**Общая форма обобщенного класса**

Синтаксис обобщений, представленных в предыдущих примерах, может быть сведен

к общей форме. Ниже приведена общая форма объявления обобщенного класса.

class имя\_класса<список\_параметров\_типа> { // ...

А вот как выглядит синтаксис объявления ссылки на обобщенный класс.

имя\_класса<список\_аргументов\_типа> имя\_переменной -

new имя\_класса<список\_параметров\_типа>(список\_аргументов\_конструктора);

Начнем рассмотрение обобщений с простого примера обобщенного класса. В приведенной

ниже программе определяются два класса. Первым из них является обобщенный

класс Gen, вторым — класс GenericsDemo, в котором используется класс Gen.

(***glava18\_1***)

//in class below Get parameter of type T change

//real type of data when its create object of type Gen

class Gen<T>

{

T ob; //variable of type T

//look closer, this constructor has parameter type T

public Gen(T o)

{

ob = o;

}

//return variable ob, it has type T

public T GetOb()

{

return ob;

}

//show type T

public void ShowType()

{

Console.WriteLine("Type of T: " + typeof(T));

}

}

//demonstrate

class GenericDemo

{

static void Main()

{

//create variable link to Gen object type int

Gen<int> iOb;

//create object type Gen<int> and assign it to iOb

iOb = new Gen<int>(102);

//show type of data int iOb

iOb.ShowType();

//get value of iOb

int v = iOb.GetOb();

Console.WriteLine("Value: " + v);

Console.WriteLine();

//create object type Gen for strings

Gen<string> strOb = new Gen<string>("Generalization increases effectivity.");

//show data type int strOb

strOb.ShowType();

//get value strOb

string str = strOb.GetOb();

Console.WriteLine("Value: " + str);

}

}

Внимательно проанализируем эту программу. Прежде всего обратите внимание на

объявление класса Gen в приведенной ниже строке кода:

class Gen<T>

где Т — это имя *параметра типа.* Это имя служит в качестве метки-заполнителя конкретного

типа, который указывается при создании объекта класса Gen. Следовательно,

имя Т используется в классе Gen всякий раз, когда требуется параметр типа. Обратите

внимание на то, что имя Т заключается в угловые скобки (< >). Этот синтаксис можно

обобщить: всякий раз, когда объявляется параметр типа, он указывается в угловых

скобках. А поскольку параметр типа используется в классе Gen, то такой класс считается

*обобщенным.*

В объявлении класса Gen можно указывать любое имя параметра типа, но по традиции

выбирается имя Т. К числу других наиболее употребительных имен параметров

типа относятся V и Е. Вы, конечно, вольны использовать и более описательные имена,

например TValue или ТКеу. Но в этом случае первой в имени параметра типа принято

указывать прописную букву Т.

Далее имя Т используется для объявления переменной ob, как показано в следующей

строке кода.

T ob; //variable of type T

Как пояснялось выше, имя параметра типа Т служит меткой-заполнителем конкретного

типа, указываемого при создании объекта класса Gen. Поэтому переменная

ob будет иметь тип, *привязываемый* к Т при получении экземпляра объекта класса Gen.

Так, если вместо Т указывается тип string, то в экземпляре данного объекта переменная

оb будет иметь тип string.

А теперь рассмотрим конструктор класса Gen.

//look closer, this constructor has parameter type T

public Gen(T o)

{

ob = o;

}

Как видите, параметр о этого конструктора относится к типу Т. Это означает, что

конкретный тип параметра о определяется типом, привязываемым к Т при создании

объекта класса Gen. А поскольку параметр о и переменная экземпляра ob относятся

к типу Т, то после создания объекта класса Gen их конкретный тип окажется одним и

тем же.

С помощью параметра типа Т можно также указывать тип, возвращаемый методом,

как показано ниже на примере метода GetOb().

//return variable ob, it has type T

public T GetOb()

{

return ob;

}

Переменная ob также относится к типу Т, поэтому ее тип совпадает с типом, возвращаемым

методом GetOb().

