

Обобщения

Обобщения (generics) появились в C# 2.0.

Обобщенные методы

Как описать универсальный метод обмена содержимым?

```
static void Swap<T>(ref T a, ref T b)
{
    T v = a; a = b; b = v;
}
```

Параметр T — *обобщенный параметр*. Вызов:

```
Swap<string>(ref s1, ref s2);
```

Логическое выведение типа (type inference):

```
Swap(ref s1, ref s2);
```

Наряду с обобщенными вариантами метода можно описывать и его необобщенные варианты, которые имеют приоритет:

```
static void Swap(ref string a, ref string b)
```

```
{
    string v = a; a = b; b = v;
    Console.WriteLine("Non-generic");
}
```

...

```
Swap(ref s1, ref s2); // "Non-generic"
```

Можно явно выбрать обобщенный вариант:

```
Swap<string>(ref s1, ref s2);
```

Метод может иметь несколько обобщенных параметров:

```
public T3 Test<T1,T2,T3>(T1 a, T2 b) {...}
```

Обобщенные типы

Недостатки коллекций с элементами типа object:

```
Stack s = new Stack();
s.Push(14);
int a = s.Pop(); // ошибка компиляции
```

Исправленный вариант:

```
s.Push(14); // упаковка числа 14
int a = (int)s.Pop(); // распаковка
```

Попытка занесения в стек данных ошибочного типа:

```
s.Push("14"); // в стек будет помещена строка
int a = (int)s.Pop(); // ошибка времени выполнения
```

Все отмеченные проблемы снимаются при использовании обобщенного класса Stack<T>:

```
Stack<int> s = new Stack<int>();
s.Push(14); // упаковки не производится
int a = s.Pop(); // ошибки компиляции нет
s.Push("14"); // ошибка компиляции
```

Стандартные коллекции

В пространстве имен System.Collections.Generic определен ряд обобщенных классов-коллекций.

Приведем таблицу, в которой указаны соответствия для «старых» (необобщенных) и «новых» (обобщенных) коллекций, а также даны их краткие описания. Во всех необобщенных коллекциях элементы хранятся как данные типа object; в обобщенных коллекциях тип данных определяется обобщенным параметром (или двумя обобщенными параметрами в случае обобщенных ассоциативных массивов).

Необобщенные коллекции	Обобщенные коллекции	Описание
ArrayList	List<T>	Динамический массив
Stack	Stack<T>	Стек
Queue	Queue<T>	Очередь
Hashtable	Dictionary<TKey, TValue>	Ассоциативный массив (набор пар «ключ-значение»)

При переборе элементов класса Dictionary<TKey, TValue> используется структура KeyValuePair<TKey, TValue>. Не-обобщенный аналог — структура DictionaryEntry.

Пример: подсчитать количество появлений одинаковых значений в целочисленном массиве data:

```
int[] data = new int[] { 3, 6, 6, 2, 3, 4, 6, 7, 2,
    3, 4, 3, 2, 5, 4, 3, 2, 45 };
Dictionary<int, int> dict = new Dictionary<int, int>();
foreach (int d in data)
    if (dict.ContainsKey(d))
        dict[d]++;
    else
        dict[d] = 1;
foreach (KeyValuePair<int, int> kv in dict)
    Console.WriteLine(kv.Key + "-" + kv.Value + " ");
// 3-5 6-3 2-4 4-3 7-1 5-1 45-1
```

Обобщенное множество HashSet<T> не имеет необобщенного аналога. Данная коллекция не может содержать одинаковые элементы, к элементам нельзя обратиться по индексу. С помощью метода s.Contains(item) можно быстро определить, входит ли элемент item в коллекцию s. Кроме того, в данном классе определены все стандартные операции над множествами; можно также организовать перебор всех элементов множества, используя цикл foreach.

Во всех рассмотренных классах-коллекциях можно определить количество элементов (свойство Count, только для чтения). Во всех коллекциях, кроме Dictionary, есть метод CopyTo для копирования элементов в массив.

