемых объектов. \*/**

**for (count=1; count < 100000; count++)**

**ob.Generator(count);**

**Console.WriteLine("Готово!");**

**Console.ReadLine();**

**}**

**}**

**}**

Эта программа работает следующим образом. Конструктор инициализирует переменную х известным значением. В данном примере переменная х служит в качестве идентификатора объекта. А деструктор выводит значение переменной х, когда объект утилизируется. Особый интерес вызывает метод Generator (), который создает и тут же уничтожает объект типа Destruct. Сначала в создается исходный объект ob типа Destruct, а затем осуществляется поочередное создание и уничтожение 100 тыс. объектов. В разные моменты этого процесса происходит "сборка мусора". Насколько часто она происходит — зависит от нескольких факторов, в том числе от первоначального объема свободной памяти, типа используемой операционной системы и т.д. Тем не менее в какой-то момент начинают появляться сообщения, формируемые деструктором. Если же они не появятся до окончания программы, т.е. до того момента, когда будет выдано сообщение "Готово!", попробуйте увеличить число создаваемых объектов, повысив предельное количество подсчитываемых шагов в цикле for. И еще одно важное замечание: метод WriteLine () вызывается в деструкторе -Destruct () исключительно ради наглядности данного примера его использования.

Как правило, деструктор должен воздействовать только на переменные экземпляра, определенные в его классе. В силу того что порядок вызова деструкторов не определен точно, их не следует применять для выполнения действий, которые должны происходить в определенный момент выполнения программы.

**Свойства**

Еще одной разновидностью члена класса является свойство. Как правило, свойство сочетает в себе поле с методами доступа к нему. Как было показано в приведенных ранее примерах программ, поле зачастую создается, чтобы стать доступным для пользователей объекта, но при этом желательно сохранить управление над операциями, разрешенными для этого поля, например, ограничить диапазон значений,

присваиваемых данному полю. Этой цели можно, конечно, добиться и с помощью закрытой переменной, а также методов доступа к ее значению, но свойство предоставляет более совершенный и рациональный путь для достижения той же самой цели. Свойства очень похожи на индексаторы. В частности, свойство состоит из имени и аксессоров **get** и **set**. Аксессоры служат для получения и установки значения переменной. Главное преимущество свойства заключается в том, что его имя может быть использовано в выражениях и операторах присваивания аналогично имени обычной переменной, но в действительности при обращении к свойству по имени автоматически вызываются его аксессоры **get** и **set**. Аналогичным образом используются аксессоры **get** и **set** индексатора.

**using System;**

**using System.Collections.Generic;**

**using System.Linq;**

**using System.Text;**

**namespace PropertiesProbe**

**{**

**class SimpProp**

**{**

**int prop; // поле, управляемое свойством МуРгор**

**public SimpProp() { prop = 0; } //Конструктор**

**/\* Это свойство обеспечивает доступ к закрытой переменной экземпляра prop.**

**Оно допускает присваивание только положительных значений. \*/**

**public int MyProp**

**{**

**get**

**{**

**return prop;**

**}**

**set**

**{**

**if (value >= 0) prop = value;**

**}**

**}**

**}**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**SimpProp ob = new SimpProp();**

**Console.WriteLine("Первоначальное значение ob.МуРгор: " + ob.MyProp );**

**ob.MyProp = 100; // присвоить значение**

**Console.WriteLine("Текущее значение ob.МуРгор: " + ob.MyProp);**

**// Переменной prop нельзя присвоить отрицательное значение.**

**Console.WriteLine("Попытка присвоить значение " +**

**"-10 свойству ob.МуРгор");**

**ob.MyProp = -10;**

**Console.WriteLine("Текущее значение ob.МуРгор: " + ob.MyProp);**

**Console.ReadLine();**

**}**

**}**

**}**

**Статические классы**

Если класс не содержит ничего кроме статических методов и свойств, этот класс сам по себе может стать статическим. Статический класс функционально представляет собой то же самое, что и класс с приватным статическим конструктором. Создать экземпляр такого класса невозможно. Если указать ключевое слово *static* в объявлении класса, компилятор будет гарантировать, что к этому классу никогда не будут добавлены нестатические члены. В противном случае будет выдана ошибка компиляции. Это гарантирует также, что экземпляры этого класса никогда не будут созданы. Синтаксис объявления статического класса выглядит следующим образом:

**static class StaticUtilities**

**{**

**public static void HelperMethod ()**

**{**

**}**

**}**

Объект типа *StaticUtilities* не нужен для вызова *HelperMethod()*. При вызове указывается имя типа: *StaticUtilities.HelperMethod();* .