В классе GenericsDemo демонстрируется применение обобщенного класса Gen.

Сначала в нем создается вариант класса Gen для типа int.

//create variable link to Gen object type int

Gen<int> iOb;

Внимательно проанализируем это объявление. Прежде всего обратите внимание на

то, что тип int указывается в угловых скобках после имени класса Gen. В этом случае

int служит *аргументом типа,* привязанным к параметру типа Т в классе Gen. В данном

объявлении создается вариант класса Gen, в котором тип Т заменяется типом int везде,

где он встречается. Следовательно, после этого объявления int становится типом

переменной ob и возвращаемым типом метода GetOb().

В следующей строке кода переменной iOb присваивается ссылка на экземпляр

объекта класса Gen для варианта типа int.

//create object type Gen<int> and assign it to iOb

iOb = new Gen<int>(102);

Обратите внимание на то, что при вызове конструктора класса Gen указывается также

аргумент типа int. Это необходимо потому, что переменная (в данном случае —

iOb), которой присваивается ссылка, относится к типу Gen<int>. Поэтому ссылка, возвращаемая

оператором new, также должна относиться к типу Gen<int>. В противном

случае во время компиляции возникнет ошибка.

Прежде чем продолжить изложение, следует дать определение некоторым терминам.

Когда для класса Gen указывается аргумент типа, например int или string, то

создается так называемый в C# *закрыто сконструированный тип.* В частности, Gen<int>

является закрыто сконструированным типом. Ведь, по существу, такой обобщенный

тип, как Gen<T>, является абстракцией. И только после того, как будет сконструирован

конкретный вариант, например Gen<int>, создается конкретный тип. А конструкция, подобная Gen<T>, называется в C# *открыто сконструированным типом,* поскольку

в ней указывается параметр типа Т, но не такой конкретный тип, как int.

**Различение обобщенных типов по аргументам типа**

Что касается обобщенных типов, то следует иметь в виду, что ссылка на один конкретный

вариант обобщенного типа не совпадает по типу с другим вариантом того же

самого обобщенного типа. Так, если ввести в приведенную выше программу следующую

строку кода, то она не будет скомпилирована.

iOb = strOb; // Неверно!

Несмотря на то что обе переменные, iOb и strOb, относятся к типу Gen<T>, они

ссылаются на разные типы, поскольку у них разные аргументы.

**Повышение типовой безопасности с помощью обобщений**

В связи с изложенным выше возникает следующий резонный вопрос: если аналогичные

функциональные возможности обобщенного класса Gen можно получить и без

обобщений, просто указав объект как тип данных и выполнив надлежащее приведение

типов, то какая польза от того, что класс Gen делается обобщенным? Ответ на этот вопрос

заключается в том, что обобщения автоматически обеспечивают типовую безопасность

всех операций, затрагивающих класс Gen. В ходе выполнения этих операций

обобщения исключают необходимость обращаться к приведению типов и проверять

соответствие типов в коде вручную.

Для того чтобы стали более понятными преимущества обобщений, рассмотрим

сначала программу, в которой создается необобщенный аналог класса Gen.

(***glava18\_2***)

class NonGen

{

object ob; //variable now type object

//pass to constructor link to object type

public NonGen(object o)

{

ob = o;

}

//return link of object type

public object GetOb()

{

return ob;

}

//show type of ob

public void ShowType()

{

Console.WriteLine("Type of variable ob: " + ob.GetType());

}

}

//show not generalitic class

class NonGenDemo

{

static void Main()

{

NonGen iOb;

//create class object NonGen

iOb = new NonGen(102);

//show type of iOb

iOb.ShowType();

//get value of iOb

//now need straight type

int v = (int)iOb.GetOb();

Console.WriteLine("Value: " + v);

Console.WriteLine();

//create one more object NonGen

//and save string in it

NonGen strOb = new NonGen("Generalization test.");

//show type of data int strOb

strOb.ShowType();

//get value of strOb

//and here need straight type

String str = (string)strOb.GetOb();

Console.WriteLine("Value: " + str);

//this code compiles, but it is not correct!

iOb = strOb;

//next code will execute Error

//v = (int)iOb.GetOb();

}

}

Как видите, результат выполнения этой программы такой же, как и у предыдущей

программы.

