

Обобщенные типы

GENERICS

GENERICS

Generics вводит в .NET концепцию параметров типа.

Это позволяет разрабатывать классы и методы, в которых тип данных будет определяться ***во время выполнения программы.***

GENERICS

- *Обобщенные* (или *параметризованные*) *типы* (*generics*) позволяют при описании пользовательских классов, структур, интерфейсов, делегатов и методов указать как параметр тип данных для хранения и обработки.
- В C++ им соответствуют шаблоны классов.

GENERICCS

Классы, описывающие структуры данных, обычно используют базовый тип **object**, чтобы хранить данные любого типа. Например, класс для стека может иметь следующее описание:

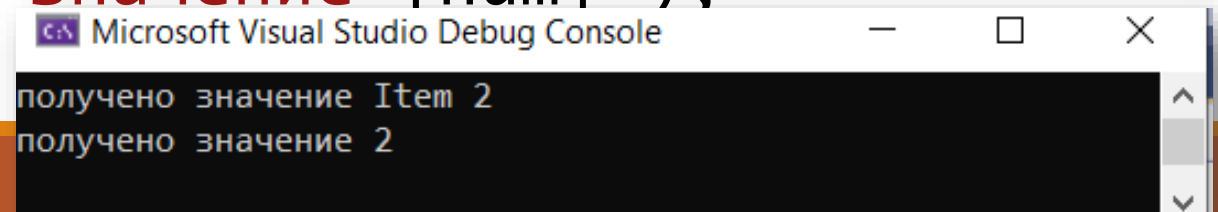
```
class UniversalStack
{
    object[] objects;
    int index;
    public UniversalStack()
    {
        objects = new object[10];
        index = 0;
    }
    public void Push(object item) => objects[index++] = item;
    public object Pop() => objects[--index];
}
```

GENERIC

Класс `UniversalStack` универсален, он позволяет хранить произвольные объекты:

```
var stack1 = new UniversalStack();  
stack1.Push("Item 1");  
stack1.Push("Item 2");  
Console.WriteLine($"получено значение {stack1.Pop()}");
```

```
var stack2 = new UniversalStack();  
stack2.Push(1);  
stack2.Push(2);  
int num = (int)stack2.Pop();  
Console.WriteLine($"получено значение {num}");
```



GENERICS

Однако универсальность класса `UniversalStack` имеет и отрицательные моменты.

1. При извлечении данных из стека необходимо выполнять приведение типов.
2. Для структурных типов (таких как `int`) при помещении/извлечении данных выполняются операции упаковки и распаковки, что отрицательно сказывается на быстродействии.
3. Неверный тип помещаемого в стек элемента может быть выявлен только **на этапе выполнения**, но не компиляции.

GENERICCS

```
var stack1 = new UniversalStack();  
stack1.Push(1);  
stack1.Push(2);  
stack1.Push("Item 1");  
  
for (int i=0; i<3; i++)  
{  
    var item = (int)stack1.Pop();  
    Console.WriteLine($"Получено значение {item}");  
}
```

GENERIC

```
5
6
7  var stack1 = new UniversalStack();
8  stack1.Push(1);
9  stack1.Push(2);
10 stack1.Push("Item 1");
11
12 for (int i=0; i<3; i++)
13 {
14     var item = (int)stack1.Pop();
15     Console.WriteLine($"Получено значение {item}");
16 }
17
18
```

Exception Unhandled

System.InvalidCastException: 'Unable to cast object of type 'System.String' to type 'System.Int32'.'

[View Details](#) | [Copy Details](#) | [Start Live Share session...](#)

▶ [Exception Settings](#)

GENERICS

- Основной причиной появления обобщенных типов была необходимость устранения описанных недостатков универсальных классов.

GENERICICS

Используя параметр универсального типа T, можно описать один класс, который может использовать клиентский код без риска приведения типов и без потерь производительности при упаковке/распаковке объектов **во время выполнения.**

GENERICCS

Сделаем класс **Stack** обобщенным.

```
class GenericStack<T>
{
    T[] objects;
    int index;
    public GenericStack()
    {
        objects = new T[10];
        index = 0;
    }
    public void Push(T item) => objects[index++] = item;
    public object Pop() => objects[--index];
}
```

GENERICICS

5
6
7
8
9
10
11
12
13
14
15
16
17
18

```
var stack1 = new GenericStack<int>();  
stack1.Push(1);  
stack1.Push(2);  
stack1.Push("Item 1");
```

