

В.В. Подбельский

Использованы иллюстрации пособия Daniel Solis, Illustrated C#

Иллюстрации к курсу лекций по дисциплине «Программирование на C#»

Модуль 3. Лекция 5а

Обобщённое программирование

Обобщённое Программирование в С#

При написании программ часто возникает необходимость обобщить функционал для различных типов.

Одним из таких механизмов является наследование, которое рассматривалось на лекциях ранее.

Другим механизмом в С# являются **обобщённые типы** (Generic types), которые позволяют использовать **параметры типов**.

Фактически, обобщённые типы представляют собой *шаблоны типов*, на основе которых при подстановке *аргументов типов* в процессе компиляции создаются *конкретные типы*.

Подробнее об обобщённом программировании: https://en.wikipedia.org/wiki/Generic_programming

Открытые и Закрытые Обобщённые Типы

Важно: помните, что на этапе выполнения существуют только обобщённые типы/методы, в которых выполнена подстановка параметров.

Такие типы называют **закрытыми** (закрытыми сконструированными).

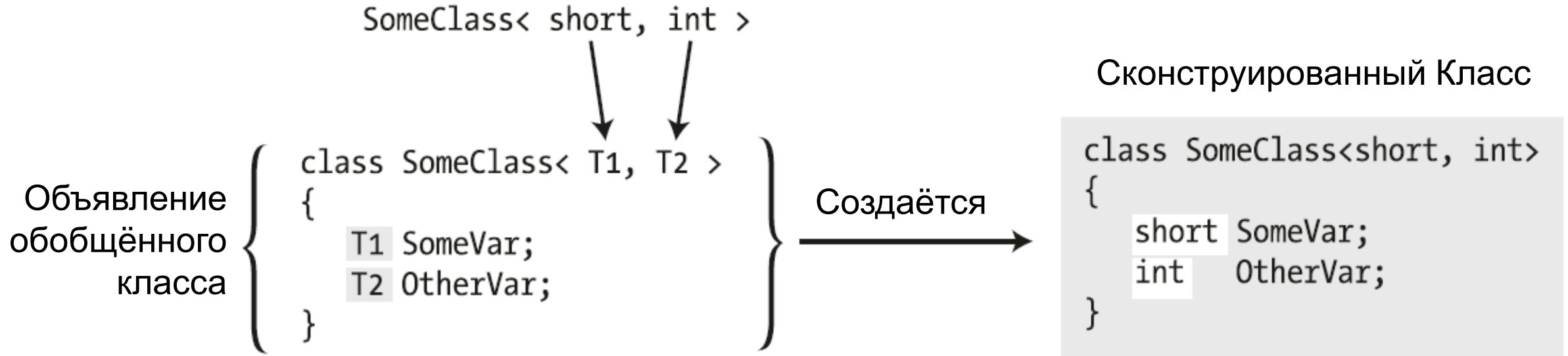
В свою очередь, объявленные обобщённые типы без подстановки аргументов типов называют **открытыми**.

```
public class SimplifiedStack<T>
{
    List<T> _stackBase = new List<T>();
```

Параметр типа T объявлен, однако аргумент типа для него не подставлен. Следовательно, тип SimplifiedStack<T> – открытый.

```
    public void Push(T value) => _stackBase.Add(value);
    public T Pop()
    {
        if (_stackBase.Count == 0)
            throw new InvalidOperationException("The stack is empty");
        T elem = _stackBase[_stackBase.Count - 1];
        _stackBase.RemoveAt(_stackBase.Count - 1);
        return elem;
    }
}
```

Схема Создания Закрытого Типа




Закрытый (сконструированный) тип получается путём подстановки `short` на место `T1` и `int` на место `T2`.

Заметьте: подстановка аргументов типов на место параметров типов позволяет избежать упаковки/распаковки при выполнении операций.

Аргументы Типа и Параметры Типа

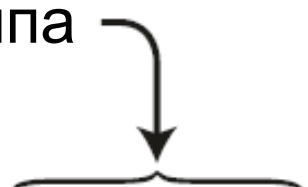
Параметры Типа



```
class SomeClass< T1, T2 >  
{  
    ...  
}
```

Объявление Обобщённого Класса

Аргументы Типа



```
SomeClass< short, int >
```

Закрытый (Сконструированный) Класс

Не путайте:

- **Параметры типа** – «заглушки», на место которых подставляются реальные типы;
- **Аргументы типа** – реальные типы, подставляемые на место параметров типа.