В этой программе обращает на себя внимание ряд любопытных моментов. Прежде

всего, тип Т заменен везде, где он встречается в классе NonGen. Благодаря этому в классе

NonGen может храниться объект любого типа, как и в обобщенном варианте этого

класса. Но такой подход оказывается непригодным по двум причинам. Во-первых, для

извлечения хранящихся данных требуется явное приведение типов. И во-вторых, многие

ошибки несоответствия типов не могут быть обнаружены вплоть до момента выполнения

программы. Рассмотрим каждую из этих причин более подробно.

Начнем со следующей строки кода.

int v = (int)iOb.GetOb();

Теперь возвращаемым типом метода GetOb() является object, а следовательно,

для распаковки значения, возвращаемого методом GetOb(), и его последующего сохранения

в переменной v требуется явное приведение к типу int. Если исключить

приведение типов, программа не будет скомпилирована. В обобщенной версии этой

программы приведение типов не требовалось, поскольку тип int указывался в качестве

аргумента типа при создании объекта iOb. А в необобщенной версии этой программы

потребовалось явное приведение типов. Но это не только неудобно, но и чревато

ошибками.

А теперь рассмотрим следующую последовательность кода в конце анализируемой

здесь программы

//this code compiles, but it is not correct!

iOb = strOb;

//next code will execute Error

//v = (int)iOb.GetOb();

В этом коде значение переменной strOb присваивается переменной iOb. Но переменная

strOb ссылается на объект, содержащий символьную строку, а не целое значение.

Такое присваивание оказывается верным с точки зрения синтаксиса, поскольку

все ссылки на объекты класса NonGen одинаковы, а значит, по ссылке на один объект

класса NonGen можно обращаться к любому другому объекту класса NonGen. Тем не

менее такое присваивание неверно с точки зрения семантики, как показывает следующая

далее закомментированная строка кода. В этой строке тип, возвращаемый методом

GetOb(), приводится к типу int, а затем предпринимается попытка присвоить

полученное в итоге значение переменной int. К сожалению, в отсутствие обобщений

компилятор не сможет выявить подобную ошибку. Вместо этого возникнет исключительная

ситуация во время выполнения, когда будет предпринята попытка приведения

к типу int.

Упомянутая выше ситуация не могла бы возникнуть, если бы в программе использовались

обобщения. Компилятор выявил бы ошибку в приведенной выше последовательности

кода, если бы она была включена в обобщенную версию программы, и сообщил

бы об этой ошибке, предотвратив тем самым серьезный сбой, приводящий

к исключительной ситуации при выполнении программы. Возможность создавать

типизированный код, в котором ошибки несоответствия типов выявляются во время

компиляции, является главным преимуществом обобщений.

По существу, обобщения переводят ошибки при выполнении в разряд ошибок при компиляции.

В этом и заключается основная польза от обобщений.

В рассматриваемой здесь необобщенной версии программы имеется еще один

любопытный момент. Обратите внимание на то, как тип переменной ob экземпляра

класса NonGen создается с помощью метода ShowType() в следующей строке кода.

Console.WriteLine("Type of variable ob: " + ob.GetType());

Как пояснялось в главе 11, в классе object определен ряд методов, доступных для

всех типов данных. Одним из них является метод GetType(), возвращающий объект

класса Туре, который описывает тип вызывающего объекта во время выполнения. Следовательно,

конкретный тип объекта, на который ссылается переменная ob, становится

известным во время выполнения, несмотря на то, что тип переменной ob указан

в исходном коде как object. Именно поэтому в среде CLR будет сгенерировано исключение

при попытке выполнить неверное приведение типов во время выполнения

программы.

**Обобщенный класс с двумя параметрами типа**

В классе обобщенного типа можно указать два или более параметра типа. В этом

случае параметры типа указываются списком через запятую. В качестве примера ниже

приведен класс TwoGen, являющийся вариантом класса Gen с двумя параметрами

типа.

