**Ограниченные типы where**

В предыдущих примерах параметры типа можно было заменить любым типом

данных. Это означает, что вполне допустимо создавать объекты класса Gen, в которых тип Т

заменяется типом int, double, string, FileStream или любым другим типом данных.

Во многих случаях отсутствие ограничений на указание аргументов типа считается

вполне приемлемым, но иногда оказывается полезно ограничить круг типов, которые

могут быть указаны в качестве аргумента типа.

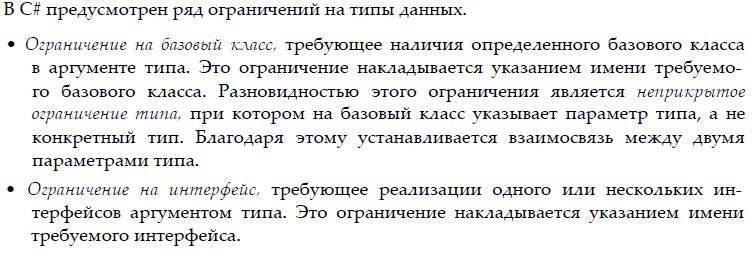
Для выхода из подобных ситуаций в C# предусмотрены *ограниченные типы.* Указывая

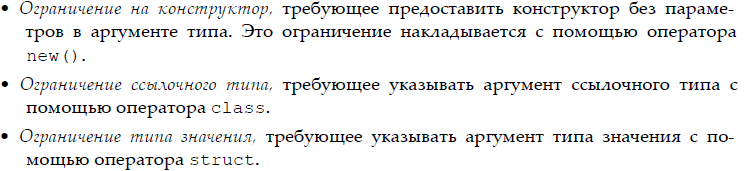
параметр типа, можно наложить определенное ограничение на этот параметр. Это

делается с помощью оператора where при указании параметра типа:

class имя\_класса<параметр\_типа> where параметр\_типа : ограничения { // ...

где *ограничения* указываются списком через запятую.





Среди всех этих ограничений чаще всего применяются ограничения на базовый

класс и интерфейс, хотя все они важны в равной степени. Каждое из этих ограничений

рассматривается далее по порядку.

**Применение ограничения на базовый класс**

Ограничение на базовый класс позволяет указывать базовый класс, который должен

наследоваться аргументом типа. Ограничение на базовый класс служит двум главным

целям. Во-первых, оно позволяет использовать в обобщенном классе те члены базового

класса, на которые указывает данное ограничение. Это дает, например, возможность

вызвать метод или обратиться к свойству базового класса. В отсутствие ограничения на

базовый класс компилятору ничего не известно о типе членов, которые может иметь

аргумент типа. Накладывая ограничение на базовый класс, вы тем самым даете компилятору

знать, что все аргументы типа будут иметь члены, определенные в этом базовом

классе.

И во-вторых, ограничение на базовый класс гарантирует использование только

тех аргументов типа, которые поддерживают указанный базовый класс. Это означает,

что для любого ограничения, накладываемого на базовый класс, аргумент типа

должен обозначать сам базовый класс или производный от него класс. Если же попытаться

использовать аргумент типа, не соответствующий указанному базовому

классу или не наследующий его, то в результате возникнет ошибка во время компиляции.

Ниже приведена общая форма наложения ограничения на базовый класс, в которой

используется оператор where:

where Т : имя\_базового\_класса

где *T* обозначает имя параметра типа, а *имя\_базового\_класса —* конкретное имя

ограничиваемого базового класса. Одновременно в этой форме ограничения может

быть указан только один базовый класс.

В приведенном ниже простом примере демонстрируется механизм наложения

ограничения на базовый класс.

(***glava18\_4***)

class A

{

public void Hello()

{

Console.WriteLine("Hello");

}

}

//class B derived from A class

class B : A { }

//class C is not derived from A class

class C { }

//in case of restriction of base class in all arguments of type

//passed to class Test, has to be base class A

class Test<T> where T : A

{

T obj;

public Test(T o)

{

obj = o;

}

public void SayHello()

{

//method Hello() calls, its in base class A

obj.Hello();

}

}

class BaseClassConstraintDemo

{

static void Main()

{

A a = new A();

B b = new B();

C c = new C();

//next code is OK, becasue of class A is base

Test<A> t1 = new Test<A>(a);

//next code is allowed, bcs of class B is derived from A class

Test<B> t2 = new Test<B>(b);

t2.SayHello();

//next code is not allowed, class C is not derived from A class

//Test<C> t3 = new Test<C>(c); //Error!

