

Обобщенные типы

GENERICS

GENERICCS

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.

GENERICS

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.

GENERIC

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.**

GENERICIS

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

GENERIC

- 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]
{
```

Dict<K,V>

```
    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");
    }
}
```


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 : IComparable, 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



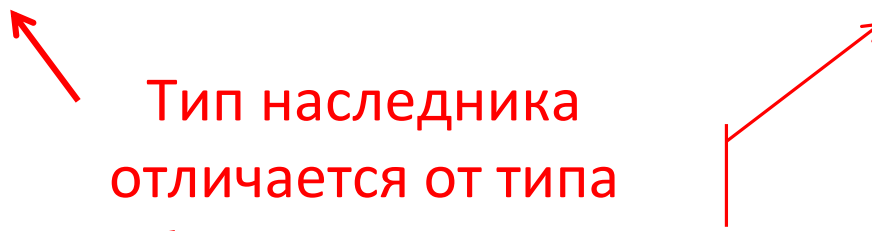
Наследование обобщенных классов

Вариант 3 (обобщенный наследник, но другого типа)

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

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

Тип наследника
отличается от типа
базового класса



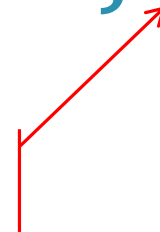
Наследование обобщенных классов

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

Параметр ковариантного типа помечается ключевым словом **out** (ключевое слово Out в Visual Basic).

Параметр ковариантного типа в качестве **возвращаемого значения метода**, принадлежащего интерфейсу, или в качестве возвращаемого типа делегата.

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

Параметр контравариантного типа помечен ключевым словом **in** (ключевое слово **In** в Visual Basic).

Параметр контравариантного типа как **тип параметра метода**, принадлежащего интерфейсу, или как тип параметра делегата. Параметр контравариантного типа можно использовать в качестве ограничения универсального типа для метода интерфейса.

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());
```


Обобщенные коллекции Microsoft

Обобщенные коллекции

Библиотека классов .NET предоставляет ряд универсальных классов коллекций в пространствах имен

System.Collections.Generic

и

System.Collections.ObjectModel

Обобщенные коллекции

Dictionary<TKey,TValue> — это универсальная версия **Hashtable**;

Здесь используется общая структура **KeyValuePair<TKey,TValue>** для перечисления вместо **DictionaryEntry**.

Обобщенные коллекции

List<T> — это универсальная версия **ArrayList**.

Универсальные классы **Queue<T>** и **Stack<T>**, соответствующие неуниверсальным версиям.

Обобщенные коллекции

Существуют общие и неуниверсальные версии **SortedList<TKey,TValue>**. Обе версии являются гибридами словаря и списка.

Универсальный класс **SortedDictionary<TKey,TValue>** является чистым словарем и не имеет неуниверсального аналога.

Обобщенные коллекции

Универсальный класс **LinkedList<T>** — это настоящий связанный список. У него нет неуниверсального аналога.

Обобщенные коллекции

Универсальный класс **Collection<T>** предоставляет базовый класс для создания собственных универсальных типов коллекций.

Обобщенные коллекции

Класс **ReadOnlyCollection<T>** предоставляет простой способ создания доступной только для чтения коллекции из любого типа, реализующего универсальный интерфейс **ICollection<T>**.

Обобщенные интерфейсы Microsoft

Интерфейсы сравнения

В пространстве имен `System` универсальные интерфейсы

`System.IComparable<T>` и `System.IEquatable<T>`

определяют методы для упорядочивания и сравнения на равенство соответственно.

Интерфейсы сравнения

В пространстве имен `System.Collections.Generic` универсальные интерфейсы

`IComparer<T>` и **`IEqualityComparer<T>`**

предлагают способ определения порядка или сравнения на равенство для типов, которые не реализуют интерфейс **`System.IComparable<T>`** или **`System.IEquatable<T>`**.

Интерфейсы коллекций

Универсальный интерфейс **ICollection<T>** — это базовый интерфейс для универсальных типов коллекций. Он предоставляет базовые функции для *добавления, удаления, копирования и перечисления* элементов.

ICollection<T> наследуется как от универсального **IEnumerable<T>**, так и от неуниверсального **IEnumerable**.