**Модификаторы**

Модификаторы могут указывать видимость метода, как, например *public* или *private*, или же их природу, например, *virtual* или *abstract*. В языке С# определено множество модификаторов, и

сейчас стоит потратить некоторое время на ознакомление с их полным списком.

**Модификаторы видимости**

Модификаторы видимости указывают, какие другие единицы кода могут видеть элемент

|  |  |  |
| --- | --- | --- |
| Модификатор | К чему относится | Описание |
| public | К любым типам или членам | Элемент видим в любом другом коде |
| protected | к любому члену типа, а также к любому вложенному типу | Элемент видим только любому производному типу |
| internal | к любым типам или членам | Элемент видим только в пределах включающей его сборки |
| private | К любому члену типа, а также к любому вложенному типу | Элемент видим только в пределах типа, которому он принадлежит |
| protected internal | к любому члену типа, а также к любому вложенному типу | Элемент видим только в пределах включающей его сборки, а также в любом коде внутри производного типа |

Определения типа могут быть общедоступными или приватными в зависимости от того, хотите ли вы обеспечить его видимость извне сборки. Указывать модификаторы *protected*, private или *protected internal* для типов нельзя, поскольку эти уровни видимости не имеют смысла для типа, находящегося в пространстве имен. Это значит, что они могут относиться только к членам. Однако возможно создавать вложенные типы (т.е. типы, содержащиеся внутри других типов) с такой видимостью, поскольку в этом случае типы имеют статус члена. Таким образом, приведенный ниже код вполне корректен:

**public class OuterClass**

**{**

**protected class InnerClass**

**{**

**// и т.д.**

**}**

**// и т.д.**

**}**

Если есть вложенный тип, он всегда может иметь доступ ко всем членам внешнего типа. Таким образом, в последнем примере любой код внутри InnerClass всегда имеет доступ ко всем членам OuterClass, даже если они объявлены как private.

**Другие модификаторы**

Модификаторы, перечисленные ранее, могут быть применены к членам типов и характеризуются различным использованием. Некоторые из них также имеет смысл использовать для типов.

|  |  |  |
| --- | --- | --- |
| Модификатор | К чему относится | Описание |
| new | К функциям-членам | Член скрывает унаследованный член с той же сигнатурой |
| static | Ко всем членам | Член не связан с конкретным экземпляром класса |
| virtual | Только к классам и функциям-членам | Член может быть переопределен в классах - наследниках |
| abstract | Только к функциям-членам | Виртуальный член, определяющий сигнатуру, но не предоставляющий реализации |
| override | Только к функциям-членам | Член переопределяет унаследованный виртуальный или абстрактный член базового класса |
| sealed | К классам, методам и свойствам | Для классов означает, что от таких классов нельзя наследовать. Для свойств и методов — член переопределяет унаследованный виртуальный член, но не может быть переопределен ни одним членом производных классов. Должен применяться в сочетании с override |
| extern | Только к статическим методам [Dllimport] | Член реализован внешне, на другом языке |

**Наследование**

В объектно-ориентированном программировании (ООП) существуют два различных типа наследования: наследование реализации и наследование интерфейса.

1. Наследование реализации (implementation inheritance) означает, что тип происходит от базового типа, получая от него все поля-члены и функции-члены. При наследовании реализации производный тип адаптирует реализацию каждой функции базового типа, если только в его определении не указано, что реализация функции должна быть переопределена. Такой тип наследования более полезен, когда нужно добавить функциональность к существующему типу или же когда несколько связанных типов разделяют существенный объем общей функциональности.

2. Наследование интерфейса (interface inheritance) означает, что тип наследует только сигнатуру функций, но не наследует никакой реализации. Этот тип наследования наиболее полезен, когда нужно специфицировать, что тип обеспечивает доступ к определенным средствам.

