Наследование является одним из трех основополагающих принципов объектно-ориентированного программирования, поскольку оно допускает создание иерархических классификаций. Благодаря наследованию можно создать общий класс, в котором определяются характерные особенности, присущие множеству связанных элементов. От этого класса могут затем наследовать другие,

более конкретные классы, добавляя в него свои индивидуальные особенности.

В языке C# класс, который наследуется, называется *базовым,* а класс, который наследует, — *производным.* Следовательно, производный класс представляет собой специализированный

вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе, добавляя к ним свои собственные элементы.

# Основы наследования

Поддержка наследования в C# состоит в том, что в объявление одного класса разрешается вводить другой класс. Для этого при объявлении производного класса указывается

базовый класс. Рассмотрим для начала простой пример. Ниже приведен класс TwoDShape, содержащий ширину и высоту двухмерного объекта, например квадрата, прямоугольника,

треугольника и т.д. (***glava11\_1***)

// Класс для двумерных объектов.

class TwoDShape

{

public double Width;

public double Height;

public void ShowDim()

{

Console.WriteLine("Ширина и высота равны " +

Width + " и " + Height);

}

}

Класс TwoDShape может стать базовым, т.е. отправной точкой для создания классов,

описывающих конкретные типы двумерных объектов. Например, в приведенной ниже

программе класс TwoDShape служит для порождения производного класса Triangle.

Обратите особое внимание на объявление класса Triangle.

// Класс Triangle, производный от класса TwoDShape.

class Triangle : TwoDShape

{

public string Style; // тип треугольника

// Возвратить площадь треугольника.

public double Area()

{

return Width \* Height / 2;

}

// Показать тип треугольника.

public void ShowStyle()

{

Console.WriteLine("Треугольник " + Style);

}

}

class Shapes

{

static void Main()

{

Triangle t1 = new Triangle();

Triangle t2 = new Triangle();

t1.Width = 4.0;

t1.Height = 4.0;

t1.Style = "isosceles";

t2.Width = 8.0;

t2.Height = 12.0;

t2.Style = "rectangle";

Console.WriteLine("Info about object t1: ");

t1.ShowStyle();

t1.ShowDim();

Console.WriteLine("Area = " + t1.Area());

Console.WriteLine();

Console.WriteLine("Info about object t2: ");

t2.ShowStyle();

t2.ShowDim();

Console.WriteLine("Area = " + t2.Area());

}

}

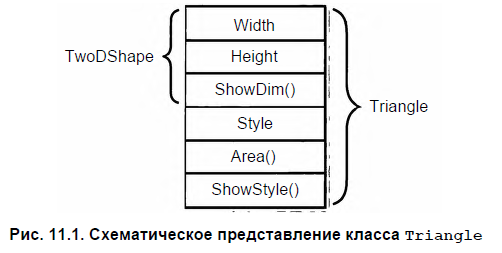
В класс Triangle входят все члены его базового класса TwoDShape, и поэтому

в нем переменные Width и Height доступны для метода Area(). Кроме того, объекты

t1 и t2 в методе Main() могут обращаться непосредственно к переменным Width и

Height, как будто они являются членами класса Triangle. На рис. 11.1 схематически

показано, каким образом класс TwoDShape вводится в класс Triangle.



Если класс служит базовым для производного класса, то это совсем не означает, что он

не может быть использован самостоятельно.

Для любого производного класса можно указать ***только один базовый класс***. В C# не

предусмотрено наследование нескольких базовых классов в одном производном классе.

(В этом отношении C# отличается от C++, где допускается наследование нескольких

базовых классов. Данное обстоятельство следует принимать во внимание при переносе

кода C++ в С#.) Тем не менее можно создать иерархию наследования, в которой производный

класс становится базовым для другого производного класса. (Разумеется, ни

один из классов не может быть базовым для самого себя как непосредственно, так и

косвенно.) Но в любом случае производный класс наследует все члены своего базового

класса, в том числе переменные экземпляра, методы, свойства и индексаторы.

# Доступ к членам класса и наследование

Как пояснялось в главе 8, члены класса зачастую объявляются закрытыми, чтобы

исключить их несанкционированное или незаконное использование. Но наследование

класса не отменяет ограничения, накладываемые на доступ к закрытым членам класса.

Поэтому если в производный класс и входят все члены его базового класса, в нем все

равно оказываются недоступными те члены базового класса, которые являются закрытыми.

Так, если сделать закрытыми переменные класса TwoDShape, они станут недоступными

в классе Triangle, как показано ниже.

//two demensional class of two objects

class TwoDShape

{

double Width; //closed

double Height; //closed

public void ShowDim()

{

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

}

}

//class triangle derived of TwoDShape

class Triangle : TwoDShape

{

public string Style;

//return area of triangle

public double Area()

{

return Width \* Height / 2; //error, cant acces

}

//show type of trgl

public void ShowStyle()

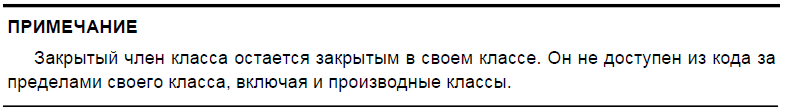
{

Console.WriteLine("Triangle " + Style);

}

}

Поскольку переменные Width и Height теперь являются закрытыми, то они доступны только для других членов своего класса, но не для членов производных классов.