//t3.SayHello(); //Error!

}

}

В данном примере кода класс А наследуется классом В, но не наследуется классом

С. Обратите также внимание на то, что в классе А объявляется метод Hello(), а класс

Test объявляется как обобщенный следующим образом.

class Test<T> where T : A

Оператор where в этом объявлении накладывает следующее ограничение: любой

аргумент, указываемый для типа Т, должен иметь класс А в качестве базового.

А теперь обратите внимание на то, что в классе Test объявляется метод

SayHello(), как показано ниже.

public void SayHello()

{

//method Hello() calls, its in base class A

obj.Hello();

}

Этот метод вызывает в свою очередь метод Hello() для объекта obj типа Т. Любопытно,

что единственным основанием для вызова метода Hello() служит следующее

требование ограничения на базовый класс: любой аргумент типа, привязанный к типу

Т, должен относиться к классу А или наследовать от класса А, в котором объявлен метод

Hello(). Следовательно, любой допустимый тип Т будет также определять метод

Hello(). Если бы данное ограничение на базовый класс не было наложено, то компилятору

ничего не было бы известно о том, что метод Hello() может быть вызван для

объекта типа Т. Убедитесь в этом сами, удалив оператор where из объявления обобщенного

класса Test. В этом случае программа не подлежит компиляции, поскольку

теперь метод Hello() неизвестен.

Помимо разрешения доступа к членам базового класса, ограничение на базовый

класс гарантирует, что в качестве аргументов типа могут быть переданы только те типы

данных, которые наследуют базовый класс. Именно поэтому приведенные ниже строки

кода закомментированы.

//Test<C> t3 = new Test<C>(c); //Error!

//t3.SayHello(); //Error!

Прежде чем продолжить изложение дальше, рассмотрим вкратце два последствия

наложения ограничения на базовый класс. Во-первых, это ограничение разрешает доступ

к членам базового класса из обобщенного класса. И во-вторых, оно гарантирует

допустимость только тех аргументов типа, которые удовлетворяют данному ограничению,

обеспечивая тем самым типовую безопасность.

В предыдущем примере показано, как накладывается ограничение на базовый

класс, но из него не совсем ясно, зачем это вообще нужно. Для того чтобы особое значение

ограничения на базовый класс стало понятнее, рассмотрим еще один, более

практический пример. Допустим, что требуется реализовать механизм управления

списками телефонных номеров, чтобы пользоваться разными категориями таких списков,

в частности отдельными списками для друзей, поставщиков, клиентов и т.д. Для

этой цели можно сначала создать класс PhoneNumber, в котором будут храниться имя

абонента и номер его телефона. Такой класс может иметь следующий вид.

(***abonents.cs***)

class Phonenumber

{

public Phonenumber(string n, string num)

{

Name = n;

Number = num;

}

//auto realised properties

//it contains name of abonent and number

public string Number { get; set; }

public string Name { get; set; }

}

Далее создадим классы, наследующие класс PhoneNumber: Friend и Supplier.

Эти классы приведены ниже.

//class for friends numbers

class Friend : Phonenumber

{

public Friend(string n, string num, bool wk) :

base(n, num)

{

IsWorkNumber = wk;

}

public bool IsWorkNumber { get; private set; }

}

//class for courier numbers

class Supplier : Phonenumber

{

public Supplier(string n, string num) :

base(n, num)

{ }

}

Обратите внимание на то, что в класс Friend введено свойство IsWorkNumber, возвращающее

логическое значение true, если номер телефона является рабочим.

Для управления списками телефонных номеров создадим еще один класс под названием

PhoneList. Его следует сделать обобщенным, поскольку он должен служить

для управления любым списком телефонных номеров. В функции такого управления

должен, в частности, входить поиск телефонных номеров по заданным именам и наоборот,

поэтому на данный класс необходимо наложить ограничение по типу, требующее,

чтобы объекты, сохраняемые в списке, были экземплярами класса, производного

от класса PhoneNumber.