В С# поддерживается как наследование реализации, так и наследование интерфейса. Оба типа наследования полностью встроены в язык с самого начала, позволяя принимать решение о том, какой из них использовать, на основе архитектуры приложения.

Некоторые языки, такие как C++, поддерживают то, что известно под названием *множественного наследования*, когда класс происходит более чем от одного базового класса. Преимущества множественного наследования спорны. С одной стороны, нет сомнений, что можно применять множественное наследование для написания чрезвычайно сложного, но при этом компактного кода, что демонстрирует библиотека C++ ATL. С другой стороны, код, использующий множественное наследование, часто трудно понять и сложно отлаживать (все это также демонстрирует библиотека C++ ATL). Как уже упоминалось, облегчение написания устойчивого кода было одной из ключевых целей проектирования С#. Соответственно, поэтому в С# множественное наследование не поддерживается. Однако С# позволяет типу наследовать множество интерфейсов. Это значит, что класс С# может наследоваться от другого класса и любого количества интерфейсов. На самом деле можно сказать точнее: благодаря наличию System.Object как всеобщего базового типа, каждый класс С# (за исключением Object) имеет строго один базовый класс и дополнительно может иметь любое количество базовых интерфейсов.

**Наследование реализации**

Для объявления, что класс наследуется от другого класса, применяется следующий синтаксис:

**class УнаследованныйКласс: БазовыйКласс**

**{**

**//Данные-члены и функции-члены**

**}**

Этот синтаксис очень похож на синтаксис C++ и Java. Однако программисты на C++, знакомые с концепцией общедоступного и приватного наследования, должны обратить внимание, что С# не поддерживает приватного наследования; этим объясняется отсутствие квалификатора public или private перед именем базового класса. Поддержка приватного наследования значительно усложняет язык, при этом принося весьма небольшую выгоду. На практике приватное наследование в C++ все равно используется чрезвычайно редко. Если класс также наследует интерфейсы, то список базового класса и интерфейсов

разделяется запятыми:

**public class MyDerivedClass: MyBaseClass, Ilnterfacel, IInterface2**

**{**

**// и т.д.**

**}**

Для структур синтаксис выглядит так:

**public struct MyDerivedStruct: Ilnterfacel, IInterface2**

**{**

**// и т.д.**

**}**

Если при определении класса базовый класс не указан, то компилятор С# предполагает, что базовым классом является System.Object. Поэтому следующие два фрагмента кода эквивалентны:

**class MyClass: Object // наследуется от System.Object**

**{**

**// и т.д.**

**}**

и

**class MyClass // наследуется от System.Object**

**{**

**// и т.д.**

**}**

Для простоты чаще применяется вторая форма.

Поскольку в С# поддерживается ключевое слово *object*, служащее псевдонимом класса *System.Object*, можно записать и так:

**class MyClass: object // наследуется от System.Object**

**{**

**// и т.д.**

**}**

Чтобы сослаться на класс Object, используйте ключевое слово object, которое распознается интеллектуальными редакторами вроде Visual Studio .NET. Это облегчит редактирование кода.

**Виртуальные методы**

Объявляя функцию базового класса как *virtual*, вы тем самым позволяете ее переопределять в классах-наследниках:

**class MyBaseClass**

**{**

**public virtual string VirtualMethod ()**

**{**

**return "Это - виртуальный метод, определенный в MyBaseClass";**

**}**

**}**

Также допускается объявление свойства как *virtual*. Для виртуального или переопределенного свойства используется такой же синтаксис, что и для невиртуального свойства, за исключением ключевого слова *virtual*, добавляемого к определению. Синтаксис выглядит следующим образом:

**public virtual string ForeName**

**{**

**get { return foreName; }**

**set { foreName = value; }**

**}**

**private string foreName;**

Для простоты далее речь пойдет в основном о методах, хотя все сведения касаются также и свойств. Концепция, лежащая в основе виртуальных функций С#, идентична стандартной концепции ООП. Виртуальную функцию можно переопределить в классе-наследнике, и когда этот метод будет вызван, запустится его версия, относящаяся к соответствующему типу объекта. В С# по умолчанию функции не являются виртуальными, но (в отличие от конструкторов) могут быть явно объявлены как virtual. Это следует методологии C++: по причинам, связанным с производительностью, функции не виртуальные, если это не указано явно. В отличие от этого, в Java все функции виртуальные. С# имеет отличающийся от C++ синтаксис, поскольку требует явного объявления, когда функция класса-наследника переопределяет другую функцию, с помощью ключевого слова override:

**class MyDerivedClass: MyBaseClass**

**{**

**public override string VirtualMethod()**

**{**

**return "Этот переопределенный метод объявлен в MyDerivedClass";**

**}**

**}**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**MyBaseClass x = new MyBaseClass();**

**Console.WriteLine(x.VirtualMethod());**

**MyDerivedClass y = new MyDerivedClass();**

**Console.WriteLine(y.VirtualMethod());**

**Console.ReadLine();**

**}**

**}**

Этот синтаксис переопределения метода исключает потенциальные ошибки времени выполнения, которые могут легко возникать в C++, когда сигнатура метода в классе-наследнике непреднамеренно оказывается отличной от базовой версии, в результате чего метод наследника не может переопределить базовый метод. В С# это всплывает в виде ошибки компиляции, поскольку компилятор легко обнаруживает метод, для которого указан модификатор override, но при этом не имеющий базового метода, который он переопределяет.

Ни поля-члены, ни статические функции не могут быть объявлены виртуальными. Эта концепция просто не имеет смысла ни для каких членов класса, за исключением функций-членов уровня экземпляра.

**Вызов базовых версий функций**

В С# предусмотрен специальный синтаксис вызова базовых версий метода из производного класса: *base. <ИмяМетода> ()*. Например, если необходимо, чтобы метод производного класса возвращал 90% значения, возвращенного методом базового класса, можете воспользоваться следующим синтаксисом:

**class CustomerAccount**

**{**

**public virtual decimal CalculatePrice ()**

**{**

**// реализация**

**return 30.0M;**

**}**

**}**

**class GoldAccount: CustomerAccount**

**{**

**public override decimal CalculatePrice ()**

**{**

**return base.CalculatePrice () \* 0.9M;**

**}**

**}**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**GoldAccount x = new GoldAccount();**

**Console.WriteLine(x.CalculatePrice());**

**Console.ReadLine();**

**}**

**}**

Отметим, что синтаксис *base.<ИмяМетода>()* можно использовать для вызова любого метода базового класса — вы не обязаны вызывать его только из переопределенной версии того же метода.

**Абстрактные классы и функции**

Язык С# позволяет объявлять абстрактными и классы, и функции. Создавать экземпляры абстрактных классов нельзя, поскольку абстрактные функции не имеют реализации и должны быть переопределены в любом неабстрактном классе-наследнике. Очевидно, что абстрактные функции автоматически являются виртуальными (хотя вы не должны применять ключевое слово virtual, так как это приведет к синтаксической ошибке). Если любой класс содержит любую абстрактную функцию, этот класс сам является абстрактным и должен быть объявлен таковым:

**abstract class Building**

**{**

**public abstract decimal CalculateHeatingCost (); // абстрактный метод**

**}**

**class MyBuilding : Building**

**{**

**public override decimal CalculateHeatingCost()**

**{**

**return 35.2M;**

**}**

**}**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**MyBuilding x = new MyBuilding();**

**Console.WriteLine(x.CalculateHeatingCost());**

**Console.ReadLine();**

**}**

**}**

Разработчики на C++ отметят также отличие в терминологии: в C++ абстрактные функции часто описываются как чистые виртуальные; в мире же С# единственный корректный термин для них — абстрактные.

**Запечатанные классы и методы**

С# позволяет объявлять классы и методы как *sealed* (запечатанные). В случае класса это значит, что наследовать от него нельзя. В случае метода это означает невозможность его переопределения.