Для преодоления данного ограничения в C# предусмотрены разные способы.

Один из них состоит в использовании защищенных (protected) членов класса, рассматриваемых

в следующем разделе, а второй — в применении открытых ***свойств*** ***для доступа*** к закрытым данным.

Ниже приведен вариант класса TwoDShape, в котором переменные Width и Height превращены в свойства.

(***glava11\_2***)

class TwoDShape

{

double pri\_width;

double pri\_height;

//propertie of hwight and width

public double Width

{

get { return pri\_width; }

set { pri\_width = value < 0 ? -value : value; }

}

public double Height

{

get { return pri\_height; }

set { pri\_height = value < 0 ? -value : value; }

}

public void ShowDim()

{

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

}

}

В этом варианте свойства Width и Height предоставляют доступ к закрытым членам

pri\_width и pri\_height класса TwoDShape, в которых фактически хранятся

значения ширины и высоты двумерного объекта. Следовательно, значения членов

pri\_width и pri\_height класса TwoDShape могут быть установлены и получены с

помощью соответствующих открытых свойств, несмотря на то, что сами эти члены по-

прежнему остаются закрытыми.

Базовый и производный классы иногда еще называют *суперклассом* и *подклассом*

соответственно. Эти термины происходят из практики программирования на Java.

То, что в Java называется суперклассом, в C# обозначается как базовый класс. А то, что

в Java называется подклассом, в C# обозначается как производный класс.

# Организация защищенного доступа

В C# допускается создание *защищенного* члена класса. Защищенный член является

открытым в пределах иерархии классов, но закрытым за пределами этой иерархии.

Защищенный член создается с помощью модификатора доступа protected. Если

член класса объявляется как protected, он становится закрытым, но за исключением

одного случая, когда защищенный член наследуется. В этом случае защищенный член

базового класса становится защищенным членом производного класса, а значит, доступным

для производного класса.

(***glava11\_3***)

class B

{

protected int i, j; //closed for B, oppened for D

public void Set(int a, int b)

{

i = a;

j = b;

}

public void Show()

{

Console.WriteLine(i + " " + j);

}

}

class D : B

{

int k; //closed

//i and j accessable for D

public void Setk()

{

k = i \* j;

}

public void ShowK()

{

Console.WriteLine(k);

}

}

class ProtectedDemo

{

static void Main()

{

D ob = new D();

ob.Set(2,3);

ob.Show();

ob.Setk();

ob.ShowK();

}

}

В данном примере класс В наследуется классом D, а его члены i и j объявлены как

protected, и поэтому они доступны для метода Setk(). Если бы члены i и j класса В

были объявлены как private, то они оказались бы недоступными для класса D, и приведенный

выше код нельзя было бы скомпилировать.

# Конструкторы и наследование

В иерархии классов допускается, чтобы у базовых и производных классов были свои

собственные конструкторы. Конструктор базового класса конструирует базовую часть объекта, а конструктор производного класса — производную часть этого объекта. И в этом есть своя

логика, поскольку базовому классу неизвестны и недоступны любые элементы производного

класса, а значит, их конструирование должно происходить раздельно.

Если конструктор определен только в производном классе, то все происходит очень

просто: конструируется объект производного класса, а базовая часть объекта автоматически

конструируется его конструктором, используемым по умолчанию.

(***glava11\_4***)

class TwoDShape

{

double pri\_width;

double pri\_height;

//properties height and width

public double Width

{

get { return pri\_width; }

set { pri\_width = value < 0 ? -value : value; }

}

public double Height

{

get { return pri\_height; }

set { pri\_height = value < 0 ? -value : value; }

}

public void ShowDim()

{

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

}

}

class Triangle : TwoDShape

{

string Style;

//constructor

public Triangle(string s, double w, double h)

{

Width = w;

Height = h;

Style = s;

}

public double Area()

{

return Width \* Height / 2;

}

public void ShowStyle()

{

Console.WriteLine("Triangle " + Style);

}

}

class Shapes3

{

static void Main()

{

Triangle t1 = new Triangle("isoscales", 4.0, 4.0);

Triangle t2 = new Triangle("rectangle", 8.0, 12.0);

Console.WriteLine("Object info t1: ");

t1.ShowStyle();

t1.ShowDim();

Console.WriteLine("Area: " + t1.Area());

Console.WriteLine();

Console.WriteLine("Object info t2: ");

t2.ShowStyle();

t2.ShowDim();

Console.WriteLine("Area: " + t2.Area());

}

}

В данном примере конструктор класса Triangle инициализирует наследуемые

члены класса TwoDShape вместе с его собственным полем Style.

