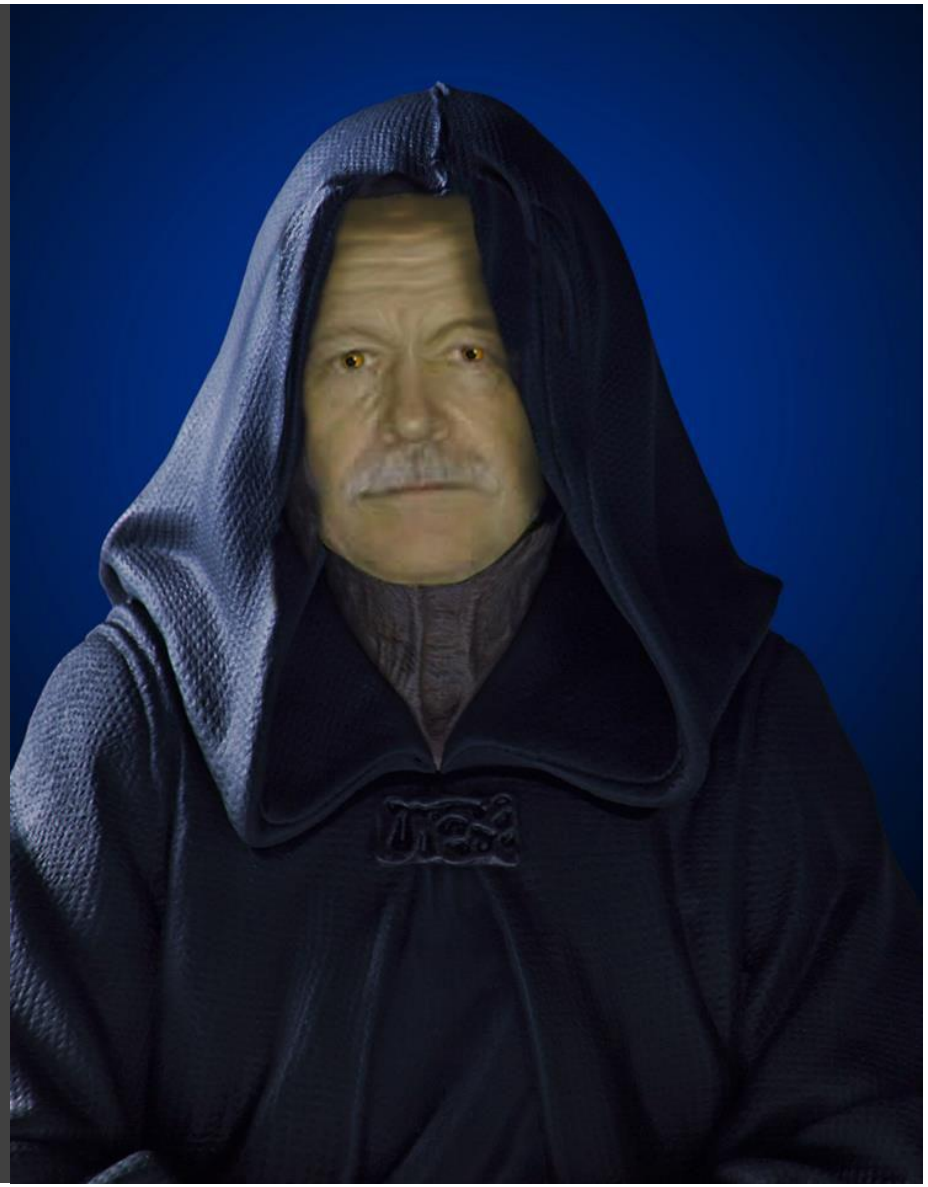
ПИШНИК: BECOME HUMAN

Глава 15: Универсальные шаблоны

Автор презентации – Сагалов Даниил БПИ-196



Обзор Возможностей Универсальных Шаблонов

Универсальные шаблоны позволяют параметризовать типы данных. Простым языком, Вы можете воспринимать их как «переменные для типов». Их применение даёт возможность добиться ещё большей гибкости кода.