(***glava18\_3***)

class TwoGen<T, V>

{

T ob1;

V ob2;

//in constructor has two params T and V

public TwoGen(T o1, V o2)

{

ob1 = o1;

ob2 = o2;

}

//showtype T and V

public void ShowTypes()

{

Console.WriteLine("Type T has type: " + typeof(T));

Console.WriteLine("Type V has type: " + typeof(V));

}

public T getob1()

{

return ob1;

}

public V getob2()

{

return ob2;

}

}

//demonstrate with two params

class SimpGen

{

static void Main()

{

TwoGen<int, string> tgObj =

new TwoGen<int, string>(119, "Alpha Beta Gamma");

//show types

tgObj.ShowTypes();

//get and show values

int v = tgObj.getob1();

string str = tgObj.getob2();

Console.WriteLine("Value: " + v);

Console.WriteLine("Value: " + str);

}

}

Обратите внимание на то, как объявляется класс TwoGen.

class TwoGen<T, V>

В этом объявлении указываются два параметра типа Т и V, разделенные запятой.

А поскольку у класса TwoGen два параметра типа, то при создании объекта этого класса

необходимо указывать два соответствующих аргумента типа, как показано ниже.

TwoGen<int, string> tgObj =

new TwoGen<int, string>(119, "Alpha Beta Gamma");

В данном случае вместо Т подставляется тип int, а вместо V — тип string.

В представленном выше примере указываются аргументы разного типа, но они могут

быть и одного типа. Например, следующая строка кода считается вполне допустимой.

TwoGen<string, string> х =

new TwoGen<string, string>("Hello", "Goodbye");

В этом случае оба типа, Т и V, заменяются одним и тем же типом, string. Ясно,

что если бы аргументы были одного и того же типа, то два параметра типа были бы

не нужны.

**Получение значения, присваиваемого параметру типа по умолчанию default**

Как упоминалось выше, при написании обобщенного кода иногда важно провести

различие между типами значений и ссылочными типами. Такая потребность возникает,

в частности, в том случае, если переменной параметра типа должно быть присвоено

значение по умолчанию. Для ссылочных типов значением по умолчанию является

null, для неструктурных типов значений — 0 или логическое значение false, если это

тип bool, а для структур типа struct — объект соответствующей структуры с полями,

установленными по умолчанию. В этой связи возникает вопрос: какое значение следует

присваивать по умолчанию переменной параметра типа: null, 0 или нечто другое?

class Test<T>

{

Т obj;

// ...

переменной obj требуется присвоить значение по умолчанию, то какой из двух вариантов

obj = null; // подходит только для ссылочных типов

или

obj = 0; // подходит только для числовых типов и

// перечислений, но не для структур

следует выбрать? Для разрешения этой дилеммы можно воспользоваться еще одной

формой оператора default, приведенной ниже.

default(тип)

Эта форма оператора default пригодна для всех аргументов типа, будь то типы

значений или ссылочные типы.

Ниже приведен короткий пример, демонстрирующий данную форму оператора

default.

(***glava18\_11***)

class MyClass { }

//get value assign to paramter of T type default

class Test<T>

{

public T obj;

public Test()

{

//next operator is only for link type

//obj = null;

//next type is only for value type

//obj = 0;

//this operator for BOTH types

obj = default(T); //!!!

}

}

class DefaultDemo

{

static void Main()

{

//make object of Test class, use link type

Test<MyClass> x = new Test<MyClass>();

if (x.obj == null)

Console.WriteLine("Variable x.obj has <null> value");

//make object of Test class, use value type

Test<int> y = new Test<int>();

if (y.obj == 0)

Console.WriteLine("Variable x.obj has <0> value");

}

}

**Обобщенные структуры**

В C# разрешается создавать обобщенные структуры. Синтаксис для них такой же,

как и для обобщенных классов. В качестве примера ниже приведена программа, в которой

создается обобщенная структура XY для хранения координат X, Y.