**sealed class FinalClass**

**{**

**// и т.д.**

**}**

**class DerivedClass: FinalClass // Неверно. Ошибка компиляции.**

**{**

**// и т.д.**

**}**

Наиболее вероятная ситуация, когда может понадобиться пометить класс или метод как *sealed* — это когда класс или метод обеспечивает внутренние действия библиотеки, класса или других разрабатываемых классов, поэтому вы уверены, что любая попытка переопределить некоторую его функциональность приведет к нестабильности кода. Также можно помечать класс или метод как sealed из коммерческих соображений, чтобы предотвратить использование классов способом, противоречащим лицензионным соглашениям. Вообще говоря, нужно быть осторожным с объявлением классов как sealed, потому что, поступая так, вы в некоторой степени ограничиваете возможности их использования. Даже если вы не думаете, что понадобится наследовать от класса или переопределять его члены, все же существует вероятность, что в какой-то момент в будущем кто-то столкнется с ситуацией, которую вы не смогли предвидеть. В библиотеке базовых классов .NET часто встречаются запечатанные классы. Это защищает их от независимых разработчиков, которые могут пожелать унаследовать от них собственные классы. Так, например, string — запечатанный класс. Объявление метода *sealed* служит той же цели, что и для класса:

**class MyClass: MyClassBase**

**{**

**public sealed override void FinalMethod()**

**{**

**// и т.д.**

**}**

**}**

**class DerivedClass: MyClass**

**{**

**public override void FinalMethod () // Неверно. Ошибка компиляции.**

**{**

**}**

**}**

Для того чтобы можно было применить ключевое слово *sealed* к методу или свойству, они должны быть сначала переопределены по отношению к базовому классу. Если вы не хотите, чтобы метод или свойство базового класса переопределялось, то просто не помечайте их как virtual.

**Интерфейсы**

Как упоминалось ранее, наследуя интерфейс, класс тем самым декларирует, что он реализует определенные функции. Поскольку не все объектно-ориентированные языки поддерживают интерфейсы, в этом разделе подробно описана реализация интерфейсов С#. В этом разделе интерфейсы рассматриваются путем представления полного определения одного из интерфейсов от Microsoft — System. IDisposable. Интерфейс IDisposable содержит один метод Dispose (), предназначенный для реализации классами, которые осуществляют очистку кода:

**public interface IDisposable**

**{**

**void Dispose();**

**}**

Этот фрагмент показывает, что объявление интерфейса синтаксически очень похоже на объявление абстрактного класса. Однако вы должны знать, что ни для одного из членов интерфейса не допускается какой-либо реализации. В общем случае, интерфейс может содержать только объявления методов, свойств, индексов и событий. Создавать экземпляр интерфейса нельзя — он содержит только сигнатуры своих членов. Интерфейс не имеет конструкторов (как можно сконструировать нечто, экземпляр чего не создается?), равно как и полей (поскольку это подразумевает некоторую внутреннюю реализацию). Определению интерфейса также не разрешено содержать перегрузки операций, причем не потому, что с этим связаны какие-то принципиальные проблемы. Причина в том, что назначение интерфейсов состоит в том, чтобы служить общедоступными контрактами, для которых перегрузка операций вызывала бы определенные

проблемы совместимости с другими языками .NET, такими как Visual Basic и .NET, которые не поддерживают перегрузку операций.

Также при определении членов интерфейса не разрешены модификаторы. Члены интерфейса всегда неявно являются *public* и не могут быть *virtual* или *static*. Это оставлено на усмотрение реализаций классов. Таким образом, вполне нормально указывать модификаторы доступа к членам интерфейса в реализующих их классах, что и делается в примерах настоящего раздела.

Рассмотрим, например, интерфейс IDisposable. Если класс пожелает объявить, что он реализует метод Dispose (), то он должен будет реализовать интерфейс IDisposable, что в терминах С# означает, что он наследуется от IDisposable.

**class SomeClass: IDisposable**

**{**

**// Этот класс ДОЛЖЕН содержать реализацию**

**// метода IDisposable.Dispose (), иначе**

**// возникнет ошибка компиляции.**

**public void Dispose ()**

**{**

**// реализация метода Dispose ()**

**}**

**// остальная часть класса**

**}**

В этом примере, если SomeClass будет наследовать IDisposable, но не будет содержать реализации Dispose (), в точности совпадающей с сигнатурой, определенной в IDisposable, будет выдана ошибка компиляции, поскольку в этом случае класс нарушит контракт реализации интерфейса IDisposable. Разумеется, для компилятора не будет никакой проблемы, если встретится класс, включающий метод Dispose (), но не унаследованный от IDisposable. Проблема будет в том, что другой код не будет иметь возможности распознать, что SomeClass согласен поддерживать средства IDisposable. IDisposable — сравнительно простой интерфейс, потому что в нем определен только один метод. Большинство интерфейсов содержат гораздо большее количество методов.