Создание Закрытого (Сконструированного) Типа

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

// Получаем 2 закрытых типа:
SimplifiedStack<int> stackOfInts = new SimplifiedStack<int>();
SimplifiedStack<string> stackOfStrings = new();
for (int i = 1; i <= 5; ++i)
{
    stackOfInts.Push(i);
    stackOfStrings.Push($"string{i}");
}
Console.WriteLine($"Number: {stackOfInts.Pop()}");
Console.WriteLine($"Line: {stackOfStrings.Pop()}");
```

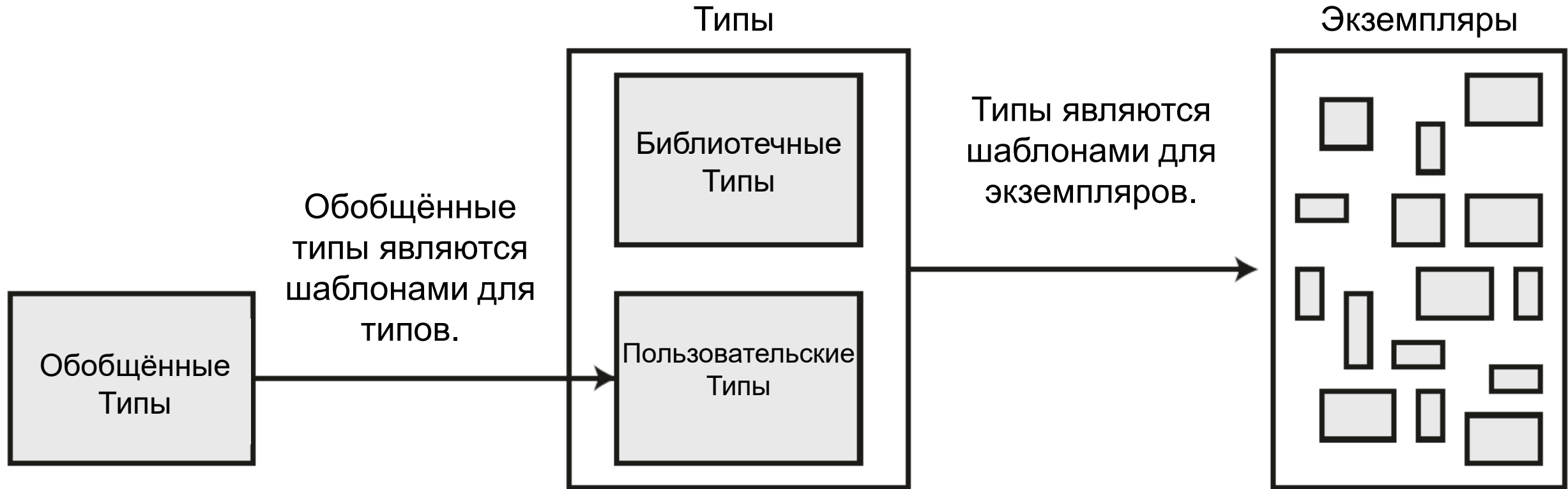
С# не позволяет опустить треугольные скобки при использовании конструктора.