Когда конструкторы определяются как в базовом, так и в производном классе, процесс

построения объекта несколько усложняется, поскольку должны выполняться

конструкторы обоих классов. В данном случае приходится обращаться к еще одному

ключевому слову языка С#: base, которое находит двоякое применение: во-первых, для

вызова конструктора базового класса; и во-вторых, для доступа к члену базового класса,

скрывающегося за членом производного класса. Ниже будет рассмотрено первое применение

ключевого слова base.

# Вызов конструкторов базового класса base

С помощью формы расширенного объявления конструктора производного класса

и ключевого слова base в производном классе может быть вызван конструктор, определенный

в его базовом классе. Ниже приведена общая форма этого расширенного

объявления:

***конструктор\_производного\_класса(список\_параметров) : base(список\_аргументов) {***

***// тело конструктора***

***}***

где *список\_аргументов* обозначает любые аргументы, необходимые конструктору

в базовом классе. Обратите внимание на местоположение двоеточия.

Для того чтобы продемонстрировать применение ключевого слова base на конкретном

примере, рассмотрим еще один вариант класса TwoDShape в приведенной

ниже программе.

(***glava11\_6***)

class TwoDShape

{

double pri\_width;

double pri\_height;

//constructor

public TwoDShape(double w, double h)

{

Width = w;

Height = h;

}

//properties height and width

public double Width

{

get { return pri\_width; }

set { pri\_width = value < 0 ? -value : value; }

}

public double Height

{

get { return pri\_height; }

set { pri\_height = value < 0 ? -value : value; }

}

public void ShowDim()

{

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

}

}

В базовом классе надо определить конструктор, чтобы в производном можно было обращаться к ***base***.

class Triangle : TwoDShape

{

string Style;

//constructor

public Triangle(string s, double w, double h) :base(w,h)

{

Style = s;

}

public double Area()

{

return Width \* Height / 2;

}

public void ShowStyle()

{

Console.WriteLine("Triangle " + Style);

}

}

Теперь конструктор класса Triangle объявляется следующим образом.

public Triangle(string s, double w, double h) :base(w,h)

С помощью ключевого слова base можно вызвать конструктор любой формы,

определяемой в базовом классе, причем выполняться будет лишь тот конструктор, параметры

которого соответствуют переданным аргументам.

В качестве примера ниже приведены расширенные варианты классов TwoDShape и Triangle, в которые включены как используемые по умолчанию конструкторы, так и конструкторы, принимающие один аргумент.

(***glava11\_5.1***)

class TwoDShape

{

double pri\_width;

double pri\_height;

//default constructor

public TwoDShape()

{

Width = Height = 0;

}

//constructor

public TwoDShape(double w, double h)

{

Width = w;

Height = h;

}

//construct objest equal width - height

public TwoDShape(double x)

{

Width = Height = x;

}

//properties height and width

public double Width

{

get { return pri\_width; }

set { pri\_width = value < 0 ? -value : value; }

}

public double Height

{

get { return pri\_height; }

set { pri\_height = value < 0 ? -value : value; }

}

public void ShowDim()

{

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

}

}

class Triangle : TwoDShape

{

string Style;

//default constructor

public Triangle()

{

Style = "null";

}

//constructor

public Triangle(string s, double w, double h) : base(w, h)

{

Style = s;

}

//construct isoscales triangle

public Triangle(double x) :base(x)

{

Style = "isoscales";

}

public double Area()

{

return Width \* Height / 2;

}

public void ShowStyle()

{

Console.WriteLine("Triangle " + Style);

}

}

class Shapes5

{

static void Main()

{

Triangle t1 = new Triangle();

Triangle t2 = new Triangle("rectangle", 8.0, 12.0);

Triangle t3 = new Triangle(4.0);

t1 = t2;

Console.WriteLine("Object info t1: ");

t1.ShowStyle();

t1.ShowDim();

Console.WriteLine("Area: " + t1.Area());

Console.WriteLine();

Console.WriteLine("Object info t2: ");

t2.ShowStyle();

t2.ShowDim();

Console.WriteLine("Area: " + t2.Area());

Console.WriteLine();

Console.WriteLine("Object info t3: ");

t3.ShowStyle();

t3.ShowDim();

Console.WriteLine("Area: " + t3.Area());

}

}

А теперь рассмотрим вкратце основные принципы действия ключевого слова base.

Когда в производном классе указывается ключевое слово base, вызывается конструктор

из его непосредственного базового класса. Следовательно, ключевое слово base всегда

обращается к базовому классу, стоящему в иерархии непосредственно над вызывающим

классом. Это справедливо даже для многоуровневой иерархии классов. Аргументы

передаются базовому конструктору в качестве аргументов метода base(). Если же

ключевое слово отсутствует, то автоматически вызывается конструктор, используемый

в базовом классе по умолчанию.