//class PfconeList can manage any list of phone numbers,

//in case of it derived from PhoneNumber class

class PhoneList<T> where T:Phonenumber

{

T[] phList;

int end;

public PhoneList()

{

phList = new T[10];

end = 0;

}

//addelement int list

public bool Add(T newEntry)

{

if (end == 10) return false;

phList[end] = newEntry;

end++;

return true;

}

//find and return data about number by Name

public T FindByName(string name)

{

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

{

//Name could be used, becs of its propertie Name

//deliver to member of PhoneNumber class, and its base

//with this restriction

if (phList[i].Name == name)

return phList[i];

}

//no such name in list

throw new NotFoundException()

}

//find and return data about number by Number

public T FindByNumber(string number)

{

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

{

//Number could be used, becs of its propertie Name

//deliver to member of PhoneNumber class, and its base

//with this restriction

if (phList[i].Number == number)

return phList[i];

}

//no such number in list

throw new NotFoundException();

}

}

Ограничение на базовый класс разрешает коду в классе PhoneList доступ к свойствам

Name и Number для управления любым видом списка телефонных номеров. Оно

гарантирует также, что для построения объекта класса PhoneList будут использоваться

только доступные типы. Обратите внимание на то, что в классе PhoneList генерируется

исключение NotFoundException, если имя или номер телефона не найдены.

Это специальное исключение, объявляемое ниже.

class NotFoundException : Exception

{

//define all constructor of class Exception

//this constructors make call to constructor of base class

//class notFoundException dont realise new methods

//not necessary any additional

public NotFoundException() : base() { }

public NotFoundException(string str) : base(str) { }

public NotFoundException(

string str, Exception inner) : base(str, inner) { }

protected NotFoundException(

System.Runtime.Serialization.SerializationInfo si,

System.Runtime.Serialization.StreamingContext sc) : base(si, sc) { }

}

В данном примере используется только конструктор, вызываемый по умолчанию,

но ради наглядности этого примера в классе исключения NotFoundException реализуются

все конструкторы, определенные в классе Exception. Обратите внимание на

то, что эти конструкторы вызывают эквивалентный конструктор базового класса, определенный

в классе Exception. А поскольку класс исключения NotFoundException

ничем не дополняет базовый класс Exception, то для любых дополнительных действий

нет никаких оснований.

В приведенной ниже программе все рассмотренные выше фрагменты кода объединяются

вместе, а затем демонстрируется применение класса PhoneList. Кроме

того, в ней создается класс EmailFriend. Этот класс не наследует от класса

PhoneNumber, а следовательно, он *не* может использоваться для создания объектов

класса PhoneList.

(***glava18\_5***)

//demonstrate restriction on base class

class UseBaseClassConstraint

{

static void Main()

{

//next code is Ok, bsc of

//class Friend derived from PhoneNumber

PhoneList<Friend> plist = new PhoneList<Friend>();

plist.Add(new Friend("Tom", "555-123", true));

plist.Add(new Friend("Harry", "555-6756", true));

plist.Add(new Friend("Math", "555-9254", false));

try

{

//find number by Name

Friend frnd = plist.FindByName("Harry");

Console.Write(frnd.Name + " " + frnd.Number);

if (frnd.IsWorkNumber)

Console.WriteLine(" (work number)");

else

Console.WriteLine();

}

catch (NotFoundException)

{

Console.WriteLine("Not found.");

}

Console.WriteLine();

//next code is ok, class Suplier derived class PhoneNumber

PhoneList<Supplier> plist2 = new PhoneList<Supplier>();

plist2.Add(new Supplier("Global Hardware", "555-8834"));

plist2.Add(new Supplier("Computer Warehouse agency", "555-9256"));

plist2.Add(new Supplier("NetworkCity Company", "555-2564"));

try

{

//find by number

Supplier sp = plist2.FindByNumber("555-2564");

Console.WriteLine(sp.Name + " " + sp.Number);

}

catch(NotFoundException)

{

Console.WriteLine("Not Found.");

}

}

}

**Применение ограничения на интерфейс**

Ограничение на интерфейс позволяет указывать интерфейс, который должен быть

реализован аргументом типа. Это ограничение служит тем же основным целям, что и

ограничение на базовый класс. Во-первых, оно позволяет использовать члены интерфейса

в обобщенном классе. И во-вторых, оно гарантирует использование только тех

аргументов типа, которые реализуют указанный интерфейс. Это означает, что для любого

ограничения, накладываемого на интерфейс, аргумент типа должен обозначать

сам интерфейс или же тип, реализующий этот интерфейс.