Параметр Т фиксирован в случае с закрытыми типами – компилятор подставляет int/string во всех местах, где используется Т.

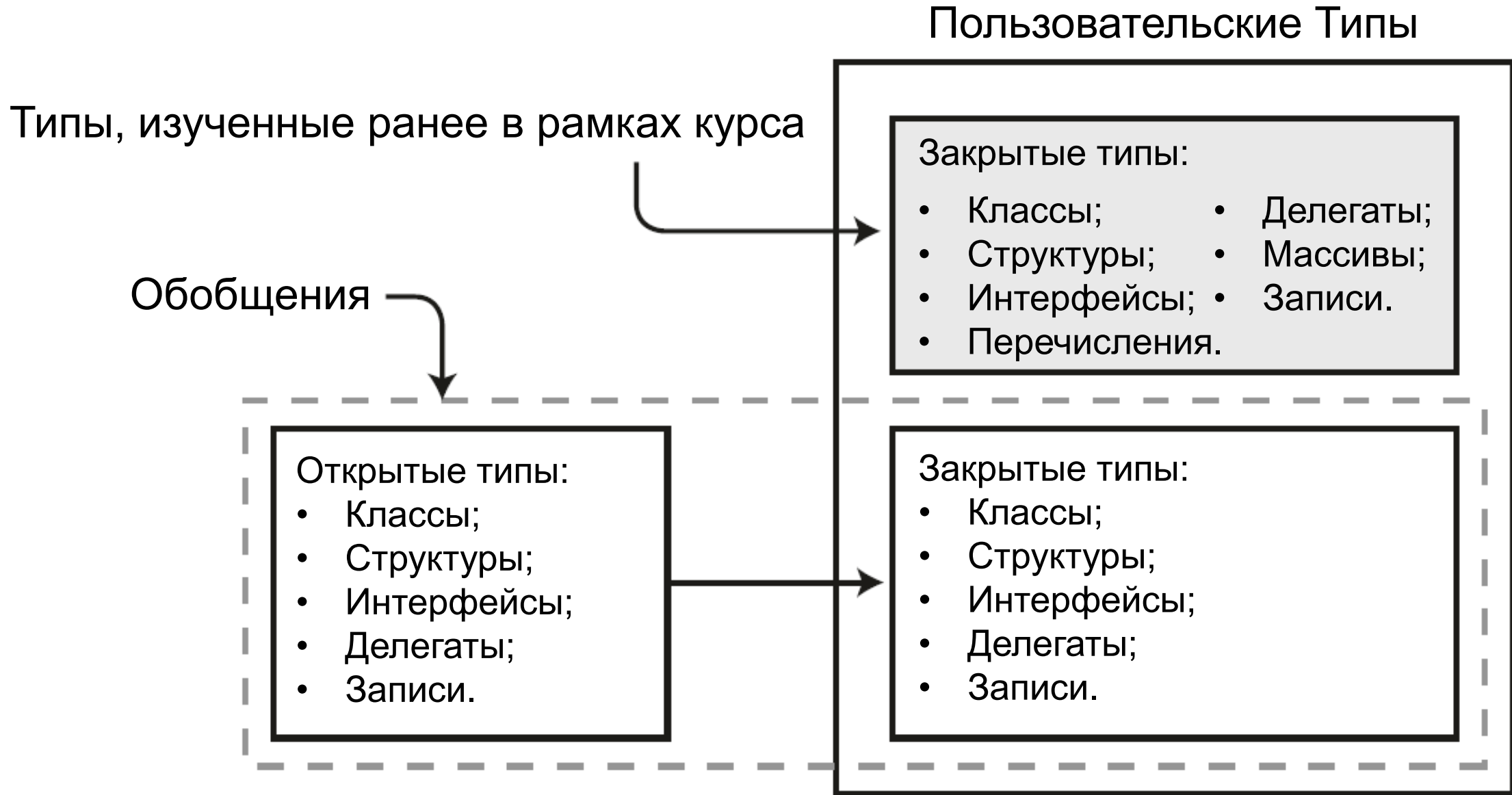
Вывод:

Number: 5
Line: string5

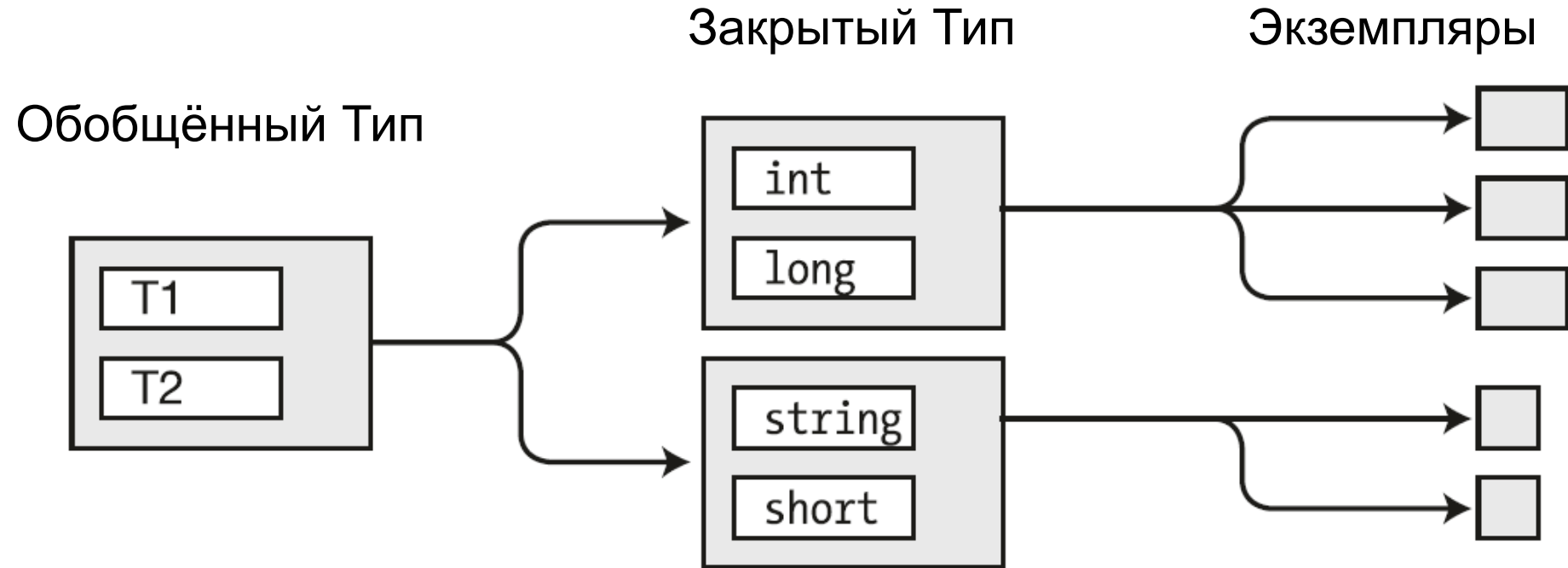
Схема Типов с Учётом Обобщений



Какие Типы Могут Быть Обобщены



Создание Экземпляров из Обобщённых Типов



① Объявляется параметр типа.


② Формируются закрытые типы путём подстановки конкретных типов.

③ Создаются экземпляры закрытых типов.


Закрытые Классы со Статическими Членами