Универсальные типы, делегаты и методы сочетают в себе такие характеристики, как возможность многократного использования, эффективность и типобезопасность.

И так, универсальные шаблоны позволяют:

- Получить возможность многократно использовать код, избегая необходимость дублирования.
- Создавать собственные типы коллекций.
- Определить собственные универсальные классы, структуры, интерфейсы, методы, события и делегаты.
- Ограничить допустимые значения типизирующих параметров с помощью ограничений.
- Узнать сведения о типах, используемых в универсальном с помощью рефлексии.

Типизирующие Параметры

Для того, чтобы создать универсальный элемент, вам при объявлении в треугольных скобках необходимо объявить один или несколько типизирующих параметров, например:

public class GenericClass<T, S> {…} , где T, S – типизирующие параметры. Помните, что сам тип GenericClass<T, S> скорее является заготовкой типа, вы не можете создать его экземпляры без указания аргументов типа.

В этот момент вводится понятие **открытого сконструированного типа** – типа, содержащего неопределённые типизирующие параметры (в данном случае, T и S). При указании значений типизирующих параметров создаётся **закрытый сконструированный тип**, создать объекты открытого типа нельзя. Важно осознавать, что все сконструированные типы с разными типизирующими параметрами являются разными типами:

```
GenericClass<int, int> gc1 = new GenericClass<int, int>( );
GenericClass<string, double> gc2 = new GenericClass<string, double>( );
GenericClass<List<int>, char> gc3 = new GenericClass<List<int>, char>( );
gc1 = gc2; // Так нельзя – используются два различных типа!
```

При именовании типизирующих параметров рекомендуется либо использовать одну букву (T, S, U…), если параметры не несут смысловой нагрузки и названия с префиксом T в противном случае (например, как TKey, TValue в System.Collections.Generic.Dictionary).

Пример

```
public class Stack<T> {
    private T[] _array;
    private int _size;
    private int _capacity = 4;
    ссылка: 1
    public Stack() => _array = new T[4];
    Ссылок: 0
    public Stack(int capacity) {
        if (capacity < 0)
            throw new ArgumentOutOfRangeException($"Invalid capacity: {capacity}");
        _array = new T[capacity];
        _capacity = capacity;
    }
    Ссылок: 2
    public int Count { get => _size; }
    ссылка: 1
    public void Clear() {
        if (_size != 0) {
            Array.Clear(_array, 0, _array.Length);
            Array.Resize(ref _array, 4);
            _capacity = 4;
            _size = 0;
        }
    }
    Ссылок: 12
    public void Push(T item) {
        if (_size == _array.Length) {
            _capacity *= 2;
            Array.Resize(ref _array, _capacity);
        }
        _array[_size] = item;
        ++_size;
    }

    Ссылок: 11
    public T Pop() {
        if (_size == 0)
            throw new ArgumentOutOfRangeException("The stack is empty.");
        T removed = _array[_size-1];
        _array[_size] = default;
        --_size;
        return removed;
    }
}
```

Данный пример показывает упрощённую реализацию коллекции Stack<T>.

Типизирующий параметр T определяет тип данных, который будет храниться в стеке. Обратите внимание, что T – любой тип, данный класс лишь задаёт правила, по которым элементы типа T будут храниться.

В качестве контейнера взят самый обычный массив.

Ограничения Параметров Универсальных Типов

В общем случае при создании типизирующих параметров они могут иметь абсолютно любой тип. Однако зачастую возникает необходимость совершать какие-либо более конкретные действия с объектами, для чего вводятся ограничения типизирующих параметров.

Ограничения позволяют сообщать компилятору о характеристиках, которые должны иметь аргументы типа. Для того, чтобы указать ограничения используется следующий синтаксис с контекстно-ключевым словом **where**:

```
public class GExample1<T> where T : class, IComparable<T>, new( ) {…}
public class GExample2<T, S>
where T : struct                    // Ограничение на 2 универсальных типа
where S : class, ICloneable {…}
```

Как видите, вы можете указать несколько ограничений через запятую, при этом существуют определённый порядок, в котором ограничения должны указываться.