Ошибка обнаружена на
этапе компиляции

```
for (int i=0; i<stack1.Count; i++)  
{  
    var item = (int)stack1.Pop();  
    Console.WriteLine($"Получено значение {item}");  
}
```

Generate method 'GenericStack.Push'

CS1503 Argument 1: cannot convert from 'string' to 'int'

Lines 19 to 20

```
public object Pop() => objects[--index];
```

```
internal void Push(string v)
```

```
{
```

```
    throw new NotImplementedException();
```

```
}
```

```
}
```

Preview changes

Приведение типа не нужно

GENERICIS

- Тип вида `Stack<int>` называется параметризованным (*обобщенным*).
- При работе с типом `Stack<int>` отпадает необходимость в выполнении приведения типов при извлечении элементов из стека.
- Теперь компилятор отслеживает, чтобы в стек помещались только данные типа `int`.
- И еще одна менее очевидная особенность. Нет необходимости в упаковке и распаковке структурного элемента, а это приводит к увеличению быстродействия.

GENERICS

- При объявлении обобщенного типа можно использовать несколько параметров.
- Сконструируем класс `class Dict<K,V>` для хранения пар «ключ-значение» с возможностью доступа к значению по ключу

Dict<K,V>

```
class Dict<T,V>
{
    T[] keys;
    V[] values;
    public Dict(int size)
    => (keys, values) = (new T[size], new V[size]);

    public void Add(T k, V v)
    {
        this[k] = v;
    }
}
```

```
public V this[T index]
```

```
{
```

```
    get
```

```
    {
```

```
        for(int i=0; i<keys.Length; i++)
```

```
        {
```

```
            if (index.Equals(keys[i])) return values[i];
```

```
        }
```

```
        return default(V);
```

```
    }
```

```
    set
```

```
    {
```

```
        // поиск такого же ключа
```

```
        for (int i = 0; i < keys.Length; i++)
```

```
        {
```

```
            if (index.Equals(keys[i]))
```

```
            {
```

```
                values[i] = value;
```

```
                return;
```

```
            }
```

```
        }
```

```
        // Поиск свободного места
```

```
        for (int i = 0; i < keys.Length; i++)
```

```
        {
```

```
            if (keys[i]==null)
```

```
            {
```

```
                keys[i] = index;
```

```
                values[i] = value;
```

```
                return;
```

```
            }
```

```
        }
```

```
        throw new Exception("коллекция заполнена");
```

```
    }
```

```
}
```

Dict<K,V>

GENERIC

```
Dict<string, Book> books = new(10);
```

```
books.Add("Book1", new Book { Id = 1, Name = "Book 1", Pages = 100 });
```

```
books.Add("Book2", new Book { Id = 2, Name = "Book 2", Pages = 300 });
```

```
Console.WriteLine($"В книге Book1 {books["Book1"].Pages} страниц");
```

Generics

ОГРАНИЧЕНИЯ ТИПА

Ограничения типа

Ограничения - это условия, налагаемые на параметры обобщенного типа.

Ограничения типа

Например, вы можете ограничить параметр типа типами, реализующими интерфейс или типами, которые имеют определенный базовый класс, имеют конструктор без параметров или которые являются ссылочными типами или типами значений.

Пользователи универсального типа не могут подставлять аргументы типа, не удовлетворяющие ограничениям.

Ограничения типа

class имя_класса<параметр> **where** параметр : ограничения

Ограничения типа

Ограничение на базовый класс

Требует наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса.

Ограничение на интерфейс

Требует реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.

Ограничение на конструктор

Требует предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора ***new()***.

Ограничение ссылочного типа

Требует указывать аргумент ссылочного типа с помощью оператора **class**.

Ограничение типа значения

Требует указывать аргумент типа значения с помощью оператора **struct**.

Ограничения класса

```
class Pet { }  
class Bird : Pet { }  
class Mammal : Pet { }
```

```
class Zoo<T> where T : Pet  
{ }
```

Ограничения интерфейса

```
class Dict<T,V> where T:IComparable  
{ }
```


Комбинация ограничений

```
class Zoo<T> where T : Comparable, new()  
    { }
```

Комбинация ограничений

Если для универсального параметра задано несколько ограничений, то они должны идти в определенном порядке:

1. Название класса (class, struct).
2. Название интерфейса
3. new()

Generics

НАСЛЕДОВАНИЕ ОБОБЩЕННЫХ КЛАССОВ

Наследование обобщенных классов

Вариант 1 (обобщенный наследник)

```
class BaseGeneric<T> { }
```

```
class ChildGeneric<T>: BaseGeneric<T>  
{ }
```