```
class SomeClass< T1, T2 >  
{  
    static T1 SomeVar;  
    T2 OtherVar;  
}
```

```
...  
var first = new SomeClass<short, int> ( );  
var second = new SomeClass<int, long> ( );  
...
```



```
class SomeClass <short,int>  
{  
    static short SomeVar;  
    int OtherVar;  
}
```



```
class SomeClass <int,long>  
{  
    static int SomeVar;  
    long OtherVar;  
}
```

Различные закрытые классы независимы друг от друга. Таким образом, у каждого из них будет свой независимый набор статических полей.

Допустимые Операции над Параметрами Типов

Обобщённые типы в С# устроены таким образом, что по умолчанию они требуют, чтобы все выполняемые операции были допустимы для любых подставляемых аргументов на этапе компиляции.

Так, подобный код не скомпилируется:

```
class Comparer<T>
{
    // Ошибка компиляции: не любой тип определяет операцию <
    static public bool LessThan(T i1, T i2) => i1 < i2;
}
```

В общем случае для параметров типа доступен только функционал класса Object.

Вывод: нужны ограничения, дающие больше информации о допустимых операциях.

Ограничения Параметров Типов-1

Ограничение	Описание Допустимых Типов-Аргументов
<code>class?</code>	Любой ссылочный тип (класс, интерфейс, делегат, массив или запись).
<code>struct</code>	Любой не-nullable тип значения. Автоматически включает в себя ограничение <code>new()</code> и не может сочетаться с ним явно. Несовместимо с ограничением <code>unmanaged</code> .
<code><ClassName>?</code>	Тип-аргумент является типом <code>ClassName</code> или его наследниками.
<code><InterfaceName>?</code>	Тип-аргумент обязан реализовывать интерфейс <code>InterfaceName</code> . На один параметр типа могут накладываться несколько ограничений на интерфейсы.
<code>new()</code>	Тип-аргумент должен иметь открытый конструктор без параметров. Несовместимо с ограничениями <code>struct</code> и <code>unmanaged</code> .
<code>notnull</code> (C# 8.0)	Тип-аргумент должен не допускать значения <code>null</code> .

? – добавление этого символа позволяет использовать типы, допускающие `null`.

Ограничения Параметров Типов-2

Ограничение	Описание Допустимых Типов-Аргументов
<code>T : U</code>	Тип-аргумент совпадает с U или является его наследником. При использовании nullable-контекста T должен быть не-nullable типом, если U – не nullable-тип.
<code>unmanaged</code>	Тип-аргумент должен быть « <u>неуправляемым</u> ». Автоматически подразумевает наличие ограничений struct и new(), поэтому несовместимо с ними.
<code>default (C# 9.0)</code>	[Только для обобщённых методов] Используется для устранения неоднозначности, связанной с переопределением методов без ограничений (<u>см. документацию</u>).

Несовместимые друг с другом ограничения параметров типов:

- class, struct, unmanaged, notnull, default.

Порядок Ограничений Параметров Типов

При указании ограничений на параметры типов необходимо придерживаться определённого порядка, иначе возникнет ошибка компиляции:

Первичное ограничение (0 или 1)	Вторичные ограничения (0 или более)	Ограничение на Конструктор (0 или 1)
<div>{ class struct unmanaged notnull ClassName default }</div>	<div>{ InterfaceName1, InterfaceName2, ... }</div>	<div>{ new() }</div>

Синтаксис Ограничений Параметров Типов

Для указания ограничений используются предложения с контекстно-ключевым словом **where**:

[Определение типа] where <Параметр типа> : <ограничения>

Для нескольких параметров определяются различные наборы предложений:

```
public class ValueList<T>
    where T : struct, IComparable<T>
{ }
```

```
public class LinkedList<T, U>
    where T : IComparable<T>
    where U : ICloneable
{ }
```

```
public class Dictionary<TKey, TValue>
    where TKey : IEnumerable<TKey>, new()
{ }
```

Особенности Ограничения where T : class

При использовании ограничения where T : class помните об одной важной особенности: операции == и != всегда будут работать как проверка равенства ссылок в случае обобщённых типов, даже при наличии явного переопределения:

```
using System;
```

```
string s = "hello";
```

```
// Специально создадим строку, не связанную с данной в памяти:
```

```
System.Text.StringBuilder sb = new("hello");
```

```
EqualityDemo<string> eq = new(s);
```

```
eq.EqualityOperatorTest(sb.ToString());
```

```
eq.EqualityOperatorTest(s);
```

```
public class EqualityDemo<T> where T : class
```

```
{
```

```
    private T _val;
```

```
    public EqualityDemo(T value) => _val = value;
```

```
    public void EqualityOperatorTest(T other) => Console.WriteLine(_val == other);
```

```
}
```

Вывод:

False

True

Хотя фактически строки равны, будет выполнено сравнение ссылок, а результат окажется false.

Наследование Обобщённых Типов-1

Механизм обобщений в C# может совмещаться с наследованием. При этом обобщённые типы могут наследоваться как от необобщённых типов, так и от других открытых и закрытых сконструированных:

```
public class GenericBase<T> { }  
public class NonGenericBase { }
```

// Открытый → открытый:

```
public class GenericDerived1<U> : GenericBase<U> { }
```

// Открытый → закрытый сконструированный:

```
public class GenericDerived2<T> : GenericBase<int> { }
```

// Открытый сконструированный → необобщённый:

```
public class GenericDerived3<T> : NonGenericBase { }
```

Наследование Обобщённых Типов-2

Необобщённые типы могут наследоваться исключительно от закрытых сконструированных, т. к. в случае с открытыми параметр типа оказывается неопределённым, что недопустимо. Пример корректного наследования:

```
public class GenericBase<T> { }  
public class NonGenericBase { }
```

// Необобщённый → закрытый сконструированный:

```
public class NonGenericDerived : GenericBase<byte> { }
```

В случае наследования от обобщённых типов с несколькими параметрами необходимо предоставить все необходимые аргументы обобщённому родителю:

```
public class GenericBase2<T, U> { }
```

// В обоих случаях все параметры родителя фиксируются:

```
public class GenericExample1<S, V> : GenericBase2<S, V> { }
```

```
public class GenericExample2<S> : GenericBase2<string, S> { }
```

Пример: Обобщённая Структура

```
public struct Nullable<T>
    where T : struct
{
    public T Value { get; private set; }
    public bool HasValue { get; private set; }

    public Nullable(T value) => (Value, HasValue) = (value, true);

    public static implicit operator Nullable<T>(T value) => new(value);
    public static explicit operator T(Nullable<T> value) => value.HasValue
        ? value.Value
        : throw new InvalidOperationException("The value is not present.");

    public T ValueOrDefault() => HasValue ? Value : default;

    public override string ToString() => HasValue ? Value.ToString() : string.Empty;
}
```

Nullable-структура имеет смысл только для типов значения, не допускающих null.

Использование Обобщённой Структуры

```
using System;
```

Значение типа по умолчанию. Эквивалентно:
new(), default(Nullable<int>), new Nullable<int>().

```
Nullable<int> intVal = default;
```

```
Console.WriteLine($"{(intVal.HasValue ? $"{intVal.Value}" : "No value present")}");
```

```
intVal = int.MaxValue;
```

```
Console.WriteLine($"inValue now contains: {intVal}");
```

```
Nullable<DateTime> dateVal = default;
```

```
Console.WriteLine($"The value for empty Nullable<DateTime>:  
{dateVal.ValueOrDefault()}");
```

```
// Строка ниже приведёт к ошибке компиляции - T не может быть ссылочным типом:  
// Nullable<string> stringVal = "this won't work";
```

Вывод:

No value present.

inValue now contains: 2147483647

The value for empty Nullable<DateTime>: 01-Jan-01 00:00:00

Пример-1: Обобщённый Интерфейс

```
using System;
```

```
Student s1 = new("Daniil", "Sagalov", 196);  
Student s2 = s1 with { Group = 191 };  
Student s3 = new("Egor", "Matveev", 183);  
Student s4 = s3;  
Console.WriteLine(s1.Equals(s2));  
Console.WriteLine(s1.Equals(s3));  
Console.WriteLine(s3.Equals(s4));
```

Вывод:

False
False
True

```
public interface IEquatable<T>  
{  
    public bool Equals(T other);  
}
```

Компилятор автоматически генерирует метод Equals для типов записей.

```
public record Student(string FirstName, string LastName, int Group)  
    : IEquatable<Student>;
```

Ошибка при Реализации Обобщённого Интерфейса

Представленный ниже код НЕ компилируется, т. к. IPrintable<TPrice> и IPrintable<decimal> могут совпадать, что недопустимо:

```
public interface IPrintable<T>
{
    public void Print(T value);
}
```

```
public record Product<TPrice>(uint ID, TPrice Price)
    : IPrintable<TPrice>, IPrintable<decimal>
{
    public void Print(TPrice value)
        => System.Console.WriteLine($"ID: {ID}, Price: {Price}");

    public void Print(decimal value)
        => System.Console.WriteLine($"Price request for {ID}: {value}");
}
```

Данная реализация является недопустимой.

Обобщённые Делегат-Типы

Как и большинство других типов, делегат-типы тоже могут быть обобщёнными:

```
public delegate R GenericDelegate1<T, R>(T value);  
public delegate TR Func<T1, T2, TR>(T1 p1, T2 p2);  
public delegate void RefAction<T1, T2>(ref T1 p1, ref T2 p2);
```

Тем не менее, необходимость в этом практически отсутствует, т. к. в библиотеке определены стандартные обобщённые делегаты Action и Func.

Практическая необходимость в определении дополнительных обобщённых делегат-типов возникает исключительно в сценарии, когда параметры типов должны использоваться с модификаторами ref, in или out.

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-delegates>

<https://docs.microsoft.com/en-us/dotnet/api/system.action-1>

<https://docs.microsoft.com/en-us/dotnet/api/system.func-1>

Пример: Обобщённый Делегат-Тип