При указании ограничений на первом месте в списке могут стоять **class/struct/notnull/unmanaged**, затем может быть указано одно конкретное **ограничение типа класса**, затем **ограничения типов интерфейсов**, в конце – **ограничение new( )**. Ограничения типов наследуются!

Виды Ограничений Универсального Типа

Ограничение	Описание
where T : class	Аргумент типа должен быть ссылочным типом. Данное ограничение применимо к любому типу класса, интерфейса, делегата или массива.
where T : struct	Аргумент типа должен быть (не nullable) типом значения. Нельзя использовать вместе с ограничениями new( ) (структуры подразумевают наличие конструктора по умолчанию) и unmanaged. Несовместимо с ограничениями на конкретный тип (т. к. структуры не наследуются).
where T : notnull (C# 8.0)	Аргумент типа должен быть любым ссылочным или значимым типом, не допускающим значение null. Данное ограничение применимо к любому типу класса, интерфейса, делегата или массива.
where T : unmanaged (C# 7.3)	Аргумент типа должен быть любым (не nullable) неуправляемым типом. Данное ограничение подразумевает ограничение struct, поэтому его нельзя совместить с ограничениями struct и new( ).
where T : <Имя Класса>	Аргумент типа должен быть данным типом или производным от него, неприменимо к структурам.
where T : <Имя Интерфейса>	Аргумент типа должен быть данным интерфейсом или реализовывать его. Можно указать несколько ограничений интерфейса через запятую, интерфейсы могут быть универсальными.
where T : new( )	Аргумент типа должен иметь публичный конструктор без параметров. При указании нескольких ограничений данное должно быть последним. Несовместимо с ограничениями struct и unmanaged.
where T : U	Аргумент типа T должен быть аргументом типа U или производным от него.

Особенность Ограничения where T : class

Всегда помните об одной особенности ограничения **where T : class**. При применении только данного ограничения не рекомендуется использовать операторы **==** и **!=** для параметров, так как для них будет всегда проверяться только равенство ссылок, а не значений. Что примечательно, эта особенность актуальны и для типов, перегружающих данные операторы.

Пример (с сайта Microsoft):

```
class Program {
    public static void OpEqualsTest<T>(T s, T t) where T : class {
        System.Console.WriteLine(s == t);
    }
    private static void TestStringEquality( ) {
        string s1 = "target";
        StringBuilder sb = new StringBuilder("target");
        string s2 = sb.ToString( );
        OpEqualsTest(s1, s2); // При вызове вернёт false, хотя строки равны!
    }
}
```

Универсальные Классы. Правила

Как мы уже поняли, для того, чтобы создать универсальный класс, вам в объявлении в треугольных скобках необходимо объявить один или несколько типизирующих параметров, например:

public class GenericClass<T, S> {…} , где T, S – типизирующие параметры.

При этом важно учитывать природу универсальных классов при работе с ними. Так, при наследовании универсальный класс может наследоваться как от конкретных (неуниверсальных) классов, так и от других открытых или закрытых сконструированных:

```
public class Generic<T> {…}
public class NonGeneric {…}
// Открытый сконструированный → открытый сконструированный
public class GenericExample1<T> : Generic<T> {…}
// Открытый сконструированный → закрытый сконструированный
public class GenericExample2<T> : Generic<int> {…}
// Открытый сконструированный → конкретный
public class GenericExample3<T> : NonGeneric{…}
```

Универсальные Классы. Правила

Конкретные классы не могут наследоваться от открытых сконструированных классов или от типизирующих параметров, возможно наследование только от закрытых сконструированных классов:

```
public class Generic1<T> {…}
// Конкретный → закрытый сконструированный
public class NonGenericExample : Generic1<int> {…}
// Ошибка Компиляции
public class BadNonGenericExample : Generic1<T> {…}
```

