Иногда требуется создать базовый класс, в котором определяется лишь самая общая

форма для всех его производных классов, а наполнение ее деталями предоставляется

каждому из этих классов. В таком классе определяется лишь характер методов,

которые должны быть конкретно реализованы в производных классах, а не в самом базовом

классе. Подобная ситуация возникает, например, в связи с невозможностью получить

содержательную реализацию метода в базовом классе. Именно такая ситуация

была продемонстрирована в варианте класса TwoDShape из предыдущего примера,

где метод Area() был просто определен как заполнитель. Такой метод не вычисляет и

не выводит площадь двумерного объекта любого типа.

# Абстрактный метод

***Абстрактный метод***создается с помощью указываемого модификатора типа

***abstract***. У абстрактного метода отсутствует тело, и поэтому он не реализуется в базовом

классе. Это означает, что он должен быть переопределен в производном классе,

поскольку его вариант из базового класса просто непригоден для использования. ***Нетрудно***

***догадаться, что абстрактный метод автоматически становится виртуальным и***

***не требует указания модификатора virtual. В действительности совместное использование***

***модификаторов virtual и abstract считается ошибкой***.

Для определения абстрактного метода служит приведенная ниже общая форма.

***abstract тип имя(список\_параметров);***

Как видите, у абстрактного метода отсутствует тело. Модификатор abstract может

применяться только в методах экземпляра, но не в статических методах (static).

Абстрактными могут быть также индексаторы и свойства.

Класс, содержащий один или больше абстрактных методов, должен быть также

объявлен как абстрактный, и для этого перед его объявлением class указывается модификатор

abstract. А поскольку реализация абстрактного класса не определяется

полностью, то у него не может быть объектов. Следовательно, попытка создать объект абстрактного класса с помощью оператора new приведет к ошибке во время компиляции.

Когда производный класс наследует абстрактный класс, в нем должны быть реализованы

***все*** ***абстрактные методы базового класса***. ***В противном случае производный***

***класс должен быть также определен как abstract***. Таким образом, атрибут abstract

наследуется до тех пор, пока не будет достигнута полная реализация класса.

Используя абстрактный класс, мы можем усовершенствовать рассматривавшийся

ранее класс TwoDShape. Для неопределенной двухмерной фигуры понятие площади

не имеет никакого смысла, поэтому в приведенном ниже варианте класса TwoDShape

метод Area() и сам класс TwoDShape объявляются как abstract. Это, конечно, означает,

что во всех классах, производных от класса TwoDShape, должен быть переопределен

метод Area().

(***glava11\_13***)

abstract class TwoDShape

{

…

public void ShowDim()

{

Console.WriteLine("Height and Width " + Width + " " + Height);

}

public abstract double Area();

}

Как показывает представленный выше пример программы, во всех производных

классах метод Area() должен быть непременно переопределен, а также объявлен абстрактным.

Обратите также внимание на то, что в класс TwoDShape по-прежнему входит метод

ShowDim() и что он не объявляется с модификатором abstract. В абстрактные классы

вполне допускается (и часто практикуется) включать конкретные методы, которые

могут быть использованы в своем исходном виде в производном классе. А переопределению

в производных классах подлежат только те методы, которые объявлены как

abstract.

# Предотвращение наследования с помощью

# ключевого слова sealed

Несмотря на всю эффективность и полезность наследования, иногда возникает потребность

предотвратить его.

Для того чтобы предотвратить наследование класса, достаточно указать ключевое

слово sealed перед определением класса. Как и следовало ожидать, класс не допускается

объявлять одновременно как abstract и sealed, поскольку сам абстрактный

класс реализован не полностью и опирается в этом отношении на свои производные

классы, обеспечивающие полную реализацию.

sealed class А

{

// ...

}

// Следующий класс недопустим.

class В : A ( // ОШИБКА! Наследовать класс А нельзя

// ...

}

Как следует из комментариев в приведенном выше фрагменте кода, класс В не может

наследовать класс А, потому что последний объявлен как sealed.

И еще одно замечание: ключевое слово sealed может быть также использовано

в виртуальных методах для предотвращения их дальнейшего переопределения. Допустим,

что имеется базовый класс В и производный класс D. Метод, объявленный

в классе В как virtual, может быть объявлен в классе D как sealed. Благодаря этому

в любом классе, наследующем от класса предотвращается переопределение данного

метода.

class В

{

public virtual void MyMethod() { /\* ... \*/ }

}

class D : В

{

// Здесь герметизируется метод MyMethod() и

// предотвращается его дальнейшее переопределение.

sealed public override void MyMethod() { /\* ... \*/ }

}

class X : D

{

// Ошибка! Метод MyMethodO герметизирован!

public override void MyMethod() { /\* ... \*/ }

}

Метод MyMethod() герметизирован в классе D, и поэтому не может быть переопределен

в классе X.

