Generic

- Tuple C# 7.0
- Deconstructors

Geeneric

Позволяют многократно использовать алгоритмы, создавая типизированные параметры.
MSDN

- Можно сделать Generic Class, method, delegate, interface

Common Class:

```
class Element
{
    public object Key { get; set; }
    public int Value { get; set; }
}

Element element = new Element { Value = 32, Key = 2 };
Element another = new Element { Value = 33, Key = "4356" };
int key = (int)element.Key;
string key2 = (string)another.Key;
```

Generic Class:

- T - type parameters
- Стандартный codestyle: использовать в этом качестве имена, начинающиеся с T: T, TKey, TValue, TResult или однобуквенные K, V

```
class Element<T>
{
    public T Key { get; set; }
    public int Value { get; set; }
}

// Boxing/Unboxing не происходит
Element<int> element = new Element<int> { Value = 32, Key = 2 };
Element<string> another = new Element<string> { Value = 33, Key = "4356" };
int key = element.Id;
string key2 = another.Id;
```

Если надо задать несколько типов:

```
class Element<TKey, TValue>
{
    public TKey Key { get; set; }
    public TValue Value { get; set; }
}
```

Generic methods

Если хочется обобщить только метод:

```
static void Swap<T>(ref T left, ref T right)
{
    T temp;
    temp = left;
    left = right;
    right = temp;
}
```

```
// Можно перегружать
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

Пример с обобщенной коллекцией List<T>:

```
[Serializable]
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList,
ICollection, IEnumerable
{
    public List();
    public void Add(T item);
    public Int32 BinarySearch(T item);
    public void Clear();
    public Boolean Contains(T item);
    public Int32 IndexOf(T item);
    public Boolean Remove(T item);
    public void Sort();
    public void Sort(IComparer<T> comparer);
    public void Sort(Comparison<T> comparison);
    public T[] ToArray();
    public Int32 Count { get; }
    public T this[Int32 index] { get; set; }
}
```

Плюсы Generic:

- быстродействие
- типизация
- простота кода
- не надо быть в курсе алгоритма, чтобы его использовать

Open / Closed constructed types

- MSDN, SOF Jon Skeet Answer, SOF Another Answer
- Типа разделяют все generic типы на открытые, которые оперируют неизвестным типом, и закрытые (все остальные)

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

Еще пример:

```
class BaseNode { }
class BaseNodeGeneric<T> { }

class NodeConcrete<T> : BaseNode { }            // concrete type

class NodeClosed<T> : BaseNodeGeneric<int> { }  // closed constructed type

class NodeOpen<T> : BaseNodeGeneric<T> { }      // open constructed type
```

- Open constructed types - нельзя создать экземпляр объекта

```
using System;
using System.Collections.Generic;

Object o = Activator.CreateInstance(typeof(List<>));
// Exception: Cannot create an instance of ... because
Type.ContainsGenericParameters is true.

o = Activator.CreateInstance(typeof(List<string>)); // Создастся
```

- Разделение в целом умозрительное и на практике не используемое

- CLR на каждый тип данных создает объект (type objects)
- Для каждого closed constructed type создается отдельный объект, что увеличивает рабочий размер приложения
- Статические поля для каждого такого объекта будут разными
- Статический конструктор (он же без параметров) будет вызван один на всех
- Для всех ссылочных таких типов CLR компилирует один код, что ускоряет ситуацию

```
internal sealed class GenericTypeThatRequiresAnEnum<T>
{
    static GenericTypeThatRequiresAnEnum()
    {
        if (!typeof(T).IsEnum)
        {
            throw new ArgumentException("T must be an enumerated type");
        }
    }
}
```

Обобщения при наследовании

Никак не препятствуют наследованию:

```
class BaseNodeGeneric<T> { }

class Node1 : BaseNodeGeneric<int> { }      //No error
class Node2<T> : BaseNodeGeneric<T> { }     //No error
class Node3<T> : BaseNodeGeneric<int> { }   //No error
class Node4<T, V> : BaseNodeGeneric<T> { }  //No error
class Node5 : BaseNodeGeneric<T> {}         //Generates an error
class Node6 : T {}                          //Generates an error
```

Для случая с двумя параметрами:

```
class BaseNodeMultiple<T, U> { }

class Node4<T> : BaseNodeMultiple<T, int> { }   //No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }  //No error
class Node6<T> : BaseNodeMultiple<T, U> {}      //Generates an error
```

Generic Interface

```
public interface IEnumerator<T> : IDisposable, IEnumerator
{
    T Current { get; }
}
```

```
public interface IList : ICollection, IEnumerable
```

Generic Delegate

```
public delegate TReturn CallMe<TReturn, TKey, TValue>(TKey key, TValue value);
```

Ограничения обобщений

- Ограничения помогают компилятору знать, что можно делать с объектами открытого типа
- Когда мы ничего не знаем об открытом типе, то generic не удобны - метод вызвать не можем, как создать элемент непонятно

```
public static T Min<T>(T o1, T o2) where T : IComparable<T>
{
    if (o1.CompareTo(o2) < 0)
        return o1;
    return o2;
}
```

Multiple constrains:

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new() { }
```

Какие бывают ограничения:

- Main constraint (может быть только одно)
  - where T : struct - T - значимый тип (кроме Nullable)
  - where T : class - любой ссылочный тип (class, interface, delegate, or array type).
  - where T : <base class name> - должен наследоваться (или являться) от базового класса. Нельзя указать System.Object, System.Array, System.Delegate, System.MulticastDelegate, System.ValueType, System.Enum и System.Void
- Secondary
  - where T : <interface name> - должен реализовывать указанный интерфейс
  - where T : U - ограничивает отношения между типами
- Constructor constraint (должен быть только один и быть последним)
  - where T : new() - должен иметь конструктор без параметров

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

```
internal sealed class ConstructorConstraint<T> where T : new()
{
    public static T Factory()
    {
        return new T();
    }
}
```

Работа с переменными обобщенного типа

- всегда можно конвертить к System.Object или явно к интерфейсу любому
- можно брать typeof и использовать reflection

Сравнение

```
private static void Comparing<T>(T o1, T o2)
{
    if (o1 == o2) { } // Ошибка
}
```

- != and == нельзя использовать, потому что компилятор не знает, поддерживает ли тип это или нет
- Вообще значимые типы можно сравнивать только после перегрузки оператора ==
- Если добавить ограничение class, то код будет работать
- Для значимых типов общего решения нет, вариантом может быть использование констрейнта интерфейса IEquatable<T> или реализация не обобщенных типов/методов

Сравнение с null

```
private static void ComparingWithNull<T>(T obj)
{
    if (obj == null)
    { /* Этот код никогда не исполняется для значимого типа */ }
}
```

- Сравнивать с null можно всегда
- для значимых типов всегда вернет false
- бессмысленно при ограничении 'struct', компилятор ругнется

default

- default используется для получения дефолтного значения в Generic типах/методах, поскольку null просто так получить не можем (для значимых)
- для ссылочных возвращает null
- для значимых 0 (инициализированное нулями значение)

```
class Element<T>
{
    T id = default(T);
}
```

```
private static void Example<T>()
{
    T temp = null; // Нельзя будет ошибка, потому что тип может быть значимым
    T temp = default(T);
}
```

Приведение переменной обобщенного типа

Так не надо:

```
static void Cast<T>(T obj)
{
    int x = (Int32) obj ;        // Ошибка
    string s = (String) obj;     // Ошибка
}
```

Надо так:

```
static void Cast<T>(T obj)
{
    int x = (Int32) (Object) obj ;       // Все хорошо
    string s = (String) (Object) obj;    // Все хорошо
    string s2 = obj as String;           // Все хорошо
}
```