Ещё одно правило – при наследовании открытых сконструированных типов наследники должны предоставить все аргументы типизирующим параметрам базового класса (которые тоже могут быть типизирующими параметрами):

```
public class Generic2<T, U> {…}
public class GenericExample1<T, U> : Generic2<T, U> {…} // OK
public class GenericExample2<T> : Generic2<string, T> {…} // Тоже OK
// Такой код не скомпилируется – типизирующий параметр U не указан
public class BadGenericExample<U> : Generic2<T, U> {…}
```

Универсальные Классы. Правила

Классы, наследующиеся от открытых сконструированных также должны задавать множество всех ограничений родительского класса для каждого из типизирующих параметров:

```
public class Generic3<T, U>
    where T : System.IComparable<T>, new( )
    where U : struct {…}
public class GenericExample<T, U> : Generic3<T, U>
    where T : System.IComparable<T>, new( )
    where U : struct {…}
```

В дополнение стоит сказать, что сконструированные классы (как открытые, так и закрытые) могут использоваться в качестве параметров методов. Также разрешается приводить их к ссылкам типов реализуемых интерфейсов.

Всегда помните, что типизирующие по умолчанию параметры инвариантны, т. е. вместо них нельзя подставлять родителей/наследников указанного типа.

Универсальные Типы: Пример

Пример на следующих слайдах демонстрирует применение типизирующих параметров и их ограничений в различных сценариях (обобщённый интерфейс, обобщённый класс, наследование, обобщённый метод).

Обобщённый интерфейс IHasBrains<T> может быть реализован любым ссылочным типом.

В обобщённом классе Zombie задаётся публичный виртуальный метод Braaaains<T>(T target), где T может быть только классами, реализующими IHasBrains<T>.

Класс Husk – наследник Zombie переопределяет метод Braaaains<T>(T target), причём ограничения типа T сохраняются.

Универсальные Типы: Пример

```
public interface IHasBrains<T> where T : class
{
    // Обобщённый интерфейс IHasBrains только для классов
    ссылка: 1
    int GetBrains();
}
```

```
public class Zombie
{
    Ссылок: 3
    public int Hunger { get; protected set; }
    public const bool HasBrains = false;
    // Nullable-параметр для возможности задать неопределённое состояние зомби
    Ссылок: 3
    public bool? BurnsOnSun { get; protected set; }
    // Обобщённый метод, принимающий любой объект, реализующий IHasBrains
    Ссылок: 3
    public virtual void Braaaains<T>(T target) where T : class, IHasBrains<T>
    {
        Hunger -= target.GetBrains();
    }

    Ссылок: 0
    public Zombie()
    {
        BurnsOnSun = true;
        Hunger = int.MaxValue;
    }
}
```

```
public sealed class Husk : Zombie
{
    Ссылок: 2
    public int Intelligence { get; private set; }
    // Обратите внимание, что мы не пишем ограничения типа T повторно,
    // Они наследуются от класса-родителя, повторное объявление вызовет
    // ошибку компиляции в данном методе.
    Ссылок: 3
    public override void Braaaains<T>(T target)
    {
        if (Intelligence < 50)
            base.Braaaains(target);
        else
            Console.WriteLine("Come on, sir... Eating brains is impolite.");
    }
    Ссылок: 0
    public Husk()
    {
        Hunger = 20;
        // Неизвестно, может ли данный подвид зомби гореть на солнце.
        BurnsOnSun = null;
        Intelligence = 8;
    }
}
```

Универсальные Типы: Пример

В данном примере создаётся универсальная коллекция объектов Zombie, причём все они обязаны иметь конструктор без параметров, а жертвы зомби обязаны быть ссылочными типами, реализующими интерфейс IHasBrains<Tvictim>.

```
public class ZombieHorde<TZombie, TVictim>
    where TZombie : Zombie, new()
    where TVictim : class, IHasBrains<TVictim>
    // ^^^ Два типизирующих параметра TZombie и TVictim.
    // Обратите внимание, что ограничения TVictim должны обязывают
    // TVictim реализовывать интерфейс IHasBrains<TVictim>
{
    TZombie[] zombies;
    TVictim target = null;

    Ссылок: 0
    public void Infect()
    {
        Array.ForEach(zombies, zombie => zombie.Braaaains(target));
    }
    // Объекты типа TVictim без проблем могут быть параметрами методов
    Ссылок: 0
    public ZombieHorde(int count, TVictim target)
    {
        zombies = new TZombie[count];
        this.target = target;
    }
}
```