```
using System;
```

```
var myDel = new Func<int, int, string>(PrintString);  
Console.WriteLine($"Total: {myDel(15, 13)}");
```

```
static string PrintString(int p1, int p2) => (p1 + p2).ToString();
```

```
public delegate TR Func<T1, T2, TR>(T1 p1, T2 p2);
```

Вывод:

Total: 28



Обобщённый делегат-тип.

Обобщённые Методы

Кроме типов в С# дополнительно можно использовать **обобщённые методы**, указывая параметры типа метода *между идентификатором метода и его списком параметров* в треугольных скобках.

Важно: все остальные функциональные члены типов в С# обобщены быть НЕ могут. Не стоит путать использование параметров типа, объявленных в обобщённом типе, методом и обобщённые методы.

Как и в случае с типами, при написании обобщённых методов допустимы ограничения на параметры типа с помощью *where* *после списка параметров метода*:

```
[Модификаторы] <Тип возвр. знач.> <Идентификатор> <<Параметры типов...>> ([Параметры методы]) [where ...] { тело метода }
```

Пример Обобщённого Метода

```
int leftInt = 20, rightInt = 10;  
string leftStr = "left", rightStr = "right";  
SwapUtil.Swap<int>(ref leftInt, ref rightInt);  
SwapUtil.Swap(ref leftStr, ref rightStr);  
System.Console.WriteLine($"leftInt: {leftInt}, rightInt: {rightInt}");  
System.Console.WriteLine($"leftStr: {leftStr}, rightStr: {rightStr}");
```

При подстановке не обязательно указывать типы: компилятор способен вывести их из аргументов.

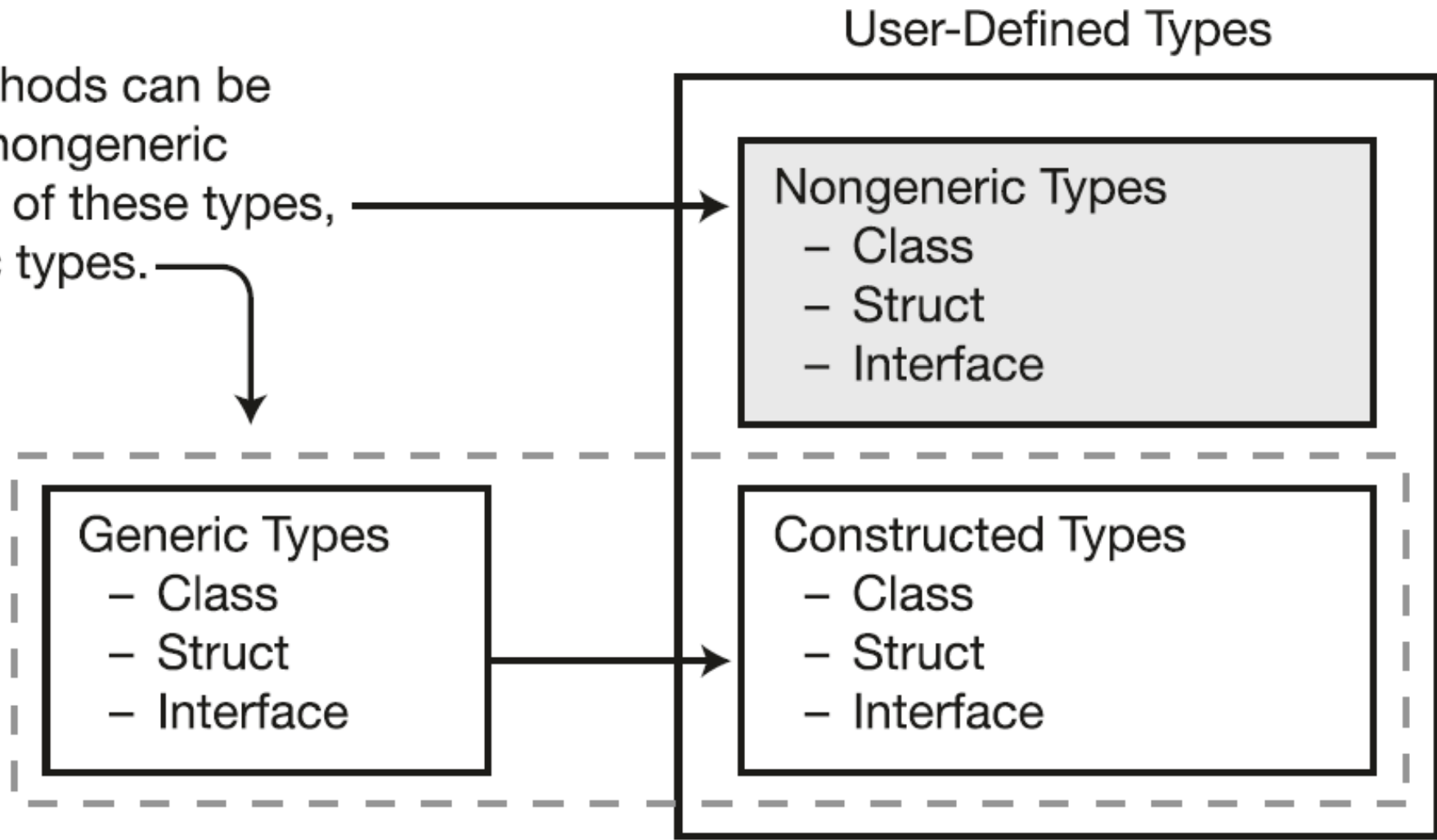
```
public static class SwapUtil  
{  
    public static void Swap<T>(ref T lhs, ref T rhs)  
    {  
        T temp = lhs;  
        lhs = rhs;  
        rhs = temp;  
    }  
}
```

Вывод:

leftInt: 10, rightInt: 20
leftStr: right, rightStr: left

Схема: Обобщённые Методы

Generic methods can be included in nongeneric declarations of these types, or in generic types.

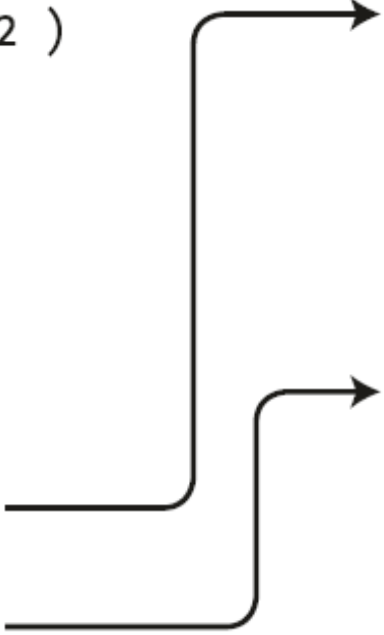


Различные Аргументы Обобщённого Метода