Использование в качестве операндов

Всё плохо

```
static T Sum<T>(T num) where T : struct
{
    T sum = default(T) ;
    for (T n = default(T); n < num ; n++)
    {
        sum += n;
    }
    return sum;
}
// error CS0019: Operator '<' cannot be applied to operands
// error CS0023: Operator '++' cannot be applied to operand of type 'T'
// error CS0019: Operator '+=' cannot be applied to operands of type 'T' and 'T'
```

Рекомендации

Общие рекомендации по generic:

- Всегда используйте generic версии вместо object / dynamic!
- Ограничивайте констрейнтами!
- Для упрощения кода
    - никогда не делайте так: class DateTimeList : List<DateTime> {}
    - можно так: using DateTimeList = System.Collections.Generic.List<System.DateTime>;

Можно отметить:

- Нельзя сделать generic свойства, индексаторы, события, операторные методы, конструкторы, деструкторы (И вообщем это не нужно)
- Нельзя делать кастомных ограничений на конструкторы, только конструктор по-умолчанию
- Слабая поддержка операндов
- Нет! блин! ограничения на Enum / Delegate SOF Eric Lippert + Jon Skeet Answers, SOF Another, EnumNet

Пример того, как вообще можно, забавный хак:

```
public abstract class AbstractEnumHelper<TClass> where TClass : class
{
    public static TStruct Parse<TStruct>(string value) where TStruct : struct,
TClass
    {
        return (TStruct) Enum.Parse(typeof(TStruct), value);
    }
}

public class EnumHelper : AbstractEnumHelper<Enum> {}

//usage:
EnumHelper.Parse<MyEnum>("value");
```

Более простой вариант:

```
public T Parse<T>(string value) where T : struct, IComparable, IFormattable,
IConvertible
{
   if (!typeof(T).IsEnum)
   {
      throw new ArgumentException("T must be an enumerated type");
   }

   return (T) Enum.Parse(typeof(T), value);
}
```

Ковариантность и контрвариантность в интерфейсах

Параметры типы в обобщенных интерфейсах могут быть инвариантными, ковариантными, контрвариантными. MSDN, SOF Eric Lippert Answet
По умолчанию параметр тип инвариантен - его тип не может изменяться.

- Ковариантность - аргумент тип можно преобразовать к одному из его базовых классов (out)
- Контравариантность - можно преобразовать к производному от него (in)

```
// Covariance
IEnumerable<string> strings = new List<string>();
IEnumerable<object> objects = strings;

// Contravariance
Action<object> actObject = SetObject; // static void SetObject(object o) { }
Action<string> actString = actObject;

Int32 Count(IEnumerable<Object> collection) { ... }
Int32 c = Count(new[] { "Grant" });
```

Вариативность в стандартных интерфейсах:

- IEnumerable<T> - T является ковариантным
- IEnumerator<T> - T является ковариантным
- IQueryable<T> - T является ковариантным
- IGrouping<TKey,TElement> - TKey и TElement являются ковариантными
- IComparer<T> - T является контравариантным
- IEqualityComparer<T> - T является контравариантным
- IComparable<T> - T является контравариантным

Мне правда кажется, что запомнить это нереально и в практике никто не помнит, какая там вариативность у часто используемых интерфейсов

```
class BaseClass { }
class DerivedClass : BaseClass { }

class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

- Параметры ref и out не могут быть вариантными
- значимые типы не поддерживают вариативность

```
IEnumerable<DateTime> dts = new List<DateTime>();
IEnumerable<object> objects = dts; // Нельзя! Со значимыми не работает
```

- Классы в любом случае инвариативны, даже при вариативном интерфейсе

```
List<Object> list = new List<String>();                 // Так нельзя
IEnumerable<Object> listObjects = new List<String>();   // Так можно
```

- Ковариантность - обозначается ключевым словом out
  - должен возвращать этот параметр
  - не быть параметром методов или ограничителем

```
interface ICovariant<out R>
{
    R GetSomething();                   // Так можно
    void SetSometing(R sampleArg);      // В качестве параметра нельзя!
    void DoSomething<T>() where T : R;  // В качестве ограничительно тоже нельзя!
}
```

- контрвариантность - обозначается ключевым словом in
  - можно использовать только в качестве аргументов метода
  - можно использовать для универсальных ограничений