Универсальные Методы

Метод называется универсальным, если он использует типизирующие параметры в треугольных скобках сразу после имени метода, например:

```
public static string GenericExampleMethod<T>(T obj) => obj.ToString( );
```

При самом вызове метода аргумент в треугольных скобках можно опустить, в таком случае он будет подобран компилятором, т. е. данные вызовы эквивалентны:

```
GenericExampleMethod<int>(100);
GenericExampleMethod(100);
```

Компилятор одинаково способен определять типы аргументов как для статических, так и для экземплярных методов, исключениями являются только where-ограничения и типы возвращаемых значений. Также вы обязательно должны указывать в треугольных скобках тип при вызове метода, если у него нет никаких параметров. Методы с различным набором типизирующих параметров считаются за разные перегрузки.

Определение типов универсальных методов происходит на этапе компиляции, до того как сигнатуры перегрузок будут разрешены. При этом компилятор будет применять логику определения методов ко всем одноимённым универсальным методам. В итоге на этапе разрешения перегрузки компилятор включит только те методы, тип которых определён успешно.

Универсальные Методы в Классах

В универсальных классах все методы по умолчанию имеют доступ к параметрам типа, т. е. такой код скомпилируется и будет работать:

```
public abstract class ExampleGenericMethodClass<T> {
    public void Equals(ref T other);
}
```

Крайне важно понимать, что при явном определении параметра(ов) типа в универсальном методе с таким же именем, как и в универсальном классе будет работать как **сокрытие типизирующего(щих) параметра(ов)**. В таких ситуациях компилятор всегда будет выдавать предупреждение, схожее с тем, что вы можете увидеть при иных видах сокрытия. В таком примере тип T в классе и тип T в методе абсолютно не связаны:

```
public class TypeOverlapGeneric<T> {
    public void SomeMethod<T>() {…}
}
```

Универсальные методы сами могут накладывать ограничения на типизирующие параметры, для этого нужно перечислить ограничения после where:

```
public class WhereConstraintMethodExample { // Вы можете использовать универсальные методы в обычном классе
    public void SwapIfGreater<T>(ref T left, ref T right) where T : IComparable<T> {…}
}
```

Вариативность Типов в C#

В курсе мы уже сталкивались с понятием **ковариантности**: Вы можете располагать объекты дочерних классов по ссылкам родительских типов.

Тем не менее, при работе с универсальными типами стоит рассмотреть также понятие контрвариантности и инвариантности.

Контрвариантность – понятие прямо противоположное ковариантности, когда вы можете использовать менее производные типы вместо указанного базового.

Инвариантность – понятие, обозначающее ситуацию, в которой указанный тип должен точь-в-точь совпадать с передаваемым.

При работе с универсальными интерфейсами и универсальными делегатами Вы можете явно указывать вариантность при помощи ключевых слов **in** (контрвариантность) и **out** (ковариантность). Без указания по умолчанию будет использована инвариантность. Вариативным могут быть только ссылочные типы (недопустима и вариантность параметров с ref/in/out); также вариативность не применима к многоадресным делегатам, то есть вы не сможете объединить два различных типа.

Универсальные Интерфейсы

Как и в случае с классами, Вы можете объявлять универсальные интерфейсы. Мы уже встречались с универсальными интерфейсами на практике (IComparable<T>, IComparer<T>), они позволяют избегать упаковки-распаковки типов.

Аналогично классам универсальные интерфейсы могут использовать несколько типизирующих параметров, они используют такие же правила наследования, как и классы; у классов, интерфейсов и структур совпадают принципы определения перегрузок.