Наследование обобщенных классов

Вариант 2 (необобщенный наследник)

```
class BaseGeneric<T> { }
```

```
class Child : BaseGeneric<int>  
{ }
```

Явное указание типа



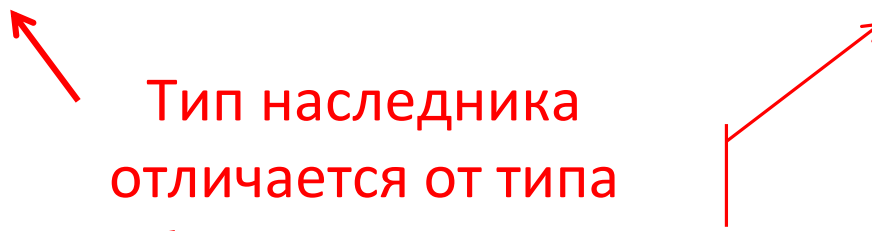
Наследование обобщенных классов

Вариант 3 (обобщенный наследник, но другого типа)

```
class BaseGeneric<T> { }
```

```
class ChildGeneric<T> : BaseGeneric<int>
{ }
```

Тип наследника
отличается от типа
базового класса



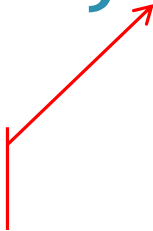
Наследование обобщенных классов

Вариант 4 (обобщенный наследник, с добавлением типа)

```
class BaseGeneric<T> { }
```

```
class ChildGeneric<T,V> : BaseGeneric<T>  
{ }
```

Добавленный тип



GENERICS

ОБОБЩЕННЫЕ МЕТОДЫ

Обобщенные методы

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.

Можно также объявить обобщенный метод со своими собственными параметрами типа.

Обобщенный метод можно описать и в необобщенном классе.

Обобщенные методы

```
public void Add(T k, V v)
{
    this[k] = v;
}
```

GENERICS

ОБОБЩЕННЫЕ ИНТЕРФЕЙСЫ

Обобщенные интерфейсы

В C# допускаются обобщенные интерфейсы.

Такие интерфейсы указываются аналогично обобщенным классам.

Применяя обобщения, можно определять интерфейсы, объявляющие методы с обобщенными параметрами.

Обобщенные интерфейсы

```
interface IUnit<T> where T : class
{
    void Move(T current);
}
```

Generics

КОВАРИАНТНОСТЬ И КОНТРАВАРИАНТНОСТЬ

Ковариантность и контравариантность

Ковариантность: позволяет использовать более конкретный тип (производный тип), чем заданный изначально

Контравариантность: позволяет использовать более универсальный тип (базовый тип), чем заданный изначально

Инвариантность: позволяет использовать только заданный тип

Ковариантность и контравариантность

```
class Hero
{
    public virtual void Greet()
    {
        Console.WriteLine("Hello");
    }
}

class IronMan : Hero
{
    public override void Greet()
    {
        Console.WriteLine("Hi! I'm Tony Stark");
    }
}
```


Ковариантный интерфейс

```
interface IPerson<out T>
{
    T CreatePerson();
}
```

Ковариантный интерфейс

```
class Person<T> : IPerson<T> where T : Hero, new()  
{  
    public T CreatePerson()  
    {  
        var hero = new T();  
        hero.Greet();  
        return hero;  
    }  
}
```

Ковариантный интерфейс

```
IPerson<Hero> person = new Person<IronMan>();
```

```
Hero hero = person.CreatePerson();
```

Контравариантный интерфейс

```
interface IAction<in T>
{
    void Greetings(T person);
}
```

Контравариантный интерфейс

```
class Action<T> : IAction<T> where T : Hero
{
    public void Greetings(T person)
    {
        person.Greet();
    }
}
```

Контравариантный интерфейс

```
IAction<IronMan> act2 = new ActionClass<Hero>();  
act2.Greetings(new IronMan());
```

Кортежи (Tuples)

Кортежи (Tuples)

Кортеж — это структура данных, которая содержит определенное число и последовательность элементов.

Кортежи (Tuples)

Кортежи описаны в пространстве имен `System.Tuples`

Кортежи (Tuples)

Платформа .NET Framework напрямую поддерживает кортежи с от одного до семи элементами.

`Tuple<T1,T2>`

Кроме того, можно создавать кортежи из восьми или более элементов путем вложения объектов кортежа в Rest свойство:

`Tuple<T1,T2,T3,T4,T5,T6,T7,TRest>`

Кортежи (Tuples)

Пример:

Можно создать экземпляр объекта **Tuple<T1,T2>** , вызвав либо конструктор

new Tuple<T1,T2>(),

либо статический метод

Tuple.Create<T1,T2>(T1, T2) .