Реализация обобщенных типов

Пример (упрощенная реализация стека):

```
public class SimpleStack<T>
{
    int top;
    T[] data = new T[100];
    public void Push(T d) { data[top++] = d; }
    public T Pop() { return data[--top]; }
    public T Peek() { return data[top]; }
    public int Count { get { return top; } }
}
```

Обобщенный тип, в имени которого хотя бы один обобщенный параметр является «заглушкой», называется *открытым* типом. Если все обобщенные параметры заменены на конкретные типы, то такой тип называется *замкнутым*. Для создания экземпляра можно использовать только замкнутые типы.

```
Type t1 = typeof(SimpleStack<>);
// t1 содержит информацию об открытом типе
Type t2 = typeof(SimpleStack<int>);
// t2 содержит информацию о замкнутом типе
Console.WriteLine(t1.Name + " " +
    t1.ContainsGenericParameters);
foreach (Type t in t1.GetGenericArguments())
    Console.WriteLine(" " + t.Name);
Console.WriteLine("\n" + t2.Name + " " +
    t2.ContainsGenericParameters);
foreach (Type t in t2.GetGenericArguments())
    Console.WriteLine(" " + t.Name);
```

Результат:

```
SimpleStack`1 True T
SimpleStack`1 False Int32
```

Еще пример:

```
class Test<T> {}
class Test<T1,T2> {}
```

```

...
Type t1 = typeof(Test<>), t2 = typeof(Test<, >);
Console.WriteLine(t1.Name);
foreach (Type t in t1.GetGenericArguments())
    Console.WriteLine(" " + t.Name);
Console.WriteLine("\n" + t2.Name);
foreach (Type t in t2.GetGenericArguments())
    Console.WriteLine(" " + t.Name);

```

Результат:

```
Test`1 T
```

```
Test`2 T1 T2
```

Значение по умолчанию для обобщенного параметра T: default(T) (null для всех ссылочных типов и структура с побитово обнуленными полями для размерных типов).

При определении классов-потомков обобщенных классов можно использовать как открытые, так и замкнутые варианты:

```

class Test1<T>: Test<T> {}
class Test2: Test<int> {}
class Test3<T1, T2>: Test<T1> {}
class Test4: Test<Test4> {}

```

Если в обобщенном типе определены статические поля, то их значения будут *уникальными* для любого замыкания этого типа (статическое поле f типа Test<int> не будет иметь никакого отношения к статическому полю f типа Test<string>).

Можно определять краткие *псевдонимы* (синонимы) типов с помощью директивы using:

```

using Graph = System.Collections.Generic.List
    <System.Collections.Generic.HashSet<int>>;

```

В правой части данного определения необходимо указывать *полные* имена всех типов. Теперь в программе имя Graph будет считаться синонимом типа, указанного справа (подобно тому как имя int считается синонимом типа System.Int32).

Ограничения для обобщенных параметров

Следующие обобщенные методы не будут компилироваться:

```

public static void M1<T>(ref T a)
{
    a = null; // если T – структура,
    // то переменной a нельзя присваивать null
}
public static void M2<T>(out T a)
{
    a = new T(); // тип T не обязан
    // иметь конструктор без параметров
}
public static T? M3<T>(ref T a, bool b)
    // тип T? определен только в случае,
    // если T является структурой
{
    T? res = null;
    if (b)
        res = default(T);
    return res;
}
public static T M4<T>(T a, T b)
{
    return a.CompareTo(b) < 0 ? a : b;
    // тип T не обязан реализовывать
    // интерфейс IComparable
}

```

К обобщенному параметру T можно применить одно или несколько *ограничений* (constraints):

- where T : class — тип T должен быть ссылочным;

- where T : struct — тип T должен быть размерным;
- where T : *имя_класса* — тип T должен либо быть указанным классом, либо порождаться от него (нельзя указывать Array, Delegate, MulticastDelegate, ValueType, Enum);
- where T : new() — тип T должен иметь открытый конструктор без параметров;
- where T : *имя_интерфейса* — тип T должен реализовывать указанный интерфейс;
- where T : *имя_другого_обобщенного_параметра* — тип T должен быть совместим с указанным обобщенным параметром (т. е. либо совпадать с ним, либо порождаться от указанного параметра, либо реализовывать его — в случае, если другой обобщенный параметр является интерфейсом).