**Определение и реализация интерфейсов**

В этом разделе показано, как определять и использовать интерфейсы при разработке короткой программы, реализующей парадигму наследования интерфейсов. Пример описывает банковский счет. Предположим, что вы пишете код, который в конечном итоге обеспечит компьютеризованный перевод денег между банковскими счетами. Пусть существует множество компаний, которые могут реализовывать банковские счета, но все они согласились с тем, что любые классы, представляющие банковские счета, должны реализовывать интерфейс IBankAccount, предусматривающий методы для внесения и съема денежных сумм, а также свойство, возвращающее баланс. Это тот интерфейс, который позволит внешнему коду распознавать различные классы банковских счетов, реализующие различные формы таких счетов. Хотя целью этого является обеспечение взаимодействия банковских счетов между собой для перевода денег, пока мы не будем представлять эту возможность.

В целях упрощения поместим весь код примера в единственный исходный файл. Конечно, если что-то вроде этого примера придется делать в реальной жизни, то можно догадаться, что различные классы банковских счетов не только будут компилироваться в различные сборки, но также будут развернуты на разных компьютерах, принадлежащих разным банкам. Пока все это чересчур сложно для немедленного рассмотрения. Однако чтобы внести определенную долю реализма, определим разные пространства имен для разных компаний.

Для начала потребуется определить интерфейс IBankAccount:

**namespace Wrox.ProCSharp**

**{**

**public interface IBankAccount**

**{**

**void Payln(decimal amount);**

**bool Withdraw(decimal amount);**

**decimal Balance**

**{**

**get;**

**}**

**}**

**}**

Обратите внимание на имя интерфейса — IBankAccount. Существует соглашение, что имя интерфейса традиционно начинается с буквы I, чтобы сразу было понятно, что это интерфейс.

В большинстве случаев руководство по .NET отвергает так называемую венгерскую нотацию, согласно которой имена предваряются буквой, указывающей на тип определяемого объекта. Интерфейсы — одно из исключений из этого правила, в котором венгерская нотация как раз рекомендуется. Идея заключается в том, что теперь имеется возможность писать классы, представляющие банковские счета. Эти классы не должны быть как-то связанными друг с другом; они могут быть полностью различными. Однако все они декларируют свое представление банковских счетов тем, что реализуют интерфейс IBankAccount.

Начнем с первого класса, описывающего сберегательный счет в Королевском Банке Венеры:

**namespace Wrox.ProCSharp.VenusBank**

**{**

**public class SaverAccount: IBankAccount**

**{**

**private decimal balance;**

**public void Payln(decimal amount)**

**{**

**balance += amount;**

**}**

**public bool Withdraw(decimal amount)**

**{**

**if (balance >= amount)**

**{**

**balance -= amount;**

**return true;**

**}**

**Console. WriteLine ("Попытка перевода денег не удалась.");**

**return false;**

**}**

**public decimal Balance**

**{**

**get**

**{**

**return balance;**

**}**

**}**

**public override string ToString()**

**{**

**return String.Format("Сберегательный Банк Венеры: Баланс = {0,6:С}", balance);**

**}**

**}**

**}**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**IBankAccount venusAccount = new SaverAccount();**

**venusAccount.Payln(200);**

**venusAccount.Withdraw(100);**

**Console.WriteLine(venusAccount.ToString());**

**Console.ReadLine();**

**}**

**}**

Достаточно очевидно, что делает реализация этого класса. Вы создаете приватное поле balance и изменяете сумму остатка, указанную в нем, при съеме и зачислении денег на счет. Если предпринимается попытка снять больше денег, чем осталось на счету, выдается сообщение об ошибке. Обратите внимание, что для простоты кода здесь не реализуются дополнительные свойства, такие как имя владельца счета. В реальной жизни эта информация совершенно необходима, но в данный пример это внесло бы излишнюю сложность. Единственной интересной строкой в этом коде является объявление класса: *public class SaverAccount: IBankAccount* . Она объявляет, что SaverAccount наследуется от одного интерфейса, IBankAccount, и никакого другого базового класса не указано (что, конечно же, означает, что SaverAccount наследуется от System.Object). Кстати говоря, наследование интерфейсов совершенно независимо от наследования классов.