Значения компонентов кортежа можно получить, используя свойства только для чтения **Item1** и **Item2** экземпляра.

Кортежи (Tuples)

```
var tuple1 = new Tuple<string, int>("Bob", 23);  
var tuple2 = Tuple.Create("Bob", 23);
```

```
Console.WriteLine($"Имя: {tuple1.Item1},    возраст:  
{tuple1.Item2}");
```

Кортежи (Tuples - с версии C# 7.0)

Кортежи, доступные в C# 7.0 и более поздних версиях, предоставляют краткий синтаксис для группирования нескольких элементов данных в упрощенную структуру данных.

Кортежи (Tuples - с версии C# 7.0)

```
(double, int) t1 = (4.5, 3);  
Console.WriteLine($"Tuple with elements {t1.Item1} and  
{t1.Item2}.");  
// Output:  
// Tuple with elements 4.5 and 3.
```

```
(double Sum, int Count) t2 = (4.5, 3);  
Console.WriteLine($"Sum of {t2.Count} elements is  
{t2.Sum}.");  
// Output:  
// Sum of 3 elements is 4.5.
```

Кортежи (Tuples - с версии C# 7.0)

Типы кортежей являются **типами значений**, а элементы кортежа — общедоступными полями.

Поэтому кортежи представляют собой **изменяемые** типы значений.

Кортежи (Tuples - с версии C# 7.0)

Пример (кортеж, возвращаемый методом):

```
(int min, int max) FindMinMax(int[] input)
{
    . . .

    return (min, max);
}
```


Кортежи (Tuples - с версии C# 7.0)

Пример (конструктор класса)

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Pages { get; set; }

    public Book()
    {
    }

    public Book(int id, string name, int pages)
        => (Id, Name, Pages) = (id, name, pages);
}
```

Кортежи (Tuples - с версии C# 7.0)

Пример (деконструктор класса)

```
public class Book
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Pages { get; set; }

    . . .

    public void Deconstruct(out int id, out string name, out int pages)
        => (id, name, pages) = (Id, Name, Pages);
}
```

Кортежи (Tuples - с версии C# 7.0)

Пример (деконструктор класса)

```
int bookId;  
string bookName;  
int numOfPages;
```

```
var book = new Book(1, "Harry Potter", 600);  
(bookId, bookName, numOfPages) = book;
```

Nullable

Nullable

Тип значений, допускающий значение NULL , или **T?**, представляет все значения своего базового типа значения **T**, а также дополнительное значение **NULL**.

Например, можно присвоить переменной **bool?** любое из следующих трех значений: **true**, **false** или **null**.

Nullable

В C# 8.0 появилась возможность использования **ССЫЛОЧНЫХ ТИПОВ**, допускающих значение NULL.

Nullable

Типы, допускающие значение NULL, представляют собой экземпляры универсальной структуры **System.Nullable<T>**.

Вы можете ссылаться на тип значения, допускающий значение NULL, с базовым типом T в любой из следующих взаимозаменяемых форм:

Nullable<T> или **T?**

Nullable

```
double? pi = 3.14;  
char? letter = 'a';
```

```
int m2 = 10;  
int? m = m2;
```

```
bool? flag = null;
```

```
// массив чисел, допускающих значение null:  
int?[] arr = new int?[10];
```


Nullable

Проверка значения:

```
int? a = 42;  
if (a is int valueOfA)  
{  
    Console.WriteLine($"a is {valueOfA}");  
}
```

Nullable

Проверка значения (вариант 2):

Nullable<T>.HasValue указывает, имеет ли экземпляр типа, допускающего значение NULL, значение своего базового типа.

Nullable<T>.Value возвращает значение базового типа, если HasValue имеет значение true. Если HasValue имеет значение false, свойство Value выдает исключение `InvalidOperationException`.

Nullable

Проверка значения (вариант 2):

```
int? b = 10;  
if (b.HasValue)  
{  
    Console.WriteLine($"b is {b.Value}");  
}  
else  
{  
    Console.WriteLine("b does not have a value");  
}
```

Приведение к базовому типу

```
int? a = 28;  
int b = a ?? -1;  
Console.WriteLine($"b is {b}");  
// output: b is 28
```

```
int? c = null;  
int d = c ?? -1;  
Console.WriteLine($"d is {d}");  
// output: d is -1
```