С применением ограничений нам удастся обеспечить компиляцию всех приведенных выше методов:

```

public static void M1<T>(ref T a) where T : class
{ a = null; }
public static void M2<T>(out T a) where T : new()
{ a = new T(); }
public static T? M3<T>(ref T a, bool b)
    where T : struct
{
    T? res = null;
    if (b)
        res = default(T);
    return res;
}
public static T M4<T>(T a, T b)
    where T : IComparable
{ return a.CompareTo(b) < 0 ? a : b; }

```

Для любого параметра T может быть определено не более одного *основного* ограничения и произвольное число *дополнительных*. Основными являются первые три из перечисленных видов ограничений, дополнительными — последние три.

Замечание. Приведем симметричный вариант метода M4:

```

public T M4<T>(T a, T b) where T : IComparable
{
    try
    { return a.CompareTo(b) < 0 ? a : b; }
    catch
    { return default(T); }
}

```

Методы и типы являются единственными конструкциями языка C#, которые могут иметь обобщенные параметры. Однако любые члены могут *использовать* обобщенные параметры их типа. Пример для индексатора класса SimpleStack<T>:

```

public T this[int index]
{ get { return data[index]; } }

```

Обобщенные делегаты

При определении делегата можно указывать обобщенные параметры. В результате получим *обобщенного делегата*. Достоинство обобщенных делегатов: при их использовании можно обойтись без определения новых делегатов.

Пример. Мы рассматривали делегат, позволяющий хранить только методы, преобразующие целые числа:

```
public delegate int Transformer(int i);
```

Если же определить обобщенный делегат Transformer<T>, то с его помощью можно будет работать с методами, преобразующими данные любого типа:

```
public delegate T Transformer<T>(T x);
```

```
...
```

```
Transformer<int> t1 = i => -i;
```

```
Transformer<string> t2 = s => '*' + s + '*';
```

```
Console.WriteLine(t1(1)); // -1
Console.WriteLine(t2("abc")); // *abc*
```

В пространстве имен System определено семейство обобщенных делегатов, с помощью которых можно сконструировать замкнутый делегат для хранения практически любого метода (не использующего ref- и out-параметров):

```
public delegate TResult Func<TResult>();
public delegate TResult Func<T, TResult>(T arg);
public delegate TResult Func<T1, T2, TResult>
    (T1 arg1, T2 arg2);
public delegate TResult Func<T1, T2, T3, TResult>
    (T1 arg1, T2 arg2, T3 arg3);
public delegate TResult Func<T1, T2, T3, T4,
    TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4);
public delegate void Action();
public delegate void Action<T>(T arg);
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);
public delegate void Action<T1, T2, T3>(T1 arg1,
    T2 arg2, T3 arg3);
public delegate void Action<T1, T2, T3, T4>(T1 arg1,
    T2 arg2, T3 arg3, T4 arg4);
```

Вместо необобщенного делегата Transformer можно использовать делегат Func<int, int>, вместо Transformer<string> можно использовать Func<string, string>, а вместо делегата InfoEventHandler — Action<object, InfoEventArgs>.

Обобщенные интерфейсы

Рассмотрим уже известный нам интерфейс IComparable:

```
public interface IComparable
{ int CompareTo(object other); }
```

Две проблемы: 1) компилятор не будет считать ошибкой передачу в качестве other объекта любого типа, однако при выполнении программы будет возбуждено исключение, если параметр other имеет тип, не приводимый к типу объекта, вызвавшего метод CompareTo; 2) при выполнении данного метода для размерных типов параметр other будет упаковываться.

Средство, позволяющее повысить как надежность, так и эффективность метода CompareTo, — *обобщенный интерфейс*:

```
public interface IComparable<T>
{ int CompareTo(T other); }
```

Для целей обратной совместимости имеет смысл реализовать для типа *оба* указанных интерфейса: и необобщенный, и обобщенный. При этом нет необходимости в явной реализации какого-либо из методов данных интерфейсов, поскольку методы не будут конфликтовать: они просто будут считаться перегруженными вариантами одного метода CompareTo.

Вернемся к методу M4 с ограничением и изменим его:

```
public static T M4a<T>(T a, T b)
    where T : IComparable<T>
{ return a.CompareTo(b) < 0 ? a : b; }
```

Это повысит эффективность метода при его использовании для размерных типов (не будет упаковки параметра b).