# Наследование и сокрытие имен (new var)

В производном классе можно определить член с таким же именем, как и у члена

его базового класса. В этом случае член базового класса скрывается в производном

классе. И хотя формально в C# это не считается ошибкой, компилятор все же выдаст

сообщение, предупреждающее о том, что имя скрывается. Если член базового класса

требуется скрыть намеренно, то перед его именем следует указать ключевое слово

***new***, чтобы избежать появления подобного предупреждающего сообщения. Следует,

однако, иметь в виду, ***что это совершенно отдельное применение ключевого слова new,***

***не похожее на его применение при создании экземпляра объекта***.

(***glava11\_6***)

class A

{

public int i = 0;

}

class B : A

{

new int i; //this i hides member i from A

public B(int b)

{

i = b; //i int B class

}

public void Show()

{

Console.WriteLine("Member i in inherent class: " + i);

}

}

class NameHiding

{

static void Main()

{

B ob = new B(2);

ob.Show();

}

}

Прежде всего обратите внимание на использование ключевого слова new в следующей

строке кода.

new int i; //this i hides member i from A

В этой строке компилятору, по существу, сообщается о том, что вновь создаваемая

переменная i намеренно скрывает переменную i из базового класса А и что автору

программы об этом известно. Если же опустить ключевое слово new в этой строке

кода, то компилятор выдаст предупреждающее сообщение.

В классе В определяется собственная переменная экземпляра i, которая скрывает

переменную i из базового класса А. Поэтому при вызове метода Show() для объекта

типа В выводится значение переменной i, определенной в классе В, а не той, что

определена в классе А.

# Применение ключевого слова base для доступа к скрытому имени, так же доступ к функциям base классу

Имеется еще одна форма ключевого слова base, которая действует подобно ключевому

слову this, за исключением того, что она всегда ссылается на базовый класс

в том производном классе, в котором она используется. Ниже эта форма приведена

в общем виде:

***base.член***

где член может обозначать метод или переменную экземпляра. Эта форма ключевого

слова base чаще всего применяется в тех случаях, когда под именами членов производного

класса скрываются члены базового класса с теми же самыми именами. В качестве

примера ниже приведен другой вариант иерархии классов из предыдущего примера.

(***glava11\_6.1***)

class A

{

public int i = 0;

}

class B : A

{

new int i; //this i hides member i from A

public B(int a, int b)

{

base.i = a; //i from A class

i = b; //i int B class

}

public void Show()

{

//Here also show i from A class

Console.WriteLine("Member i in base class: " + base.i);

Console.WriteLine("Member i in inherent class: " + i);

}

}

class UncoverName

{

static void Main()

{

B ob = new B(1, 2);

ob.Show(); }}

Несмотря на то что переменная экземпляра i в производном классе В скрывает

переменную i из базового класса А, ключевое слово base разрешает доступ к переменной

i, определенной в базовом классе.

С помощью ключевого слова base могут также вызываться скрытые методы.

Например, в приведенном ниже коде класс В наследует класс А и в обоих классах

объявляется метод Show(). А затем в методе Show() класса В с помощью ключевого

слова base вызывается вариант метода Show(), определенный в классе А.

(***glava11\_6.2***)

class A

{

public int i = 0;

public void Show()

{

Console.WriteLine("Member i in base class: " + i);

}

}

class B : A

{

new int i; //this i hides member i from A

public B(int a, int b)

{

base.i = a; //i from A class

i = b; //i int B class

}

new public void Show()

{

base.Show(); //method from A class

Console.WriteLine("Member i in inherent class: " + i);

}

}

class UncoverName

{

static void Main()

{

B ob = new B(1, 2);

ob.Show();

}

}

Как видите, в выражении base.Show() вызывается вариант метода Show() из базового

класса.

***Обратите также внимание на следующее: ключевое слово new используется в приведенном***

***выше коде с целью сообщить компилятору о том, что метод Show(), вновь***

***объявляемый в производном классе В, намеренно скрывает другой метод Show(),***

***определенный в базовом классе А.***

# Создание многоуровневой иерархии классов

В представленных до сих пор примерах программ использовались простые иерархии

классов, состоявшие только из базового и производного классов. Но в C# мож**Глава**

но также строить иерархии, состоящие из любого числа уровней наследования. Как

упоминалось выше, многоуровневая иерархия идеально подходит для использования

одного производного класса в качестве базового для другого производного класса. Так,

если имеются при класса, А, В и С, то класс С может наследовать от класса В, а тот,

в свою очередь, от класса А. В таком случае каждый производный класс наследует характерные

особенности всех своих базовых классов. В частности, класс С наследует все

члены классов В и А.

Для **того** чтобы показать, насколько полезной может оказаться многоуровневая

иерархия классов, рассмотрим следующий пример программы. В ней производный

класс Triangle служит в качестве базового для создания другого производного класса

— ColorTriangle. При этом класс ColorTriangle наследует все характерные особенности,

а по существу, члены классов Triangle и TwoDShape, к которым добавляется

поле color, содержащее цвет треугольника.