То, что SaverAccount наследуется от IBankAccount, означает, что у SaverAccount есть все члены IBankAccount. Но поскольку сам интерфейс не реализует ни одного из своих методов, SaverAccount должен предоставить для них собственную реализацию. Если реализация любого из них опущена, компилятор это заметит и выдаст соответствующее уведомление. Вспомните также о том, что интерфейс просто указывает на присутствие своих членов. Решение о том, объявлять их virtual или abstract, возложено на класс (хотя абстрактные функции, конечно же, допускаются только в абстрактных классах). Что касается конкретного примера, то здесь нет никаких причин объявлять любую из функций интерфейса виртуальной. Чтобы проиллюстрировать, как различные классы могут реализовать один и тот же интерфейс, предположим, что Планетарный Банк Юпитера также реализует собственный класс, представляющий банковские счета — GoldAccount:

**namespace Wrox.ProCSharp.JupiterBank**

**{**

**public class GoldAccount: IBankAccount**

**{**

**// и т.д.**

**}**

**}**

Мы не будем здесь рассматривать детали класса GoldAccount; в коде примера он будет практически идентичен реализации SaverAccount. Подчеркнем, что GoldAccount не имеет никакой связи с SaverAccount кроме того, что оба они реализуют один и тот же интерфейс.

Теперь, имея готовые классы, их можно протестировать. Первым делом, понадобится несколько операторов using:

**using System;**

**using Wrox.ProCSharp;**

**using Wrox.ProCSharp.VenusBank;**

**using Wrox.ProCSharp.JupiterBank;**

**namespace Wrox.ProCSharp**

**{**

**class MainEntryPoint**

**{**

**static void Main()**

**{**

**IBankAccount venusAccount = new SaverAccount ();**

**IBankAccount jupiterAccount = new GoldAccount ();**

**venusAccount.Payln(200);**

**venusAccount.Withdraw(100) ;**

**Console.WriteLine(venusAccount.ToString());**

**jupiterAccount.Payln(500) ;**

**jupiterAccount.Withdraw(600);**

**jupiterAccount.Withdraw(100);**

**Console.WriteLine(jupiterAccount.ToString());**

**}**

**}**

**}**

Этот код сгенерирует следующий вывод:

**Сберегательный Банк Венеры: Баланс = £100.00**

**Попытка перевода денег не удалась.**

**Планетарный Банк Юпитера: Баланс = £400.00**

Главный момент, который следует здесь отметить — способ объявления обеих переменных как ссылок на IBankAccount. Это значит, что они могут указывать на любой экземпляр любого класса, реализующего интерфейс. Это также означает, что через эти ссылки можно вызывать только те методы, которые являются частью интерфейса. Если понадобится вызвать любые методы, реализованные классом, но не являющиеся частью интерфейса, то придется выполнить приведение ссылки к соответствующему типу. В примере кода вызывается метод ToString () (не объявленный в IBankAccount) без какого-либо явного приведения, просто потому, что ToString () — это метод System.Object, поэтому компилятор С# знает о том, что он поддерживается любым классом (иначе говоря, приведение любого интерфейса к System.Object осуществляется неявно).

Ссылки на интерфейсы во всех отношениях могут трактоваться как ссылки на классы — однако мощь интерфейсных ссылок в том, что они могут указывать на любые классы, реализующие данный интерфейс. Например, это позволяет формировать массивы интерфейсов, элементы которых являются объектами разных классов:

**IBankAccount [ ] accounts = new IBankAccount [2] ;**

**accounts [0] = new SaverAccount ();**

**accounts [1] = new GoldAccount () ;**

Однако если попытаться сделать что-то вроде такого:

**accounts [1] = new SomeOtherClass (); // SomeOtherClass не реализует**

**// IBankAccount: НЕВЕРНО!!**

будет получена следующая ошибка компиляции: Cannot implicitly convert type 'Wrox.ProCSharp.SomeOtherClass' to 'Wrox. ProCSharp.IBankAccount' Неявное преобразование типа 'Wrox.ProCSharp.SomeOtherClass' в 'Wrox.ProCSharp. IBankAccount' невозможно