Интерфейсы коллекций

Универсальный интерфейс **IList<T>** расширяет универсальный интерфейс **ICollection<T>** методами для ***индексированного поиска***.

Интерфейсы коллекций

Универсальный интерфейс **IDictionary<TKey,TValue>** расширяет универсальный интерфейс **ICollection<T>** методами для извлечения по ключу.

Универсальные типы словарей в библиотеке базовых классов .NET также реализуют неуниверсальный интерфейс **IDictionary**.

Интерфейсы коллекций

Универсальный интерфейс **IEnumerable<T>** предоставляет универсальную структуру перечислителя (для использования оператора foreach).

Статические члены интерфейса

Статические члены интерфейса

```
public enum Gender
{
    Bith, Male
}
public interface IBreed
{
    Gender Gender { get; }
    string Name { get; set; }
    int Size { get; set; }
    static abstract int MaleMaxSize { get; }
    static abstract int BitchMaxSize { get; }
}
```

Статические члены интерфейса

```
public class GreyHound : IBreed
{
    public Gender Gender { get; }

    public GreyHound(Gender gender) => Gender = gender;
    public string Name { get; set; }
    public int Size { get; set; }

    public static int BitchMaxSize => 71;
    public static int MaleMaxSize => 76;
}
public class MiniatureSchnauzer : IBreed
{
    public Gender Gender { get; }

    public MiniatureSchnauzer(Gender gender) => Gender = gender;
    public string Name { get; set; }
    public int Size { get; set; }

    public static int BitchMaxSize => 36;
    public static int MaleMaxSize => 36;
}
```

Статические члены интерфейса

```
void GetDogInfo<T>(T dog) where T : IBreed
{
    string message = dog.Gender switch
    {
        Gender.Male => dog.Size <= T.MaleMaxSize,
        Gender.Bith => dog.Size <= T.BitchMaxSize
    }
    ? "соответствует стандарту по росту"
    : "НЕ соответствует стандарту по росту";

    Console.WriteLine($"{dog.Name} {message}");
}
```

Статические члены интерфейса

```
var gh1 = new GreyHound(Gender.Male){ Name = "GH1", Size = 80 };  
var gh2 = new GreyHound(Gender.Bith){ Name = "GH2", Size = 60 };  
var ms1 = new MiniatureSchnauzer(Gender.Male)  
           { Name = "MS1", Size = 35 };  
var ms2 = new MiniatureSchnauzer(Gender.Bith)  
           { Name = "MS2", Size = 36 };
```

```
GetDogInfo(gh1);  
GetDogInfo(gh2);  
GetDogInfo(ms1);  
GetDogInfo(ms2);
```

```
GH1 НЕ соответствует стандарту по росту  
GH2 соответствует стандарту по росту  
MS1 соответствует стандарту по росту  
MS2 соответствует стандарту по росту
```

Generic Math

Generic Math

.NET 7 представляет новые универсальные **математические интерфейсы** для библиотеки базовых классов. Доступность этих интерфейсов означает, что вы можете ограничить параметр типа универсального типа или метода «числовым» типом.

Generic Math

Поскольку C# 11 и более поздние версии позволяют определять члены **статического** виртуального интерфейса, можно объявлять операторы в новых интерфейсах для числовых типов (перегрузка операторов).

Generic Math (примеры числовых интерфейсов)

IBinaryInteger<TSelf>

IBinaryNumber<TSelf>

IFloatingPoint<TSelf>

INumber<TSelf>

INumberBase<TSelf>

ISignedNumber<TSelf>

IUnsignedNumber<TSelf>

IAdditiveIdentity<TSelf,TResult>

IMinMaxValue<TSelf>

IMultiplicativeIdentity<TSelf,TResult>

Generic Math (примеры интерфейсов операторов)