Ниже приведена общая форма наложения ограничения на интерфейс, в которой

используется оператор where:

where Т : имя\_интерфейса

где *Т* — это имя параметра типа, а *имя\_интерфейса* — конкретное имя ограничиваемого

интерфейса. В этой форме ограничения может быть указан список интерфейсов

через запятую. Если ограничение накладывается одновременно на базовый класс и интерфейс,

то первым в списке должен быть указан базовый класс.

Ниже приведена программа, демонстрирующая наложение ограничения на интерфейс

и представляющая собой переработанный вариант предыдущего примера

программы, управляющей списками телефонных номеров. В этом варианте класс

PhoneNumber преобразован в интерфейс IPhoneNumber, который реализуется в классах

Friend и Supplier. (Короче батя вещает, имеется в виду, что обобщенный класс допускает использовать в себе только те классы, которые реализовали интерфейс, который указан после слова where)

(***glava18\_6***)

class NotFoundException : Exception

{

public NotFoundException() : base() { }

public NotFoundException(string str) : base(str) { }

public NotFoundException(

string str, Exception inner) : base(str, inner) { }

protected NotFoundException(

System.Runtime.Serialization.SerializationInfo si,

System.Runtime.Serialization.StreamingContext sc) : base(si, sc) { }

}

//interface with Name and Number

public interface IPhoneNumber

{

string Number { get; set; }

string Name { get; set; }

}

//class for Friends number, with interface IPhoneNumber

class Friend : IPhoneNumber

{

//realise interface

public string Number { get; set; }

public string Name { get; set; }

public Friend(string n, string num, bool wk)

{

Name = n;

Number = num;

IsWorkNumber = wk;

}

public bool IsWorkNumber { get; private set; }

}

//class for Suppliers

class Supplier : IPhoneNumber

{

public string Number { get; set; }

public string Name { get; set; }

public Supplier(string n, string num)

{

Number = num;

Name = n;

}

}

//IPhoneNumber NOT realised

class EmailFriend

{

//...

}

//class PhoneList can manage any list of numbers,

//only if class realize interface IPhoneNumber

class PhoneList<T> where T : IPhoneNumber

{

T[] phList;

int end;

public PhoneList()

{

phList = new T[10];

end = 0;

}

public bool Add(T newEntry)

{

if (end == 10) return false;

phList[end] = newEntry;

end++;

return true;

}

//find and return number by Name

public T FindByName(string name)

{

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

{

//Name could be used, its property delivers to

//members of IPhoneNumber interface

if (phList[i].Name == name)

return phList[i];

}

//no such Name in list

throw new NotFoundException();

}

//find info by Number

public T FindByNumber(string number)

{

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

{

//Number could be used, its property delivers to

//members of IPhoneNumber interface

if (phList[i].Number == number)

return phList[i];

}

//no such number in list

throw new NotFoundException();

}

}

//Demonstrate

class UseInterfaceConstraint

{

static void Main()

{

//next code is allowed, class Friend realised

//IPhoneNumber interface

PhoneList<Friend> plist = new PhoneList<Friend>();

plist.Add(new Friend("Tom", "555-1234", true));

plist.Add(new Friend("Harry", "555-6756", true));

plist.Add(new Friend("Matt", "555-9254", false));

try

{

//find by Name

Friend frnd = plist.FindByName("Harry");

Console.WriteLine(frnd.Name + " " + frnd.Number);

if (frnd.IsWorkNumber) Console.WriteLine("Work number");

else Console.WriteLine();

}

catch(NotFoundException)

{

Console.WriteLine("Number has not found.");

}

Console.WriteLine();

//next code is allowed, class Supplier realised

//IPhoneNumber interface

PhoneList<Supplier> plist2 = new PhoneList<Supplier>();

plist2.Add(new Supplier("Global Hardware", "555-8834"));

plist2.Add(new Supplier("Computer Warehouse", "555-9256"));

plist2.Add(new Supplier("NetworkCity", "555-2564"));

try

{

//find name by Number

Supplier sp = plist2.FindByNumber("555-2564");

Console.WriteLine(sp.Name + " " + sp.Number);

}

catch(NotFoundException)

{

Console.WriteLine("Has not found.");

}

}

}

**Применение ограничения new() на конструктор**

Ограничение new() на конструктор позволяет получать экземпляр объекта обобщенного

типа. Как правило, создать экземпляр параметра обобщенного типа не удается.

Но это положение изменяет ограничение new(), поскольку оно требует, чтобы

аргумент типа предоставил конструктор без параметров. Им может быть конструктор,

вызываемый по умолчанию и предоставляемый автоматически, если явно определяемый

конструктор отсутствует или же конструктор без параметров явно объявлен пользователем.