# Класс object

В C# предусмотрен специальный класс object, который неявно считается базовым

классом для всех остальных классов и типов, включая и типы значений. Иными словами,

все остальные типы являются производными от object. Это, в частности, означает,

что переменная ссылочного типа object может ссылаться на объект любого другого

типа. Кроме того, переменная типа object может ссылаться на любой массив, поскольку

в C# массивы реализуются как объекты. Формально имя object считается в

C# еще одним обозначением класса System.Object, входящего в библиотеку классов

для среды .NET Framework.

В классе object определяются методы, приведенные в табл. 11.1. Это означает, что

они доступны для каждого объекта.

Некоторые из этих методов требуют дополнительных пояснений. По умолчанию

метод Equals(object) определяет, ссылается ли вызывающий объект на тот же самый

объект, что и объект, указываемый в качества аргумента этого метода, т.е. он определяет,

являются ли обе ссылки одинаковыми. ***Метод Equals(object) возвращает***

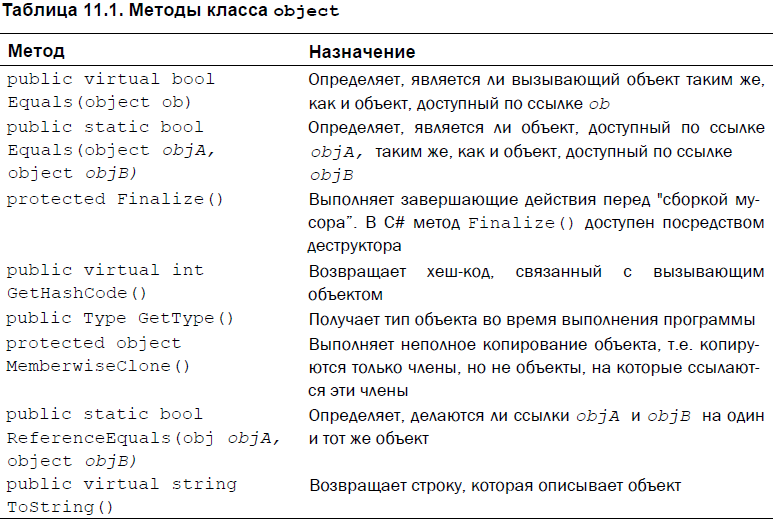
***логическое значение true, если сравниваемые объекты одинаковы, в противном случае***

***— логическое значение false.*** Он может быть также переопределен в создаваемых

классах. Это позволяет выяснить, что же означает равенство объектов для создаваемого

класса. Например, метод Equals(object) можно определить таким образом, чтобы

в нем сравнивалось содержимое двух объектов.



Метод GetHashCode() возвращает хеш-код, связанный с вызывающим объектом.

Этот хеш-код можно затем использовать в любом алгоритме, где хеширование применяется

в качестве средства доступа к хранимым объектам. Следует, однако, иметь в

виду, что стандартная реализация метода GetHashCode() не пригодна на все случаи

применения.

Как упоминалось в главе 9, если перегружается оператор ==, то обычно приходится

переопределять методы Equals(object) и GetHashCode(), поскольку чаще всего

требуется, чтобы метод Equals(object) и оператор == функционировали одинаково.

Когда же переопределяется метод Equals(object), то следует переопределить и метод

GetHashCode(), чтобы оба метода оказались совместимыми.

Метод ToString() возвращает символьную строку, содержащую описание того

объекта, для которого он вызывается. Кроме того, метод ToString() автоматически

вызывается при выводе содержимого объекта с помощью метода WriteLine(). Этот

метод переопределяется во многих классах, что позволяет приспосабливать описание

к конкретным типам объектов, создаваемых в этих классах. Ниже приведен пример

применения данного метода.

**(glava11\_14)**

class MyClass

{

static int count = 0;

int id;

public MyClass()

{

id = count;

count++;

}

public override string ToString()

{

return "Object #" + id + " of MyClass type.";

}

}

class Test

{

static void Main()

{

MyClass ob1 = new MyClass();

MyClass ob2 = new MyClass();

MyClass ob3 = new MyClass();

Console.WriteLine(ob1);

Console.WriteLine(ob2);

Console.WriteLine(ob3);

}

}

# Упаковка и распаковка

Как пояснялось выше, все типы в С#, включая и простые типы значений, являются

производными от класса object. Следовательно, ссылкой типа object можно воспользоваться

для обращения к любому другому типу, в том числе и к типам значений.