(***glava11\_7***)

class TwoDShape

{

double pri\_width;

double pri\_height;

//constructor default

public TwoDShape()

{

pri\_width = pri\_height = 0;

}

//constructor

public TwoDShape(double w, double h)

{

Width = w;

Height = h;

}

//constructor equal angles

public TwoDShape(double x)

{

Width = Height = x;

}

//prop of Height and Width

public double Height

{

get { return pri\_height; }

set { pri\_height = value < 0 ? -value : value; }

}

public double Width

{

get { return pri\_width; }

set { pri\_width = value < 0 ? -value : value; }

}

public void ShowDim()

{

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

}

}

class Triangle : TwoDShape

{

string Style;

public Triangle() { Style = "null"; }

public Triangle(string s, double w, double h) :base(w,h)

{

Style = s;

}

public Triangle(double x) :base(x)

{

Style = "isoscales";

}

public double Area()

{

return Width \* Height / 2;

}

public void ShowStyle()

{

Console.WriteLine("Triangle " + Style);

}

}

//expand Triangle class

class ColorTriangle : Triangle

{

string color;

public ColorTriangle(string c, string s,

double w, double h) :base(s,w,h)

{

color = c;

}

public void ShowColor()

{

Console.WriteLine("Color " + color);

}

}

class Shape6

{

static void Main()

{

ColorTriangle t1 =

new ColorTriangle("blue", "rectangle", 8.0, 12.0);

ColorTriangle t2 =

new ColorTriangle("red", "isoscales", 2.0, 2.0);

Console.WriteLine("Info about object t1: ");

t1.ShowStyle();

t1.ShowDim();

t1.ShowColor();

Console.WriteLine("Area = " + t1.Area());

Console.WriteLine();

Console.WriteLine("Info about object t2: ");

t2.ShowStyle();

t2.ShowDim();

t2.ShowColor();

Console.WriteLine("Area = " + t2.Area());

}

}

Благодаря наследованию в классе ColorTriangle могут использоваться определенные

ранее классы Triangle и TwoDShape, к элементам которых добавляется лишь

та информация, которая требуется для конкретного применения данного класса.

В этом отчасти и состоит ценность наследования, поскольку оно допускает повторное

использование кода.

Приведенный выше пример демонстрирует еще одно важное положение: ключевое

слово base всегда обозначает ссылку на конструктор ближайшего по иерархии

базового класса. Так, ключевое слово base в классе ColorTriangle обозначает вызов

конструктора из класса Triangle, а ключевое слово base в классе Triangle — вызов

конструктора из класса TwoDShape. Если же в иерархии классов конструктору базового

класса требуются параметры, то все производные классы должны предоставлять

эти параметры вверх по иерархии, независимо от того, требуются они самому производному

классу или нет.

# Порядок вызова конструкторов

В связи с изложенными выше в отношении наследования и иерархии классов может

возникнуть следующий резонный вопрос: когда создается объект производного

класса и какой конструктор выполняется первым — тот, что определен в производном

классе, или же тот, что определен в базовом классе? Так, если имеется базовый класс А

и производный класс В, то вызывается ли конструктор класса А раньше конструктора

класса В? Ответ на этот вопрос состоит в том, что в иерархии классов конструкторы вызываются

по порядку выведения классов: ***от базового к производному***. Более того, этот

порядок остается неизменным независимо от использования ключевого слова base.

Так, если ключевое слово base не используется, то выполняется конструктор по умолчанию,

т.е. конструктор без параметров.

(***glava11\_8***)

class A

{

public A()

{

Console.WriteLine("Construct class A.");

}

}

class B : A

{

public B()

{

Console.WriteLine("Construct class B.");

}

}

class C : B

{

public C()

{

Console.WriteLine("Construct class C.");

}

}

class OrderOfConst

{

static void Main()

{

C c = new C();

}

}

# Ссылки на базовый класс и объекты производных классов

Как вам должно быть уже известно, C# является строго типизированным языком

программирования. Помимо стандартных преобразований и автоматического продвижения

простых типов значений, в этом языке строго соблюдается принцип совместимости

типов. Это означает, что переменная ссылки на объект класса одного типа,

как правило, не может ссылаться на объект класса другого типа.

static void Main()

{

X x = new X(10);

X x2;

Y у = new Y(5);

x2 = x; // верно, поскольку оба объекта относятся к одному и тому же типу

х2 = у; // ошибка, поскольку это разнотипные объекты

}

Несмотря на то что классы X и Y в данном примере совершенно одинаковы по своей

структуре, ссылку на объект типа Y нельзя присвоить переменной ссылки на объект

типа X, поскольку типы у них разные.

Вообще говоря, переменная ссылки на объект может ссылаться только на объект

своего типа.

Но из этого принципа строгого соблюдения типов в C# имеется одно важное исключение:

переменной ссылки на объект базового класса может быть присвоена ссылка

на объект любого производного от него класса. Такое присваивание считается вполне

допустимым, поскольку экземпляр объекта производного типа инкапсулирует экземпляр

объекта базового типа. Следовательно, по ссылке на объект базового класса

можно обращаться к объекту производного класса.