(***glava18\_12***)

//generalizated structure

struct XY<T>

{

T x;

T y;

public XY(T a, T b)

{

x = a;

y = b;

}

public T X

{

get { return x; }

set { x = value; }

}

public T Y

{

get { return y; }

set { y = value; }

}

}

class StructTest

{

static void Main()

{

XY<int> xy = new XY<int>(10, 20);

XY<double> xy2 = new XY<double>(88.0, 99.0);

Console.WriteLine(xy.X + ", " + xy.Y);

Console.WriteLine(xy2.X + ", " + xy2.Y);

}

}

Как и на обобщенные классы, на обобщенные структуры могут накладываться ограничения.

Например, на аргументы типа в приведенном ниже варианте структуры XY

накладывается ограничение типа значения.

struct XY<T> where T : struct

**Создание обобщенного метода**

Как следует из приведенных выше примеров, в методах, объявляемых в обобщенных

классах, может использоваться параметр типа из данного класса, а следовательно,

такие методы автоматически становятся обобщенными по отношению к параметру

типа. Но помимо этого имеется возможность объявить обобщенный метод со своими

собственными параметрами типа и даже создать обобщенный метод, заключенный в

необобщенном классе.

Рассмотрим для начала простой пример. В приведенной ниже программе объявляется

необобщенный класс ArrayUtils, а в нем — статический обобщенный метод

CopyInsert(). Этот метод копирует содержимое одного массива в другой, вводя по

ходу дела новый элемент в указанном месте. Метод CopyInsert() можно использовать

вместе с массивами любого типа.

(***glava18\_13***)

//this class is not Generalized

class ArrayUtils

{

//copy array, with new element

//this method is Generalized

public static bool CopyInsert<T>(T e, uint idx,

T[] src, T[] target)

{

//check how big array is

if (target.Length < src.Length)

return false;

//copy array src into indicated array,

//place value e by index idx

for (int i = 0, j = 0; i < src.Length; i++, j++)

{

if (i == idx)

{

target[j] = e;

j++;

}

target[j] = src[i];

}

return true;

}

}

class GenMethDemo

{

static void Main()

{

int[] nums = { 1, 2, 3 };

int[] nums2 = new int[4];

//output values nums array

Console.Write("Array nums: ");

foreach (int x in nums)

Console.Write(x + " ");

Console.WriteLine();

//proccess aray int type

ArrayUtils.CopyInsert(99, 2, nums, nums2);

//output values nums2 array

Console.Write("Array nums2: ");

foreach (int x in nums2)

Console.Write(x + " ");

Console.WriteLine();

//now string array

string[] strs = { "Gens", "are", "effective" };

string[] strs2 = new string[4];

//output values strs array

Console.Write("Array strs: ");

foreach (string x in strs)

Console.Write(x + " ");

Console.WriteLine();

//get value into string rray

ArrayUtils.CopyInsert("in C#", 1, strs, strs2);

//output values strs2 array

Console.Write("Array strs2: ");

foreach (string x in strs2)

Console.Write(x + " ");

Console.WriteLine();

}

}

Внимательно проанализируем метод CopyInsert(). Прежде всего обратите внимание

на объявление этого метода в следующей строке кода.

public static bool CopyInsert<T>(T e, uint idx,

T[] src, T[] target)

Параметр типа объявляется *после* имени метода, но *перед* списком его параметров.

Обратите также внимание на то, что метод CopyInsert() является статическим, что

позволяет вызывать его независимо от любого объекта. Следует, однако, иметь в виду,

что обобщенные методы могут быть либо статическими, либо нестатическими. В этом

отношении для их не существует никаких ограничений.

Далее обратите внимание на то, что метод CopyInsert() вызывается в методе

Main() с помощью обычного синтаксиса и без указания аргументов типа. Дело в том,

что типы аргументов различаются автоматически, а тип Т соответственно подстраивается.