Другие примеры стандартных обобщенных интерфейсов:

```
public interface IComparer<T>
{ int Compare(T a, T b); }
public interface IEqualityComparer<T>
{
    bool Equals(T a, T b);
    int GetHashCode(T obj);
}
```

Вспомним интерфейс IEnumerator:

```
public interface IEnumerator
{
```

```
bool MoveNext();
object Current { get; }
void Reset();
}
```

Было бы удобнее, если бы свойство Current возвращало значение того же типа, что и перебираемая перечислителем последовательность:

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

Это пример *порождения* («наследования») одного интерфейса от других. Порожденный интерфейс-потомок «наследует» все методы своих интерфейсов-предков, поэтому тот класс, который реализует подобный интерфейс-потомок, должен реализовать не только собственные методы потомка, но и все методы предков. Подключение к классу интерфейса-потомка означает автоматическое подключение к нему всех интерфейсов-предков (поэтому перечислять их в списке реализуемых интерфейсов не требуется, хотя и не запрещается).

Таким образом, в любом классе, реализующем интерфейс IEnumerator<T>, следует реализовать не только его свойство Current (типа T), но и свойство Current типа object, методы MoveNext и Reset, а также метод Dispose. В результате этот класс будет автоматически реализовывать не только интерфейс IEnumerator<T>, но также и интерфейсы IEnumerator и IDisposable. При этом все требуемые методы не удастся описать неявно. Разумнее всего явно реализовать «менее удобное» свойство Current, возвращающее значение типа object:

```
class EnumDemo : IEnumerator<EnumDemo>
{
    public EnumDemo Current
    {
        get { ... }
    }
    object IEnumerator.Current
    {
        get { return Current; } // возвращается
            // значение неявно реализованного свойства
    }
    public void Dispose() { ... }
    public bool MoveNext() { ... }
    public void Reset() { ... }
}
```

Для объекта eDemo типа EnumDemo операции eDemo is IEnumerator, eDemo is IEnumerator<EnumDemo> и eDemo is IDisposable будут возвращать true.

Обобщенный интерфейс IEnumerableable<T> позволяет получить *обобщенный перечислитель* типа IEnumerator<T>:

```
public interface IEnumerableable<T> : IEnumerable
{ IEnumerator<T> GetEnumerator(); }
```

Этот интерфейс, как и все базовые обобщенные интерфейсы, связанные с коллекциями, описан в пространстве имен System.Collections.Generic.

Важнейшая особенность интерфейса IEnumerableable<T> (по сравнению с его необобщенным аналогом) состоит в обеспечении большей надежности программного кода. Пример:

```
class EnumDemo : IEnumerableable
{
    public IEnumerator GetEnumerator()
    { yield return 1; yield return 2; }
}
...
```

```
EnumDemo en = new EnumDemo1();
foreach (string a in en)
    Console.WriteLine(a); // ошибка времени выполнения
Изменим наш класс, реализовав в нем обобщенный интерфейс IEnumerable<T>:
class EnumDemo : IEnumerable<int>
{
    public IEnumerator<int> GetEnumerator()
    { yield return 1; yield return 2; }
    IEnumerator IEnumerable.GetEnumerator()
    { return GetEnumerator(); }
}
Теперь ошибочная попытка организовать перебор элементов типа string будет выявлена уже на этапе компиляции.
```

Обобщенные интерфейсы для коллекций

Рассмотренные выше интерфейсы IEnumerable и IEnumerable<T> предназначены для организации *перебора* элементов коллекций; в частности, для обеспечения *унификации* этого перебора посредством цикла foreach. Для унификации других методов и свойств, связанных с коллекциями, в стандартной библиотеке .NET Framework имеются дополнительные интерфейсы, как необобщенные, так и обобщенные. Основными из них являются ICollection и ICollection<T>, IList и IList<T>, IDictionary и IDictionary<TKey, TValue>.