```
void DoStuff<T1, T2>( T1 t1, T2 t2 )  
{  
    T1 someVar = t1;  
    T2 otherVar = t2;  
    ...  
}  
...
```

```
DoStuff<short, int>(sVal, iVal);
```

```
DoStuff<int, long>(iVal, lVal);
```



```
void DoStuff <short,int >(short t1, int t2 ) {  
    short someVar = t1;  
    int otherVar = t2;  
    ...  
}
```

```
void DoStuff <int,long >(int t1, long t2 ) {  
    int someVar = t1;  
    long otherVar = t2;  
    ...  
}
```

Обобщённый Метод. Пример 2. Часть 1

```
using System;
```

```
public static class ArrayReverseTool
```

```
{
```

```
    static public void ReverseAndPrint<T>(T[] arr)
```

```
    {
```

```
        Array.Reverse(arr);
```

```
        Array.ForEach(arr, elem => Console.Write(elem + " "));
```

```
        Console.WriteLine();
```

```
    }
```

```
}
```

Обобщённый Метод. Пример 2. Часть 2

```
using System;
```

```
int[] intArray = { 3, 5, 7, 9, 11 };  
string[] stringArray = { "first", "second", "third" };  
double[] doubleArray = { 3.567, 7.891, 2.345 };  
ArrayReverseTool.ReverseAndPrint<int>(intArray);  
ArrayReverseTool.ReverseAndPrint(intArray);  
ArrayReverseTool.ReverseAndPrint<string>(stringArray);  
ArrayReverseTool.ReverseAndPrint(stringArray);  
ArrayReverseTool.ReverseAndPrint<double>(doubleArray);  
ArrayReverseTool.ReverseAndPrint(doubleArray);
```

Вывод:

11 9 7 5 3

3 5 7 9 11

third second first

first second third

2.345 7.891 3.567

3.567 7.891 2.345

Обобщённый Метод Расширения. Часть 1

```
public static class ExtendHolder
{
    public static void Print<T>(this Holder<T> h)
    {
        T[] vals = h.Values;
        System.Console.WriteLine($"{vals[0]},\t{vals[1]},\t{vals[2]}");
    }
}

public class Holder<T>
{
    public T[] Values { get; init; } = new T[3];
    public Holder(T v0, T v1, T v2)
        => (Values[0], Values[1], Values[2]) = (v0, v1, v2);
}
```

Обобщённый Метод Расширения. Часть 2

```
class Program
{
    static void Main()
    {
        Holder<int> intHolder = new(3, 5, 7);
        Holder<string> stringHolder = new("a1", "b2", "c3");
        intHolder.Print();
        stringHolder.Print();
    }
}
```

Вывод:

3, 5, 7
a1, b2, c3

Ковариантность и Контравариантность. Введение

Данные механизмы представляют собой способ переноса наследования типов на производные от них типы — контейнеры, обобщённые типы, делегаты и т. п.

С помощью ковариантности и контравариантности можно неявно преобразовывать ссылки на типы коллекций, типы делегатов и аргументы обобщений.

Ковариантность (*covariance*) сохраняет совместимость операции присваивания, а **контравариантность** (*contravariance*) заменяет ее на обратную.