<code>IAdditionOperators<TSelf,TOther,TResult></code>	$x + y$
<code>IBitwiseOperators<TSelf,TOther,TResult></code>	$x \& y$, $x \mid y$, $x \wedge y$, and $\sim x$
<code>IComparisonOperators<TSelf,TOther,TResult></code>	$x < y$, $x > y$, $x \leq y$, and $x \geq y$
<code>IDecrementOperators<TSelf></code>	$--x$ and $x--$
<code>IDivisionOperators<TSelf,TOther,TResult></code>	x / y
<code>IEqualityOperators<TSelf,TOther,TResult></code>	$x == y$ and $x != y$
<code>IncrementOperators<TSelf></code>	$++x$ and $x++$
<code>IModulusOperators<TSelf,TOther,TResult></code>	$x \% y$
<code>IMultiplyOperators<TSelf,TOther,TResult></code>	$x * y$
<code>IShiftOperators<TSelf,TOther,TResult></code>	$x \ll y$ and $x \gg y$
<code>ISubtractionOperators<TSelf,TOther,TResult></code>	$x - y$
<code>IUnaryNegationOperators<TSelf,TResult></code>	$-x$
<code>IUnaryPlusOperators<TSelf,TResult></code>	$+x$

Generic Math (примеры интерфейсов функций)

<code>IExponentialFunctions<TSelf></code>	e^x , $e^x - 1$, 2^x , $2^x - 1$, 10^x , and $10^x - 1$.
<code>IHyperbolicFunctions<TSelf></code>	$\operatorname{acosh}(x)$, $\operatorname{asinh}(x)$, $\operatorname{atanh}(x)$, $\cosh(x)$, $\sinh(x)$, and $\tanh(x)$.
<code>ILogarithmicFunctions<TSelf></code>	$\ln(x)$, $\ln(x + 1)$, $\log_2(x)$, $\log_2(x + 1)$, $\log_{10}(x)$, and $\log_{10}(x + 1)$.
<code>IPowerFunctions<TSelf></code>	x^y .
<code>IRootFunctions<TSelf></code>	$\operatorname{cbrt}(x)$ and $\operatorname{sqrt}(x)$.
<code>ITrigonometricFunctions<TSelf></code>	$\operatorname{acos}(x)$, $\operatorname{asin}(x)$, $\operatorname{atan}(x)$, $\cos(x)$, $\sin(x)$, and $\tan(x)$.

Generic Math(форматирование и приведение типа)

IParsable<TSelf>	T.Parse(string, IFormatProvider) T.TryParse(string, IFormatProvider, out TSelf).
ISpanParsable<TSelf>	T.Parse(ReadOnlySpan<char>, IFormatProvider) T.TryParse(ReadOnlySpan<char>, IFormatProvider, out TSelf).
IFormattable	value.ToString(string, IFormatProvider).
ISpanFormattable	value.TryFormat(Span<char>, out int, ReadOnlySpan<char>, IFormatProvider).

Generic Math

```
var complexList = new List<Complex>  
{ new Complex(1, 1), new Complex(2, 2) };  
  
Console.WriteLine(complexList.Sum());
```

Generic Math

```
var complexList = new List<Complex> { new Complex(1, 1), new Complex(2, 2) };  
Console.WriteLine(complexList.Sum());
```

 (local variable) List<Complex>? complexList

'complexList' is not null here.

CS1929: 'List<Complex>' does not contain a definition for 'Sum' and the best extension method overload 'ParallelEnumerable.Sum(ParallelQuery<decimal>)' requires a receiver of type 'System.Linq.ParallelQuery<decimal>'

Generic Math

```
public readonly struct Complex :  
    IEquatable<Complex>, IFormattable,  
    IParsable<Complex>, ISpanFormattable,  
    IAdditionOperators<Complex, Complex, Complex>  
    . . .  
    IUnaryPlusOperators<Complex, Complex>,  
    ISignedNumber<Complex>
```