Накладывая ограничение new(), можно вызывать конструктор без параметров

для создания объекта.

Ниже приведен простой пример, демонстрирующий наложение ограничения

(***glava18\_7***)

class MyClass

{

public MyClass() { }

}

class Test<T> where T : new()

{

T obj;

public Test()

{

//but for new(), it will work

obj = new T();

}

}

class ConsConstraintDemo

{

static void Main()

{

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

}

}

Прежде всего обратите внимание на объявление класса Test.

class Test<T> where T : new()

В силу накладываемого ограничения new() любой аргумент типа должен предоставлять

конструктор без параметров.

Далее проанализируем приведенный ниже конструктор класса Test.

public Test()

{

//but for new(), it will work

obj = new T();

}

В этом фрагменте кода создается объект типа Т, и ссылка на него присваивается

переменной экземпляра obj. Такой код допустим только потому, что ограничение

new() требует наличия конструктора. Для того чтобы убедиться в этом, попробуйте

сначала удалить ограничение new(), а затем попытайтесь перекомпилировать программу.

В итоге вы получите сообщение об ошибке во время компиляции.

Следует особо подчеркнуть, что в классе MyClass совсем не обязательно определять

конструктор без параметров явным образом. Его используемый по умолчанию конструктор

вполне удовлетворяет накладываемому ограничению. Но если классу потребуются

другие конструкторы, помимо конструктора без параметров, то придется

объявить явным образом и вариант без параметров. Что касается применения ограничения new(), то следует обратить внимание на три других важных момента. Во-первых, его можно использовать вместе с другими ограничениями, но последним по порядку. Во-вторых, ограничение new() позволяетконструировать объект, используя только конструктор без параметров, — даже если доступны другие конструкторы. Иными словами, передавать аргументы конструктору параметра типа не разрешается. И в-третьих, ограничение new() нельзя использовать одновременно с ограничением типа значения, рассматриваемым далее.

**Ограничения ссылочного типа и типа значения**

Два других ограничения позволяют указать на то, что аргумент, обозначающий тип,

должен быть либо ссылочного типа, либо типа значения. Эти ограничения оказываются

полезными в тех случаях, когда для обобщенного кода важно провести различие

между ссылочным типом и типом значения. Ниже приведена общая форма ограничения

ссылочного типа.

where Т : class

В этой форме с оператором where ключевое слово class указывает на то, что аргумент

*Т* должен быть ссылочного типа. Следовательно, всякая попытка использовать

тип значения, например int или bool, вместо *Т* приведет к ошибке во время компиляции.

where Т : struct

В этой форме ключевое слово struct указывает на то, что аргумент Т должен быть

типа значения. (Напомним, что структуры относятся к типам значений.) Следовательно,

всякая попытка использовать ссылочный тип, например string, вместо *T* приведет

к ошибке во время компиляции. Но если имеются дополнительные ограничения, то

в любом случае class или struct должно быть первым по порядку накладываемым

ограничением.

Ниже приведен пример, демонстрирующий наложение ограничения ссылочного типа.

(***glava18\_8***)

class MyClass { }

//restriction link type

class Test<T> where T : class

{

T obj;

public Test()

{

//next operator is allowed only because of

//argument T is guaranted deliverd to Link Type

//its allowed to define it as null value

obj = null;

}

}

class ClassConstraintDemo

{

static void Main()

{

//next code is OK, MyClass is a class

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

//next code wount work, int is value type

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

}

}

Обратите внимание на следующее объявление класса Test.

class Test<T> where T : class

Ограничение class требует, чтобы любой аргумент Т был ссылочного типа. В данном

примере кода это необходимо для правильного выполнения операции присваивания

в конструкторе класса Test.

public Test()

{

//next operator is allowed only because of

//argument T is guaranted deliverd to Link Type

//its allowed to define it as null value

obj = null;

}

В этом фрагменте кода переменной obj типа T присваивается пустое значение. Такое

присваивание допустимо только для ссылочных типов. Как правило, пустое значение

нельзя присвоить переменной типа значения. (Исключением из этого правила

является *обнуляемый тип,* который представляет собой специальный тип структуры,

инкапсулирующий тип значения и допускающий пустое значение (null). Подробнее

об этом — в главе 20.) Следовательно, в отсутствие ограничения такое присваивание

было бы недопустимым, и код не подлежал бы компиляции. Это один из тех случаев,

когда для обобщенного кода может оказаться очень важным различие между типами

значений и ссылочными типами.