Однако есть и особенности – Вы можете указывать вариативность. Ковариантность позволяет методам иметь тип возвращаемого значения, степень наследования которого больше степени параметра указанного типа. Контрвариантность позволяет входным параметрам метода иметь меньшую степень наследования по сравнению с данным типизирующим параметром. Универсальный интерфейс, который имеет ковариантные или контрвариантные параметры называется **вариантным**. Вы можете одновременно добавить ковариантные и контрвариантные параметры в один интерфейс.

Объявление Ковариантных Интерфейсов

Для объявления интерфейса с ковариантным параметром с ключевым словом out Вы должны выполнить следующие условия:

- Тип используется только в качестве возвращаемого значения, а не параметра метода (иначе возникнет ошибка компиляции). Исключением является случай, когда в качестве параметра используется универсальный контрвариантный делегат, и сам тип является аргументом типа этого делегата.

- Тип не используется в качестве универсального ограничения типа для методов интерфейса

```
public interface ICovariant<out T>
{
    // OK - параметр - возвращаемое значение
    Ссылок: 0
    T DoSomeStuff();
    // Ошибка компиляции - ковариантный параметр типа T
    // не может быть использован в качестве параметра.
    Ссылок: 0
    void DoSomeWrongStuff(T RIP);
    // Исключение из правила, T - агрумент для
    // контрвариантного делегата Action<in T>.
    Ссылок: 0
    void DoSomeDelegateStuff(Action<T> del);
    // Ошибка компиляции - T не может быть ограничением типа.
    Ссылок: 0
    void WrongConstraint<U>(U obj) where U : T;
}
```

Объявление Контрвариантных Интерфейсов

Для объявления интерфейса с контрвариантным параметром с ключевым словом in Вы должны выполнить следующие условия:

- Тип используется только в качестве входного параметра метода, он никогда не может быть типом возвращаемого значения. В дополнение контрвариантные параметры могут быть использованы в качестве ограничения типа.

```
public interface IContravariant<in T>
{
    // Типичный случай - T - входной параметр
    Ссылок: 0
    void DoSomeStuff(T target);
    // T может использоваться в качестве ограничения параметра типа
    Ссылок: 0
    void DoSomeLimitedStuff<U>(U target) where U : T;
    // Ошибка компиляции - контравариантные параметры не могут
    // быть типами возвращаемого значения.
    Ссылок: 0
    T CannotReturnContravariant(T target);
}
```

Шаблоны C++ vs. Универсальные Шаблоны C#

Универсальные шаблоны в C# и шаблоны в C++ на первый взгляд могут показаться схожими: и те, и другие нужны для поддержки параметризации типов. Однако между ними есть фундаментальные различия: замена типизирующих параметров в C# осуществляется во время выполнения – для создаваемых объектах остаётся информация об универсальном типе.

Список основных отличий от C++:

- В универсальных шаблонах нельзя вызывать арифметические операторы, разрешены только операторы, определённые пользователем.
- Параметрами шаблонов могут быть только типы.
- Явная специализация не поддерживается – нельзя ограничить шаблоны конкретным типом.
- Частная специализация не поддерживается – нельзя определить различные реализации для наследников переданного типа.
- Типизирующие параметры не могут использоваться в качестве базовых (родителей) для других типов.
- Для типизирующих параметров нельзя указать тип по умолчанию.
- Параметры универсальных типов не могут сами по себе быть универсальными, однако сконструированные типы могут использоваться в качестве универсальных (пример – List<List<int>>).
- C# требует, чтобы код, написанный с использованием универсальных типов выполнялся для всех подходящих под ограничение типов.

ЗАДАЧИ

Задание 1

Обобщено может быть (*укажите все верные ответы*)**:**

1) Приватное поле.
2) Статический метод.
3) Запечатанный класс.
4) Абстрактное свойство.
5) Виртуальный индексатор.

Задание 2

Может быть обобщённым (*укажите все верные ответы*)**:**

1) Статическое поле.
2) Виртуальный метод.
3) Абстрактный класс.
4) Публичное свойство.
5) Защищённый индексатор.