Необобщенные и обобщенные варианты интерфейсов ICollection и IList различаются между собой гораздо существеннее, чем ранее рассмотренные необобщенные и обобщенные варианты интерфейсов Comparable, Enumerator и Enumerable. По-видимому, это объясняется недостаточной продуманностью более «старых», необобщенных интерфейсов, из-за которой разработчики их обобщенных вариантов были вынуждены внести в них существенные модификации. Часть функциональности, связанной с необобщенным интерфейсом IList, при реализации обобщенных интерфейсов была перенесена в интерфейс ICollection<T>, после чего эта часть была просто унаследована интерфейсом IList<T>. В результате интерфейс ICollection<T> оказался «богаче», чем интерфейс ICollection, однако при этом интерфейс IList<T>, с учетом членов, унаследованных от ICollection<T>, получил практически ту же функциональность, что и интерфейс IList. Следует отметить, что «расширение» интерфейса ICollection<T> не отразилось на большинстве стандартных обобщенных классов-коллекций, поскольку они либо, подобно своим необобщенным аналогам, реализуют лишь необобщенный интерфейс ICollection (таковы классы Stack<T> и Queue<T>), либо реализуют более полный интерфейс IList<T> (класс List<T>) или IDictionary<TKey, TValue> (класс Dictionary<TKey, TValue>). Из всех рассмотренных нами стандартных обобщенных коллекций только класс HashSet<T>, появившийся в .NET Framework 3.5, служит некоторым «оправданием» факта добавления в интерфейс ICollection<T> новых членов, поскольку этот класс реализует интерфейс ICollection<T> и *не реализует* интерфейс IList<T>.

Приведем наиболее важные члены интерфейсов коллекций.

<p>ICollection (представители — Stack, Queue, Stack<T>, Queue<T>):</p> <pre>int Count { get; } void CopyTo(Array array, int index);</pre>	<p>ICollection<T> (представитель — HashSet<T>):</p> <pre>int Count { get; } void CopyTo(T[] array, int index); void Clear(); bool Contains(T item); void Add(T item);</pre>
--	--

```
| bool Remove(T item);
```

В классах Stack<T>, Queue<T> реализован более безопасный с точки зрения типов метод void CopyTo(T[] array, int index), а реализация одноименного метода интерфейса ICollection скрыта. Кроме того, методы Clear и Contains также реализованы в классах Stack, Queue, Stack<T>, Queue<T> (хотя в интерфейсе ICollection они отсутствуют).

<p>IList (представитель — ArrayList):</p> <pre>object this[int index] { get; set; } int IndexOf(object value); void Insert(int index, object value); void RemoveAt(int index); void Clear(); bool Contains(object value); int Add(object value); bool Remove(object value);</pre>	<p>IList<T> (представитель — List<T>):</p> <pre>T this[int index] { get; set; } int IndexOf(T item); void Insert(int index, T item); void RemoveAt(int index);</pre>
--	---

Интерфейс IList<T>, как и IList, фактически включает методы Clear, Contains, Add и Remove, поскольку интерфейс IList<T> порождается от интерфейса ICollection<T> (при этом метод Add, в отличие от одноименного метода интерфейса IList, имеет возвращаемый тип void). Так как интерфейс IList порождается от интерфейса ICollection, оба интерфейса — и IList, и IList<T> — также включают свойство Count и метод CopyTo.

<p>IDictionary (представитель — Hashtable):</p> <pre>object this[object key] { get; set; } ICollection Keys { get; } ICollection Values { get; } void Add(object key, object value); void Remove(object key); bool Contains(object key); void Clear();</pre>	<p>IDictionary<TKey, TValue> (представитель — Dictionary<TKey, TValue>):</p> <pre>TValue this[TKey key] { get; set; } ICollection<TKey> Keys { get; } ICollection<TValue> Values { get; } void Add(TKey key, TValue value); bool Remove(TKey key); bool ContainsKey(TKey key); bool TryGetValue(TKey key, out TValue value);</pre>
---	---

Интерфейс IDictionary порождается от интерфейса ICollection, а интерфейс IDictionary<TKey, TValue> — от интерфейса ICollection<KeyValuePair<TKey, TValue>>. Поэтому и IDictionary, и IDictionary<TKey, TValue> содержат также свойство Count и метод CopyTo (а интерфейс IDictionary<TKey, TValue> — метод Clear). В классе Hashtable метод CopyTo реализован в виде экземплярного метода класса и поэтому может быть вызван непосредственно для объекта Hashtable, тогда как в классе Dictionary<TKey, TValue> метод CopyTo реализован лишь как *метод интерфейса*, поэтому для его вызова необходимо привести объект Dictionary<TKey, TValue> к типу ICollection<KeyValuePair<TKey, TValue>>. Метод CopyTo класса Hashtable возвращает массив объектов типа DictionaryEntry, одноименный метод класса Dictionary<TKey, TValue> (вызываемый через интерфейс) возвращает массив объектов типа KeyValuePair<TKey, TValue>.