(***glava11\_9***)

class X

{

public int a;

public X(int i)

{

a = i;

}

}

class Y : X

{

public int b;

public Y(int i, int j) : base(i)

{

b = j;

}

}

class BaseRef

{

static void Main()

{

X x = new X(10);

X x2;

Y y = new Y(5, 6);

x2 = x; //two object one class

Console.WriteLine("x2.a: " + x2.a);

x2 = y; //OK, Y class is inherent from X

Console.WriteLine("x2.a: " + x2.a);

x2.a = 19;

//x2.b = 27 //wrong, no memb b in X class

}

}

В данном примере класс Y является производным от класса X. Поэтому следующая

операция присваивания:

x2 = y; //OK, Y class is inherent from X

считается вполне допустимой. Ведь по ссылке на объект базового класса (в данном случае

— это переменная х2 ссылки на объект класса X) можно обращаться к объекту производного

класса, т.е. к объекту, на который ссылается переменная у.

Eсли ссылка на объект производного класса присваивается переменной

ссылки на объект базового класса, то доступ разрешается только к тем частям этого

объекта, которые определяются базовым классом. Именно поэтому переменной х2

недоступен член b класса Y, когда она ссылается на объект этого класса. И в этом есть

своя логика, поскольку базовому классу ничего не известно о тех членах, которые добавлены в производный от него класс. Именно поэтому последняя строка кода в приведенномвыше примере была закомментирована.

Один из самых важных моментов для присваивания ссылок на объекты производного

класса переменным базового класса наступает тогда, когда конструкторы вызываются

в иерархии классов. Как вам должно быть уже известно, в классе нередко

определяется конструктор, принимающий объект своего класса в качестве параметра.

Благодаря этому в классе может быть сконструирована копия его объекта. Этой особенностью

можно выгодно воспользоваться в классах, производных от такого класса.

(***glava11\_10***)

class TwoDShape

{

double pri\_width;

double pri\_height;

//options for width and height

public double Width

{

get { return pri\_width; }

set { pri\_width = value < 0 ? -value : value; }

}

public double Height

{

get { return pri\_height; }

set { pri\_height = value < 0 ? -value : value; }

}

//default constructor

public TwoDShape()

{

Width = Height = 0;

}

//constructor

public TwoDShape(double w, double h)

{

Width = w;

Height = h;

}

//constructor equal sides

public TwoDShape(double x)

{

Width = Height = x;

}

public TwoDShape(TwoDShape ob)

{

Width = ob.Width;

Height = ob.Height;

}

public void ShowDim()

{

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

}

}

//Triangle class inherent from TwoDShape

class Triangle : TwoDShape

{

string Style;

//default constructor

public Triangle()

{

Style = "null";

}

//constructor

public Triangle(string s, double w, double h) : base(w, h)

{

Style = s;

}

//constructor equal sidex

public Triangle(double x) : base(x)

{

Style = "isoscale";

}

//copy of Triangle

public Triangle(Triangle ob) : base(ob)

{

Style = ob.Style;

}

//area of Triangle

public double Area()

{

return Width \* Height / 2;

}

//show style

public void ShowStyle()

{

Console.WriteLine("Triangle " + Style);

}

}

class Shapes7

{

static void Main()

{

Triangle t1 = new Triangle("rectangle", 8.0, 12.0);

//make copy of t1

Triangle t2 = new Triangle(t1);

Console.WriteLine("Info object t1: ");

t1.ShowStyle();

t1.ShowDim();

Console.WriteLine("Area is " + t1.Area());

Console.WriteLine();

Console.WriteLine("Info object t2: ");

t2.ShowStyle();

t2.ShowDim();

Console.WriteLine("Area is " + t2.Area());

}

}

Обратите особое внимание на следующий конструктор класса Triangle:

//copy of Triangle

public Triangle(Triangle ob) : base(ob)

{

Style = ob.Style;

}

Он принимает объект типа Triangle в качестве своего параметра и передает его

(с помощью ключевого слова base) следующему конструктору класса TwoDShape.

public TwoDShape(TwoDShape ob)

{

Width = ob.Width;

Height = ob.Height;

}

Самое любопытное, что конструктор TwoDShape() предполагает получить объект

класса TwoDShape, тогда как конструктор Triangle() передает ему объект класса

Triangle. Как пояснялось выше, такое вполне допустимо, поскольку по ссылке на

объект базового класса можно обращаться к объекту производного класса. Следовательно,

конструктору TwoDShape() можно на совершенно законных основаниях передать

ссылку на объект класса, производного от класса TwoDShape. А поскольку конструктор

TwoDShape() инициализирует только те части объекта производного класса,

которые являются членами класса TwoDShape, то для него не имеет никакого значения,

содержит ли этот объект другие члены, добавленные в производном классе.