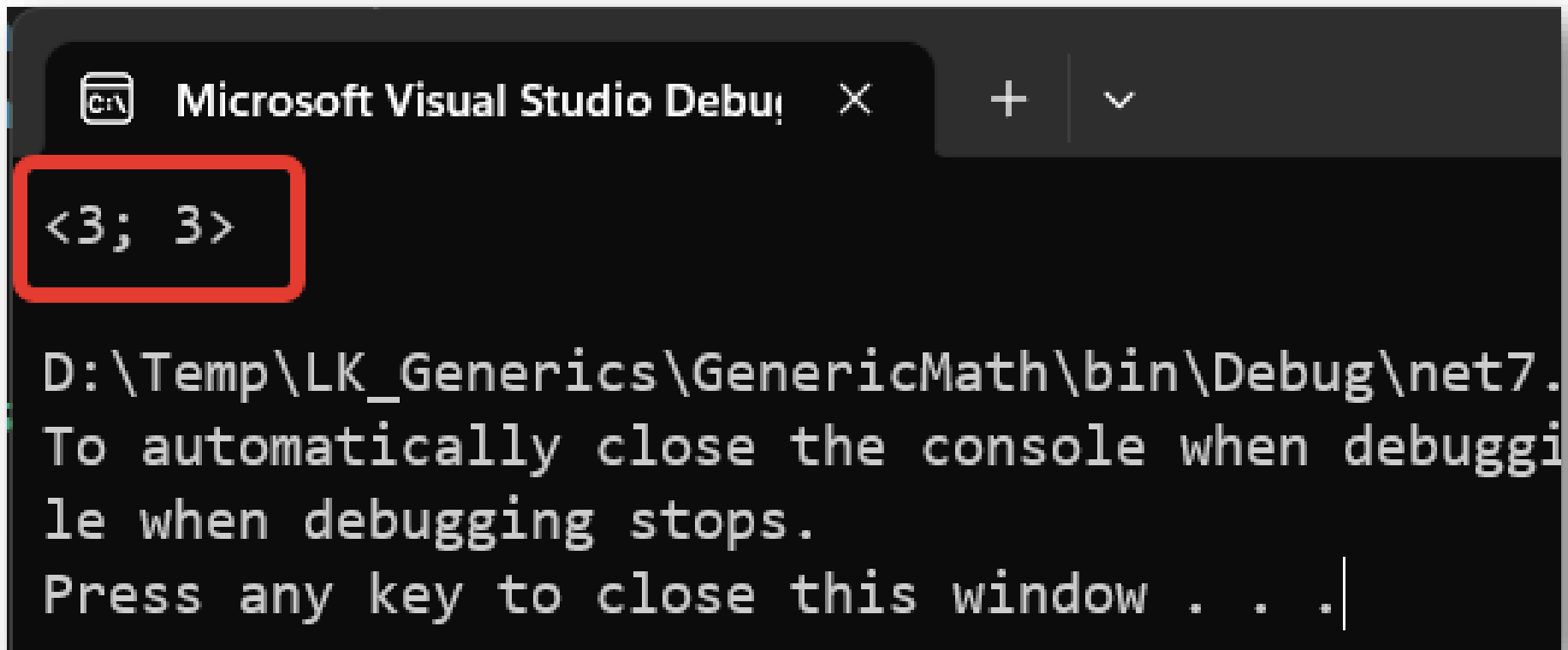
Generic Math

```
internal static class Extensions
{
    public static T Sum<T>(this IEnumerable<T> data)
        where T : IAdditionOperators<T, T, T>, new()
    {
        T result = new();
        foreach (var item in data)
        {
            result += item;
        }
        return result;
    }
}
```

Generic Math

`using` GenericMath;

Generic Math



```
Microsoft Visual Studio Debug Console
```

<3; 3>

D:\Temp\LK_Generics\GenericMath\bin\Debug\net7.
To automatically close the console when debugging
is complete when debugging stops.
Press any key to close this window . . .

Generic Math

```
internal record struct CartItem(string Name,  
double Price){}
```