```
// напоминание - совместимость типов при операции присваивания:  
string str = "line";  
// Приведение производного типа к базовому:  
object obj = str;
```

Ковариантность

Ковариантность позволяет использовать производный тип с большей глубиной наследования, чем задано изначально.

```
// Ссылки в C# ковариантны: по ссылке типа родителя  
// всегда можно разместить объект типа наследника:  
Base derived = new Derived();
```

```
public class Base { }  
public class Derived : Base { }
```

В случае необходимости использования параметров ковариантных типов для обобщённых интерфейсов и делегат-типов соответствующие параметры типа объявляются с ключевым словом **out**.

Контравариантность

Контравариантность позволяет использовать более общий тип (с меньшей глубиной наследования), чем заданный изначально.

```
class Base { }  
class Derived : Base { }
```

Предполагается приведение объекта базового типа к производному типу:

```
< Derived d = Base b; >
```

В обобщениях параметр контравариантного типа объявляется с ключевым словом **in**.

Инвариантность

Инвариантность допускает использование только изначально заданного типа.

```
class Base { }  
class Derived : Base { }
```

Т.е. параметр инвариантного обобщённого типа не является ни ковариантным, ни контравариантным:

```
< Derived d = Base b; >    // Ошибка.  
< Base b = Derived d; >    // Ошибка.
```

В обобщениях параметр типа без **in/out** является инвариантным (**по умолчанию**)!

Сравнение Ковариантности и Контравариантности

// Совместимость операции присваивания:

```
string str = "test";
```

```
object obj = str; // Объект типа наследника присваивается по ссылке базового типа.
```

// Ковариантность: *public interface IEnumerable<out T> : System.Collections.IEnumerable*

```
IEnumerable<Derived> strings = new List<Derived>();
```

```
IEnumerable<Base> objects = strings; // Сохраняется совместимость типов операции =.
```

// Контравариантность: *public delegate void Action<in T>(T obj);*

```
static void SetObject(Base o) { }
```

```
Action<Base> actObject = SetObject;
```

```
Action<Derived> actString = actObject; // обратная операции = //совместимость типов
```

```
class Base { }
```

```
class Derived : Base { }
```

Ковариантность и Контравариантность Делегат-Типов

При объявлении делегатов:

```
public delegate void Action<in T>(T obj);
```

контравариантность

Важно: При отсутствии in/out – инвариантность (чёткое совпадение типа)!

```
public delegate TResult Func<out TResult>();
```

ковариантность

```
public delegate TResult Func<in T1, out TResult>(T1 arg1);
```

контравариантность

ковариантность

```
public delegate TOutput Converter<in TInput, out TOutput>(TInput input);
```

контравариантность

ковариантность

Применение Ковариантности и Контравариантности

При объявлении делегат-типов:

```
public delegate void Action<in T>(T obj);  
public delegate TResult Func<out TResult>();  
public delegate TResult Func<in T1, out TResult>(T1 arg1);
```

При использовании делегатов:

```
static object GetObject() { return null; }  
static void SetObject(object obj) { }  
  
static string GetString() { return ""; }  
static void SetString(string str) { }  
  
static void Test() {  
    // Ковариантность (тип возвращаемого значения):  
    Func<object> del = GetString; // string вместо object.  
  
    // Контравариантность (тип входного параметра):  
    Action<string> del2 = SetObject; // object вместо string.  
}
```

Ковариантность и Контравариантность

- Ковариантность параметров типа доступна только для **обобщённых интерфейсов** и **делегатов** (generic delegate). Соответственно, **in** и **out** можно использовать только в этих типах.
- Обобщённые интерфейсы и делегаты могут иметь и ковариантные и контравариантные параметры типа одновременно:

```
delegate TResult Func<in T1, out TResult>(T1 arg1);
```
- Вариативность применяется только к **ссылочным** типам; если указать тип значения для аргумента вариативного типа, то этот параметр типа становится в результате инвариантным.
- **Вариативность не применима к многоадресным делегатам** (multicast delegate). Поэтому для заданных двух делегатов типов Action<Derived> и Action<Base> нельзя объединять первый делегат со вторым, несмотря на то что результат будет безопасным типом. Вариативность позволяет присвоить второй делегат переменной типа Action<Derived>, но делегаты можно объединять, только если их типы точно совпадают.
- Hint: Ковариантность (**out**) – для параметров типа, которые используются только как выходные параметры, контравариантность (**in**) – для "--" входных параметров.

Ковариантность и Контравариантность. Пример

```
public class Type1 { }
public class Type2 : Type1 { }
public class Type3 : Type2 { }

public class Program {
    public static Type3 MyMethod(Type1 t) {
        return t as Type3 ?? new Type3();
    }

    static void Main() {
        Func<Type1, Type3> f0 = MyMethod;
        Func<Type2, Type2> f1 = f0;
        // Ковариантный тип возвр. значения и контравариантный тип параметра.
        Func<Type3, Type1> f2 = f1;
        Type1 t1 = f2(new Type3());
    }
}
```