Ограничение типа значения является дополнением ограничения ссылочного типа.

Оно просто гарантирует, что любой аргумент, обозначающий тип, должен быть типа

значения, в том числе struct и enum. (В данном случае обнуляемый тип не относится

к типу значения.) Ниже приведен пример наложения ограничения типа значения.

(***glava18\_8.1***)

class MyClass { }

struct MyStruct { }

//restriction link type

class Test<T> where T : struct

{

T obj;

public Test(T x)

{

obj = x;

}

}

class ValueConstraintDemo

{

static void Main()

{

//next code is OK

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

//next code is NOT OK, MyClass is a class type

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

}

}

В этом примере кода класс Test объявляется следующим образом.

class Test<T> where T : struct

На параметр типа Т в классе Test накладывается ограничение struct, и поэтому

к нему могут быть привязаны только аргументы типа значения. Это означает, что

объявления Test<MyStruct> и Test<int> вполне допустимы, тогда как объявление

Test<MyClass> недопустимо.

**Установление связи между двумя параметрами типа с помощью ограничения**

Существует разновидность ограничения на базовый класс, позволяющая установить

связь между двумя параметрами типа. В качестве примера рассмотрим следующее

объявление обобщенного класса.

class Gen<T; V> where V : T {

В этом объявлении оператор where уведомляет компилятор о том, что аргумент

типа, привязанный к параметру типа V, должен быть таким же, как и аргумент типа,

привязанный к параметру типа Т, или же наследовать от него. Если подобная связь

отсутствует при объявлении объекта типа Gen, то во время компиляции возникнет

ошибка. Такое ограничение на параметр типа называется неприкрытым ограничением

типа. В приведенном ниже примере демонстрируется наложение этого ограничения.

(***glava18\_9***)

class A { }

class B : A { }

//parameter V has to bo derived from T

class Gen<T, V> where V : T

{

}

class NakedConstraintDemo

{

static void Main()

{

//next code is allowed, B derived A

Gen<A, B> x = new Gen<A, B>();

//next code is not allowed, A is not derived from B

//Gen<B, A> y = new Gen<B, A>();

}

}

Обратите внимание на то, что класс В наследует от класса А. Проанализируем далее

оба объявления объектов класса Gen в методе Main(). Как следует из комментария

к первому объявлению

Gen<A, B> x = new Gen<A, B>();

оно вполне допустимо, поскольку класс В наследует от класса А. Но второе объявление

//Gen<B, A> y = new Gen<B, A>();

недопустимо, поскольку класс А не наследует от класса В.

**Применение нескольких ограничений**

С параметром типа может быть связано несколько ограничений. В этом случае

ограничения указываются списком через запятую. В этом списке первым должно быть

указано ограничение class либо struct, если оно присутствует, или же ограничение

на базовый класс, если оно накладывается. Указывать ограничения class или struct

одновременно с ограничением на базовый класс не разрешается. Далее по списку

должно следовать ограничение на интерфейс, а последним по порядку — ограничение

new(). Например, следующее объявление считается вполне допустимым.

class Gen<T> where Т : MyClass, IMyInterface, new()

В данном случае параметр типа Т должен быть заменен аргументом типа, наследующим

от класса MyClass, реализующим интерфейс IMyInterface и использующим

конструктор без параметра.

Если же в обобщении используются два или более параметра типа, то ограничения

на каждый из них накладываются с помощью отдельного оператора where, как в приведенном

ниже примере.

(***glava18\_10***)

//class Gen has two parameters type, both has constraints

//wtih diferent where operators

class Gen<T, V> where T:class

where V : struct

{

T ob1;

V ob2;

public Gen(T t, V v)

{

ob1 = t;

ob2 = v;

}

}

class MultipleConstrainsDemo

{

static void Main()

{

//string - link type, int - value type

Gen<string, int> obj = new Gen<string, int>("test", 11);

//bool is not link type

//Gen<bool, int> obj = new Gen<bool, int>(true, 11);

}

}

В данном примере класс Gen принимает два аргумента с ограничениями, накладываемыми

с помощью отдельных операторов where. Обратите особое внимание на

объявление этого класса.

class Gen<T, V> where T:class

where V : struct

Как видите, один оператор where отделяется от другого только пробелом. Другие

знаки препинания между ними не нужны и даже недопустимы.