Generic Math

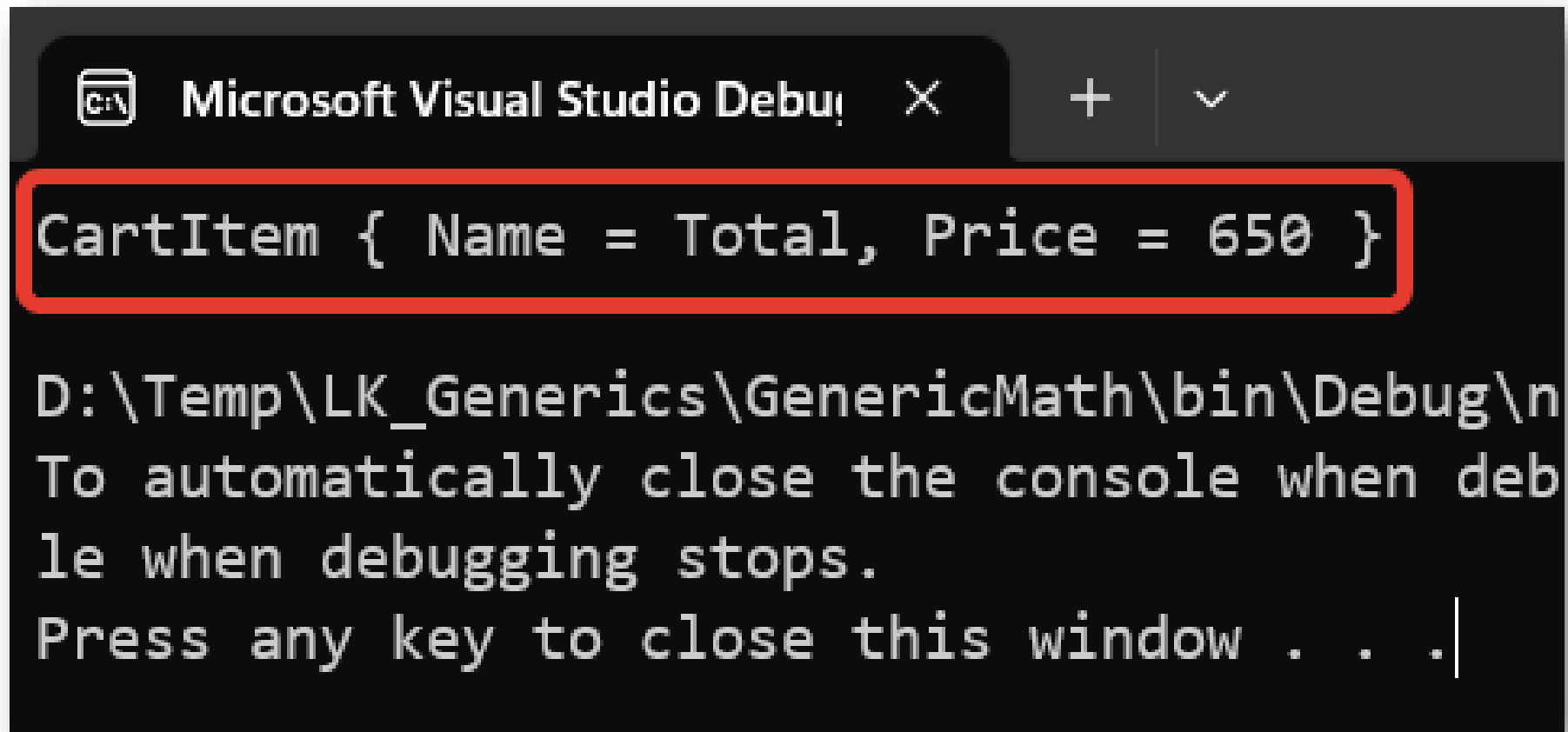
```
var cart = new List<CartItem>
{
    new CartItem("Car", 150),
    new CartItem("Flower", 200),
    new CartItem("Shoes", 300)
};

Console.WriteLine(cart.Sum());
```

Generic Math

```
internal record struct CartItem(string Name, double Price)
    : IAdditionOperators<CartItem, CartItem, CartItem>
{
    public static CartItem operator +(CartItem left,
                                      CartItem right)
    {
        return new CartItem("Total", left.Price+right.Price);
    }
}
```

Generic Math



The image shows a screenshot of the Microsoft Visual Studio Debug Console window. The window has a dark theme and a tab labeled 'Microsoft Visual Studio Debug' with a close button (X) and a dropdown arrow (v). The console output shows a breakpoint hit for 'CartItem' with the following details: 'Name = Total, Price = 650'. This line is highlighted with a red rectangular border. Below this, the console shows the file path 'D:\Temp\LK_Generics\GenericMath\bin\Debug\n' and a message: 'To automatically close the console when debugging stops. Press any key to close this window . . .'. A cursor is visible at the end of the last line.

```
CartItem { Name = Total, Price = 650 }  
  
D:\Temp\LK_Generics\GenericMath\bin\Debug\n  
To automatically close the console when deb  
le when debugging stops.  
Press any key to close this window . . .
```

Generic Math

<https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/static-virtual-interface-members>

<https://devblogs.microsoft.com/dotnet/dotnet-7-generic-math/>