Задание 3

Обобщения нужны для (*укажите все верные ответы*)**:**

1) Ускорения работы программы за счёт сокращения времени на упаковку и распаковку.
2) Возможности использования одних и тех же функциональных членов для различных типов данных.
3) Сокращения используемой памяти.
4) Реализации обобщённой парадигмы программирования.
5) Соблюдения стиля кода в программировании.

Задание 4

Выберите верные утверждения (*укажите все верные ответы*)**:**

1) Классы могут иметь несколько параметров универсального типа.
2) При унаследовании обобщённого класса, параметры универсального типа должны быть определены при объявлении производного класса.
3) Все функциональные члены могут быть обобщёнными.
4) В обобщённом классе все методы должны быть обобщёнными.
5) Обобщённые члены могут быть статическими.

Задание 5

В результате выполнения фрагмента программы:

```
using System;

class MyClass<T> {
    public T x;
    public T y;
}

class MyClass2<Y> : MyClass<Y> {
    public Y x;
    public Y y;
}

class Program {
    static void Main() {
        MyClass<int> obj = new MyClass2<double>();
        obj.y = 5.5;
        obj.x = obj.y + 10;
        Console.Write(obj.y + obj.x);
    }
}
```

на экран будет выведено:

Примечание:
Если возникнет ошибка компиляции, введите: ***
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---
Если возникнет ошибка исполнения или исключение, введите: +++

Задачи

Задание 6

В результате выполнения фрагмента программы:

```
using System;

class MyClass<T> {
    public static T x;
    public T y;
}

class Program {
    static void Main() {
        MyClass<int> obj = new MyClass<int>();
        MyClass<double>.x = 7.5;
        MyClass<int>.x = 5;
        obj.y = 7;
        Console.Write((int)MyClass<short>.x + MyClass<int>.x +
obj.y);
    }
}
```

на экран будет выведено:

Примечание:
Если возникнет ошибка компиляции, введите: ***
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---
Если возникнет ошибка исполнения или исключение, введите: +++

Задание 7

В качестве ограничений для параметров универсального типа можно использовать (*укажите все верные ответы*):

1) Класс.
2) Структуру.
3) Интерфейс.
4) Делегат.
5) Пространство имён.

Задание 8

В качестве ограничений для параметров универсального типа можно использовать следующие ключевые слова (*укажите все верные ответы*):

1) `class;`
2) `interface;`
3) `struct;`
4) `new;`
5) `static;`

Задание 9

Выберите верные объявления ограничений для параметров универсального типа (*укажите все верные ответы*):

1) `where T : class, struct, new();`
2) `where T : struct where X : class;`
3) `where T : new(), IComparable;`
4) `where T : ICloneable, class;`
5) `where T : class, ISerializable, new();`

ЗАДАЧИ

Задачи

Задание 10

Выберите верные утверждения (*укажите все верные ответы*)**:**

1) По умолчанию все универсальные типы – инвариантны.
2) Все типы, которые могут быть обобщёнными, могут иметь универсальные типы с поддержкой ковариантности и контравариантности.
3) IComparable имеет обобщённую реализацию с ковариативным универсальным параметром.
4) Контравариатным НЕ может быть тип возвращаемого значения.
5) Ковариантным может быть тип входного параметра.

Задание 11

В результате выполнения фрагмента программы:

```
using System;

class MyClass<T> {
    public static T x;
    public T y;

    public MyClass(T z, T y) {
        x = z;
        this.y = y;
    }
}


class Program {
    static void Main() {
        MyClass<int> obj = new MyClass<int>(30, 60);
        MyClass<int>.x = 10;
        long.Parse(MyClass<int>.x.ToString());
        Console.Write(obj.y + MyClass<int>.x);
    }
}
```

на экран будет выведено:

Примечание:
*Если возникнет ошибка компиляции, введите: ****
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---
Если возникнет ошибка исполнения или исключение, введите: +++

Ответы

Задание	Ответ
1	**12345**
2	**12345**
3	**1234**
4	**125** *(деструкторы нельзя обобщить)*
5	*******
6	**12**
7	**1234** *(C# 7.3 допускает ограничение where T : Delegate)*
8	**134**
9	**25**
10	**14**
11	**70**