Этот процесс называется *выводимостью типов.* Например, в первом вызове данного

Метода

ArrayUtils.CopyInsert(99, 2, nums, nums2);

тип T становится типом int, поскольку числовое значение 99 и элементы массивов

nums и nums2 относятся к типу int. А во втором вызове данного метода используются

строковые типы, и поэтому тип Т заменяется типом string.

**Вызов обобщенного метода с явно указанными аргументами типа**

В большинстве случаев неявной выводимости типов оказывается достаточно для вызова

обобщенного метода, тем не менее аргументы типа могут быть указаны явным образом.

Для этого достаточно указать аргументы типа после имени метода при его вызове.

В качестве примера ниже приведена строка кода, в которой метод CopyInsert()

вызывается с явно указываемым аргументом типа string.

ArrayUtils.CopyInsert<string>("В С#", 1, strs, strs2);

Тип передаваемых аргументов необходимо указывать явно в том случае, если компилятор

не сможет вывести тип параметра Т или если требуется отменить выводимость

типов.

**Применение ограничений в обобщенных методах**

На аргументы обобщенного метода можно наложить ограничения, указав их после

списка параметров. В качестве примера ниже приведен вариант метода CopyInsert()

для обработки данных только ссылочных типов.

public static bool CopyInsert<T>(T e, uint idx,

T[] src, T[] target) where T : class

Если попробовать применить этот вариант в предыдущем примере программы обработки

массивов, то приведенный ниже вызов метода CopyInsert() не будет скомпилирован,

поскольку int является типом значения, а не ссылочным типом.

**Иерархии обобщенных классов**

Обобщенные классы могут входить в иерархию классов аналогично необобщенным

классам. Следовательно, обобщенный класс может действовать как базовый или производный

класс. Главное отличие между иерархиями обобщенных и необобщенных

классов заключается в том, что в первом случае аргументы типа, необходимые обобщенному

базовому классу, должны передаваться всеми производными классами вверх

по иерархии аналогично передаче аргументов конструктора.

Ниже приведен простой пример иерархии, в которой используется обобщенный

базовый класс.

(***glava18\_17***)

//generalized base class

class Gen<T>

{

T ob;

public Gen(T o)

{

ob = o;

}

//return value ob

public T GetOb()

{

return ob;

}

}

//derive class from Gen

class Gen2<T> : Gen<T>

{

public Gen2(T o) : base(o) { }

}

class GenHierDemo

{

static void Main()

{

Gen2<string> g2 = new Gen2<string>("Hello");

Console.WriteLine(g2.GetOb());

}

}

В этой иерархии класс Gen2 наследует от обобщенного класса Gen. Обратите внимание

на объявление класса Gen2 в следующей строке кода.

class Gen2<T> : Gen<T>

Параметр типа Т указывается в объявлении класса Gen2 и в то же время передается

классу Gen. Это означает, что любой тип, передаваемый классу Gen2, будет передаваться

также классу Gen. Например, в следующем объявлении:

Gen2<string> g2 = new Gen2<string>("Hello");

параметр типа string передается классу Gen. Поэтому переменная ob в той части

класса Gen2, которая относится к классу Gen, будет иметь тип string.

Обратите также внимание на то, что в классе Gen2 параметр типа Т не используется,

а только передается вверх по иерархии базовому классу Gen. Это означает, что

в производном классе следует непременно указывать параметры типа, требующиеся

его обобщенному базовому классу, даже если этот производный класс не обязательно

должен быть обобщенным.

Разумеется, в производный класс можно свободно добавлять его собственные параметры

типа, если в этом есть потребность. В качестве примера ниже приведен вариант

предыдущей иерархии классов, где в класс Gen2 добавлен собственный параметр

типа.