Когда ссылка на объект класса object используется для обращения к типу значения,

то такой процесс называется *упаковкой.* Упаковка приводит к тому, что значение простого

типа сохраняется в экземпляре объекта, т.е. *"*упаковывается*"* в объекте, который

затем используется как и любой другой объект. Но в любом случае упаковка происходит

автоматически. Для этого достаточно присвоить значение переменной ссылочного

типа object, а об остальном позаботится компилятор С#.

*Распаковка* представляет собой процесс извлечения упакованного значения из

объекта. Это делается с помощью явного приведения типа ссылки на объект класса

object к соответствующему типу значения. Попытка распаковать объект в другой тип

может привести к ошибке во время выполнения.

Ниже приведен простой пример, демонстрирующий упаковку и распаковку.

**(glava11\_15)**

class BoxingDemo

{

static void Main()

{

int x;

object obj;

x = 10;

obj = x; //pack uo value of x int obj

int y = (int)obj; //unpack value from obj to variable type int

Console.WriteLine(y);

}

}

B этом примере кода выводится значение 10. Обратите внимание на то, что значение

переменной х упаковывается в объект простым его присваиванием переменной

obj, ссылающейся на этот объект. А затем это значение извлекается из объекта, доступного

по его ссылке obj, и далее приводится к типу int.

Ниже приведен еще один, более интересный пример упаковки. В данном случае

значение типа int передается в качестве аргумента методу Sqr(), который, в свою

очередь, принимает параметр типа object.

(***glava11\_15.1***)

class BoxingDemo

{

static void Main()

{

int x;

x = 10;

Console.WriteLine("Value of z: " + x);

//value of x automaticly packing up

//when it passes to Sqr() method

x = BoxingDemo.Sqr(x);

Console.WriteLine("Value x \* x = " + x);

}

static int Sqr(object o)

{

return (int)o \* (int)o;

}

}

В данном примере значение переменной х автоматически упаковывается при передаче

методу Sqr()

Упаковка и распаковка позволяют полностью унифицировать систему типов в С#.

Благодаря тому что все типы являются производными от класса object, ссылка на

значение любого типа может быть просто присвоена переменной ссылочного типа

object, а все остальное возьмут на себя упаковка и распаковка. Более того, методы

класса object оказываются доступными всем типам, поскольку они являются производными

от этого класса. В качестве примера рассмотрим довольно любопытную

программу.

// Благодаря упаковке становится возможным вызов методов по значению!

using System;

class MethOnValue

{

static void Main()

{

Console.WriteLine(10.ToString());

}

}

В результате выполнения этой программы выводится значение 10. Дело в том, что

метод ToString() возвращает строковое представление объекта, для которого он вызывается.

В данном случае строковым представлением значения 10 как вызывающего

объекта является само значение 10!

# Класс object как универсальный тип данных

Если object является базовым классом для всех остальных типов и упаковка значений

простых типов происходит автоматически, то класс object можно вполне использовать

в качестве *"*универсального*"* типа данных. Для примера рассмотрим программу,

в которой сначала создается массив типа object, элементам которого затем

присваиваются значения различных типов данных.

(***glava11\_15.2***)

class GenericDemo

{

static void Main()

{

object[] ga = new object[10];

//save integer numbers

for (int i = 0; i < 3; i++)

ga[i] = i;

//save double numms

for (int i = 3; i < 6; i++)

ga[i] = (double)i / 2;

//save two string, and bool values and char

ga[6] = "Hi";

ga[7] = true;

ga[8] = 'X';

ga[9] = "The End";

for (int i = 0; i < 10; i++)

Console.WriteLine(ga[i] + " ");

}

}

Как показывает данный пример, по ссылке на объект класса object можно обращаться

к данным любого типа, поскольку в переменной ссылочного типа object допускается

хранить ссылку на данные всех остальных типов. Следовательно, в массиве

типа object из рассматриваемого здесь примера можно сохранить данные практически

любого типа. В развитие этой идеи можно было бы, например, без особого труда

создать класс стека со ссылками на объекты класса object. Это позволило бы хранить

в стеке данные любого типа.

Несмотря на то что универсальный характер класса object может быть довольно

эффективно использован в некоторых ситуациях, было бы ошибкой думать, что с помощью

этого класса стоит пытаться обойти строго соблюдаемый в C# контроль типов.

Вообще говоря, целое значение следует хранить в переменной типа int, строку — в переменной

ссылочного типа string и т.д.

А самое главное, что начиная с версии 2.0 для программирования на C# стали доступными

подлинно обобщенные типы данных — обобщения (более подробно они

рассматриваются в главе 18). Внедрение обобщений позволило без труда определять

классы и алгоритмы, автоматически обрабатывающие данные разных типов, соблюдая

типовую безопасность. Благодаря обобщениям отпала необходимость пользоваться

классом object как универсальным типом данных при создании нового кода. Универсальный

характер этого класса лучше теперь оставить для применения в особых

случаях.