```
interface IContravariant<in A>
{
    void SetSomething(A sampleArg);     // Можно
    void DoSomething<T>() where T : A;  // Можно
    A GetSomething();                   // Нельзя!
}

interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSometings(A sampleArg);
}
```

Реализация ковариативности:

```
interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        return default(R);
    }
}

ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

SampleImplementation<Button> button = new SampleImplementation<Button>();
SampleImplementation<Object> obj = button; // Так нельзя!
```

При расширении интерфейсов надо явно задавать in/out, чтобы тип параметр не был инвариантным

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

```
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

class Pets : IEnumerable<Cat>, IEnumerable<Dog> // IEnumerable<out T> is covariant
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        return null;
    }
    IEnumerator IEnumerable.GetEnumerator() { return null; }
    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
        return null;
    }
}

IEnumerable<Animal> pets = new Pets();
pets.GetEnumerator(); // !! LUL WTF !! Ambiguity
```

Tuple

Есть два вида кортежей:

1. Класс System.Tuple
2. Класс System.ValueTuple - кортежи C# 7.0

Класс System.Tuple

- Статический класс для генерации конкретных Tuple<T1>, Tuple<T1, T2>
- System.Tuple - ссылочные типы
- У всех конкреных классов кортежа переопределены методы Equals, GetHashCode
- Immutable - все элементы Readonly

```
var test = new Tuple<int, int>(3, 33);
Console.WriteLine("{0}, {1}", test.Item1, test.Item2);
```

```
var test = new Tuple.Create(3, 33);
Console.WriteLine("{0}, {1}", test.Item1, test.Item2);
```

```
Create<T1>(T1)
Create<T1, T2>(T1, T2)
Create<T1, T2, T3>(T1, T2, T3)
Create<T1, T2, T3, T4>(T1, T2, T3, T4)
Create<T1, T2, T3, T4, T5>(T1, T2, T3, T4, T5)
Create<T1, T2, T3, T4, T5, T6>(T1, T2, T3, T4, T5, T6)
Create<T1, T2, T3, T4, T5, T6, T7>(T1, T2, T3, T4, T5, T6, T7)
```

Tuple C# 7.0

- Появились из этого предложения
- Компилируются в System.ValueTuple
- Значимый тип, причем mutable MSDN Blog, SOF

```
var tuple = (5, 10);
Console.WriteLine(tuple.Item1); // 5
Console.WriteLine(tuple.Item2); // 10

(int, int) pair = (5, 10);
(string, int, double) trinity = ("Tom", 25, 81.23);
```

Examples:

```
var tuple = (count:5, sum:10);
Console.WriteLine(tuple.count); // 5
Console.WriteLine(tuple.sum); // 10
```

```
var (name, age) = ("Tom", 23);
Console.WriteLine(name);    // Tom
Console.WriteLine(age);     // 23
```

```
static void Main(string[] args)
{
    (int sum, int count) tuple = GetNamedValues(Enumerable.Range(0, 10));
    Console.WriteLine(tuple.count);
    Console.WriteLine(tuple.sum);
}
private static (int sum, int count) GetNamedValues(int[] numbers)
{
    var result = (sum:0, count: 0);
    foreach (var value in numbers)
    {
        result.sum += value;
        result.count++;
    }
    return result;
}
```

ValueTuple limitations

```
var person = (Name: "John", Last: "Smith");
var result = JsonConvert.SerializeObject(person);

Console.WriteLine(result);
// {"Item1":"John","Item2":"Smith"}
```

- no reflection
- no dynamic access to named elements
- no razor usage

Deconstructors

```
public class Person
{
    public string Name => "John Smith";
    public int Age => 43;
    public void Deconstruct(out string name, out int age)
    {
        name = Name;
        age = Age;
    }
}

var person = new Person();
var (name, age) = person;
Console.WriteLine(name);    // John Smith
```