# Виртуальные методы и их переопределение

*Виртуальным* называется такой метод, который объявляется как ***virtual*** в базовом

классе. Виртуальный метод отличается тем, что он может быть переопределен в одном

или нескольких производных классах. Следовательно, у каждого производного класса

может быть свой вариант виртуального метода. Кроме того, виртуальные методы интересны

тем, что именно происходит при их вызове по ссылке на базовый класс. В этом

случае средствами языка C# определяется именно тот вариант виртуального метода,

который следует вызывать, исходя из *типа* объекта, к которому происходит обращение

*по ссылке,* причем это делается *во время выполнения.*

Bариант выполняемого виртуального метода выбирается по типу объекта, а не по типу

ссылки на этот объект. Так, если базовый класс содержит виртуальный метод и от него

получены производные классы, то при обращении к разным типам объектов по ссылке

на базовый класс выполняются разные варианты этого виртуального метода.

Метод объявляется как виртуальный в ***базовом классе*** с помощью ключевого слова

***virtual***, указываемого перед его именем. Когда же виртуальный метод переопределяется

в ***производном классе***, то для этого используется модификатор ***override***.

А сам процесс повторного определения виртуального метода в производном классе

называется *переопределением метода***. При переопределении имя, возвращаемый тип**

**и сигнатура переопределяющего метода должны быть точно такими же, как и у того**

**виртуального метода, который переопределяется**. Кроме того, виртуальный метод не

может быть объявлен как static или abstract.

(***glava11\_11***)

class Base

{

//virtual method base class

public virtual void Who()

{

Console.WriteLine("Method Who() base class");

}

}

class Derived1 : Base

{

public override void Who()

{

Console.WriteLine("Method Who() derived class.");

}

}

class Derived2:Base

{

public override void Who()

{

Console.WriteLine("Method Who() derived2 class");

}

}

class OverrideDemo

{

static void Main()

{

Base baseOb = new Base();

Derived1 dOb1 = new Derived1();

Derived2 dOb2 = new Derived2();

Base baseRef; //link on base class

baseRef = baseOb;

baseRef.Who();

baseRef = dOb1;

baseRef.Who();

baseRef = dOb2;

baseRef.Who();

}

}

В коде из приведенного выше примера создаются базовый класс Base и два производных

от него класса — Derived1 и Derived2. В классе Base объявляется виртуальный

метод Who(), который переопределяется в обоих производных классах. Затем в

методе Main() объявляются объекты типа Base, Derived1 и Derived2. Кроме того,

объявляется переменная baseRef ссылочного типа Base. Далее ссылка на каждый тип

объекта присваивается переменной baseRef и затем используется для вызова метода

Who().

Но переопределять виртуальный метод совсем не обязательно. ***Ведь если в производном***

***классе не предоставляется собственный вариант виртуального метода, то используется***

***его вариант из базового класса***, как в приведенном ниже примере.

(***glava11\_11.1***)

using System;

class Base

{

//virtual method base class

public virtual void Who()

{

Console.WriteLine("Method Who() base class");

}

}

class Derived1 : Base

{

public override void Who()

{

Console.WriteLine("Method Who() derived class.");

}

}

class Derived2 : Base

{

}

class OverrideDemo

{

static void Main()

{

Base baseOb = new Base();

Derived1 dOb1 = new Derived1();

Derived2 dOb2 = new Derived2();

Base baseRef; //link on base class

baseRef = baseOb;

baseRef.Who();

baseRef = dOb1;

baseRef.Who();

baseRef = dOb2;

baseRef.Who();

}

}

В данном примере метод Who() не переопределяется в классе Derived2. Поэтому

для объекта класса Derived2 вызывается метод Who() из класса Base.

Если при наличии многоуровневой иерархии виртуальный метод не переопределяется

в производном классе, то выполняется ближайший его вариант, обнаруживаемый

вверх по иерархии, как в приведенном ниже примере.

(***glava11\_11.2***)

class Base

{

//virtual method base class

public virtual void Who()

{

Console.WriteLine("Method Who() base class");

}

}

class Derived1 : Base

{

public override void Who()

{

Console.WriteLine("Method Who() derived1 class.");

}

}

class Derived2 : Derived1

{ }

class Derived3 : Derived2

{ }

class OverrideDemo

{

static void Main()

{

Derived3 dOb = new Derived3();

Base baseRef; //link on base class

baseRef = dOb;

baseRef.Who();

}

}

В данном примере класс Derived3 наследует класс Derived2, который наследует

класс Derived1, а тот, в свою очередь, — класс Base. Как показывает приведенный

выше результат, выполняется метод Who(), переопределяемый в классе Derived1,

поскольку это первый вариант виртуального метода, обнаруживаемый при продвижении

вверх по иерархии от классов Derived3 и Derived2, где метод Who() не переопределяется,

к классу Derived1.

И еще одно замечание: свойства также подлежат модификации ключевым словом

virtual и переопределению ключевым словом override. Это же относится и к индексаторам.

# Что дает переопределение методов

Благодаря переопределению методов в C# поддерживается динамический полиморфизм.