(***glava18\_18***)

//Generalizated base class

class Gen<T>

{

T ob;

//pass to constructor type T link

public Gen(T o)

{

ob = o;

}

//return value of ob variable

public T GetOb()

{

return ob;

}

}

//class derived from Gen. This class

//has second parameter type V

class Gen2<T, V> : Gen<T>

{

V ob2;

public Gen2(T o, V o2) : base(o)

{

ob2 = o2;

}

public V GetObj2()

{

return ob2;

}

}

//create object class Gen2

class GenHierDemo2

{

static void Main()

{

//create object class Gen2 with parameters

//type string and int

Gen2<string, int> x =

new Gen2<string, int>("Vale :", 99);

Console.Write(x.GetOb());

Console.WriteLine(x.GetObj2());

}

}

Обратите внимание на приведенное ниже объявление класса Gen2 в данном варианте

иерархии классов.

class Gen2<T, V> : Gen<T>

В этом объявлении Т — это тип, передаваемый базовому классу Gen; а V — тип, характерный

только для производного класса Gen2. Он служит для объявления объекта

оb2 и в качестве типа, возвращаемого методом GetObj2(). В методе Main() создается

объект класса Gen2 с параметром Т типа string и параметром V типа int. Поэтому

код из приведенного выше примера дает следующий результат.

Значение равно: 99

**Обобщенный производный класс**

Необобщенный класс может быть вполне законно базовым для обобщенного производного

класса. В качестве примера рассмотрим следующую программу.

(***glava18\_19***)

//not generalized base class

class NonGen

{

int num;

public NonGen(int i)

{

num = i;

}

public int GetNum()

{

return num;

}

}

//generalized derive class

class Gen<T> : NonGen

{

T ob;

public Gen(T o, int i) :base(i)

{

ob = o;

}

//return value of ob

public T GetOb()

{

return ob;

}

}

//create object of Gen

class HierDemo3

{

static void Main()

{

//create Gen object with parameter type string

Gen<string> w = new Gen<string>("Hello", 47);

Console.Write(w.GetOb() + " ");

Console.WriteLine(w.GetNum());

}

}

В данной программе обратите внимание на то, как класс Gen наследует от класса

NonGen в следующем объявлении.

class Gen<T> : NonGen

Класс NonGen не является обобщенным, и поэтому аргумент типа для него не указывается.

Это означает, что параметр т, указываемый в объявлении обобщенного производного

класса Gen, не требуется для указания базового класса NonGen и даже не

может в нем использоваться. Следовательно, класс Gen наследует от класса NonGen

обычным образом, т.е. без выполнения каких-то особых условий.

**Переопределение виртуальных методов в обобщенном классе**

В обобщенном классе виртуальный метод может быть переопределен таким же образом,

как и любой другой метод. В качестве примера рассмотрим следующую программу,

в которой переопределяется виртуальный метод GetOb().

(***glava18\_20***)

//generalized base class

class Gen<T>

{

protected T ob;

public Gen(T o)

{

ob = o;

}

//return value ov ob. This method is virtual

public virtual T GetOb()

{

Console.Write("Method GetOb() from Gen class"

+ " returns result: ");

return ob;

}

}

//derive class from Gen

//this class redefines methos GetOb()

class Gen2<T> : Gen<T>

{

public Gen2(T o) :base(o) { }

//redefine method GetOb()

public override T GetOb()

{

Console.Write("Method GetOb() from class Gen2"

+ " returns result: ");

return ob;

}

}

//demonstrate redefined method int generalized class

class OverrideDemo

{

static void Main()

{

//create object Gen class with int type

Gen<int> iOb = new Gen<int>(88);

//here is GetOb() being called from Gen class

Console.WriteLine(iOb.GetOb());

//Now create object of Gen2 class and assign

//link to iOb type Gen<int>

iOb = new Gen2<int>(99);

//call method GetOb() from Gen2

Console.WriteLine(iOb.GetOb());

}

}

Как следует из результата выполнения приведенной выше программы, переопределяемый

вариант метода GetOb() вызывается для объекта типа Gen2, а его вариант из

базового класса вызывается для объекта типа Gen.

Обратите внимание на следующую строку кода.

iOb = new Gen2<int>(99);

Такое присваивание вполне допустимо, поскольку iOb является переменной типа

Gen<int>. Следовательно, она может ссылаться на любой объект типа Gen<int> или

же объект класса, производного от Gen<int>, включая и Gen2<int>. Разумеется, переменную

iOb нельзя использовать, например, для ссылки на объект типа Gen2<int>,

поскольку это может привести к несоответствию типов.

**Перегрузка методов с несколькими параметрами типа**

Методы, параметры которых объявляются с помощью параметров типа, могут быть

перегружены. Но правила их перегрузки упрощаются по сравнению с методами без

параметров типа. Как правило, метод, в котором параметр типа служит для указания

типа данных параметра этого метода, может быть перегружен при условии, что сигнатуры

обоих его вариантов отличаются. Это означает, что оба варианта перегружаемого

метода должны отличаться по типу или количеству их параметров. Но типовые

различия должны определяться не по параметру обобщенного типа, а исходя из аргумента

типа, подставляемого вместо параметра типа при конструировании объекта

этого типа. Следовательно, метод с параметрами типа может быть перегружен таким

образом, что он окажется пригодным не для всех возможных случаев, хотя и будет выглядеть

верно.

В качестве примера рассмотрим следующий обобщенный класс.

(***glava18\_21***)

//generalized class, contains method Set(), override

//of it can follow to error

class Gen <T, V>

{

T ob1;

V ob2;

//in some cases this two method wount be

//diferent by its parameters

public void Set(T o)

{

ob1 = o;

}

public void Set(V o)

{

ob2 = o;

}

}

class AmbiguityDemo

{

static void Main()

{

Gen<int, double> ok = new Gen<int, double>();

Gen<int, int> notOK = new Gen<int, int>();

ok.Set(10); //right, arguments is diferent

//notOK.Set(10); //ambigious, arguments are the same

}

}

Рассмотрим приведенный выше код более подробно. Прежде всего обратите внимание

на то, что класс Gen объявляется с двумя параметрами типа: Т и V. В классе Gen

метод Set() перегружается по параметрам типа Т и V, как показано ниже.

public void Set(T o)

{

ob1 = o;

}

public void Set(V o)

{

ob2 = o;

}

}

Такой подход кажется вполне обоснованным, поскольку типы Т и V ничем внешне

не отличаются. Но подобная перегрузка таит в себе потенциальную неоднозначность.

При таком объявлении класса Gen не соблюдается никаких требований к различению

типов Т и V. Например, нет ничего принципиально неправильного в том, что

объект класса Gen будет сконструирован так, как показано ниже.

Gen<int, int> notOK = new Gen<int, int>();

В данном случае оба типа, Т и V, заменяются типом int. В итоге оба варианта метода

Set() оказываются совершенно одинаковыми, что, разумеется, приводит к ошибке.

Следовательно, при последующей попытке вызвать метод Set() для объекта notOK в

методе Main() появится сообщение об ошибке вследствие неоднозначности во время

компиляции.

Как правило, методы с параметрами типа перегружаются при условии, что объект

конструируемого типа не приводит к конфликту. Следует, однако, иметь в виду, что

ограничения на типы не учитываются при разрешении конфликтов, возникающих при

перегрузке методов. Поэтому ограничения на типы нельзя использовать для исключения

неоднозначности. Конструкторы, операторы и индексаторы с параметрами типа

могут быть перегружены аналогично конструкторам по тем же самым правилам.

**Некоторые ограничения, присущие обобщениям**

Ниже перечислен ряд ограничений, которые следует иметь в виду при использовании

обобщений.

• Свойства, операторы, индексаторы и события не могут быть обобщенными. Но

эти элементы могут использоваться в обобщенном классе, причем с параметрами

обобщенного типа этого класса.

• К обобщенному методу нельзя применять модификатор extern.

• Типы указателей нельзя использовать в аргументах типа.

• Если обобщенный класс содержит поле типа static, то в объекте *каждого* конструируемого

типа должна быть *своя* копия этого поля. Эго означает, что во всех

экземплярах объектов *одного* конструируемого типа совместно используется одно

и то же поле типа static. Но в экземплярах объектов *другого* конструируемого

типа совместно используется другая копия этого поля. Следовательно, поле типа

static не может совместно использоваться объектами *всех* конструируемых типов.