В объектно-ориентированном программировании полиморфизм играет

очень важную роль, потому что он позволяет определить в общем классе методы,

которые становятся общими для всех производных от него классов, а в производных

классах — определить конкретную реализацию некоторых или же всех этих методов.

Переопределение методов — это еще один способ воплотить в C# главный принцип

полиморфизма: один интерфейс — множество методов.

Удачное применение полиморфизма отчасти зависит от правильного понимания

той особенности, что базовые и производные классы образуют иерархию, которая продвигается

от меньшей к большей специализации. При надлежащем применении базовый

класс предоставляет все необходимые элементы, которые могут использоваться

в производном классе непосредственно. А с помощью виртуальных методов в базовом

классе определяются те методы, которые могут быть самостоятельно реализованы в

производном классе. Таким образом, сочетая наследование с виртуальными методами,

можно определить в базовом классе общую форму методов, которые будут использоваться

во всех его производных классах.

# Применение виртуальных методов

Для того чтобы стали понятнее преимущества виртуальных методов, применим их

в классе TwoDShape. В предыдущих примерах в каждом классе, производном от класса

TwoDShape, определялся метод Area(). Но, по-видимому, метод Area() лучше было

бы сделать виртуальным в классе TwoDShape и тем самым предоставить возможность

переопределить его в каждом производном классе с учетом особенностей расчета площади

той двумерной формы, которую инкапсулирует этот класс. Именно это и сделано

в приведенном ниже примере программы. Ради удобства демонстрации классов в

этой программе введено также свойство name в классе TwoDShape.

(***glava11\_12***)

class TwoDShape

{

double pri\_width;

double pri\_height;

public string name { get; set; }

//properties

public double Width

{

get { return pri\_width; }

set { pri\_width = value < 0 ? -value : value; }

}

public double Height

{

get { return pri\_height; }

set { pri\_height = value < 0 ? -value : value; }

}

//default constructor

public TwoDShape()

{

Width = Height = 0.0;

name = "null";

}

//constructor

public TwoDShape(string n, double w, double h)

{

name = n;

Width = w;

Height = h;

}

//equal sides

public TwoDShape(double x, string n)

{

Width = Height = x;

name = n;

}

public TwoDShape(TwoDShape ob)

{

Width = ob.Width;

Height = ob.Height;

name = ob.name;

}

public void ShowDim()

{

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

}

public virtual double Area()

{

Console.WriteLine("Method are has to be redefined");

return 0.0;

}

}

//triangle class

class Triangle : TwoDShape

{

string Style;

//default constructor

public Triangle() { Style = "null"; }

//constructor

public Triangle(string s, double w, double h) : base("triangle", w, h)

{

Style = s;

}

//equal sides

public Triangle(double x) : base(x, "Triangle")

{

Style = "isoscale";

}

public Triangle(Triangle ob) : base(ob)

{

Style = ob.Style;

}

public override double Area()

{

return Width \* Height / 2;

}

public void ShowStyle()

{

Console.WriteLine("Triangle " + Style);

}

}

class Rectangle : TwoDShape

{

//constructor

public Rectangle(double w, double h) : base("rectangle", w, h) { }

//construct square

public Rectangle(double x) : base(x, "Square") { }

//copy of rectangle

public Rectangle(Rectangle ob) : base(ob) { }

//return if square

public bool IsSquare()

{

if (Height == Width) return true;

return false;

}

public override double Area()

{

return Width \* Height;

}

}

class DynShapes

{

static void Main()

{

TwoDShape[] shapes = new TwoDShape[5];

shapes[0] = new Triangle("rectangle", 8.0, 12.0);

shapes[1] = new Rectangle(10);

shapes[2] = new Rectangle(10, 4);

shapes[3] = new Triangle(7.0);

shapes[4] = new TwoDShape("common form", 10, 20);

for (int i = 0; i < shapes.Length; i++)

{

Console.WriteLine("Object - " + shapes[i].name);

Console.WriteLine("Area = " + shapes[i].Area());

Console.WriteLine();

}

}

}

Рассмотрим данный пример программы более подробно. Прежде всего, метод

Area() объявляется как virtual в классе TwoDShape и переопределяется в классах

Triangle и Rectangle по объяснявшимся ранее причинам. В классе TwoDShape метод

Area() реализован в виде заполнителя, который сообщает о том, что пользователь

данного метода должен переопределить его в производном классе. Каждое переопределение

метода Area() предоставляет конкретную его реализацию, соответствующую

типу объекта, инкапсулируемого в производном классе. Так, если реализовать класс

для эллипсов, то метод Area() должен вычислять площадь эллипса.

У программы из рассматриваемого здесь примера имеется еще одна примечательная

особенность. Обратите внимание на то, что в методе Main() двумерные формы

объявляются в виде массива объектов типа TwoDShape, но элементам этого массива

присваиваются ссылки на объекты классов Triangle, Rectangle и TwoDShape. И это

вполне допустимо, поскольку по ссылке на базовый класс можно обращаться к объекту

производного класса.