ЛЕКЦИЯ 9

- Модуль 3
- 07.02.2024
- Обобщения

ЦЕЛИ ЛЕКЦИИ

- Познакомиться с концепцией обобщений в программировании
- Изучить базовый синтаксис обобщённых типов на примере классов
- Обсудить некоторые ограничения на обобщённые параметры

ОБОБЩЕННОЕ ПРОГРАММИРОВАНИЕ

- **Обобщённое программирование** [Generic Programming] - стиль программирования, при котором алгоритм описывается в терминах отложенного определения типов (данных); когда указанный алгоритм используется для конкретных данных, переданных как параметры, происходит его **инстанцирование**

ОБОБЩЁННОЕ ПРОГРАММИРОВАНИЕ В C#

При написании программ часто возникает необходимость обобщить функционал для различных типов, чтобы

- Не возникало дублирование кода
- Обеспечивалась типовая безопасность

Варианты обобщения (генерализации):

- наследование
- обобщённые типы (Generic types), которые позволяют использовать параметры типов

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

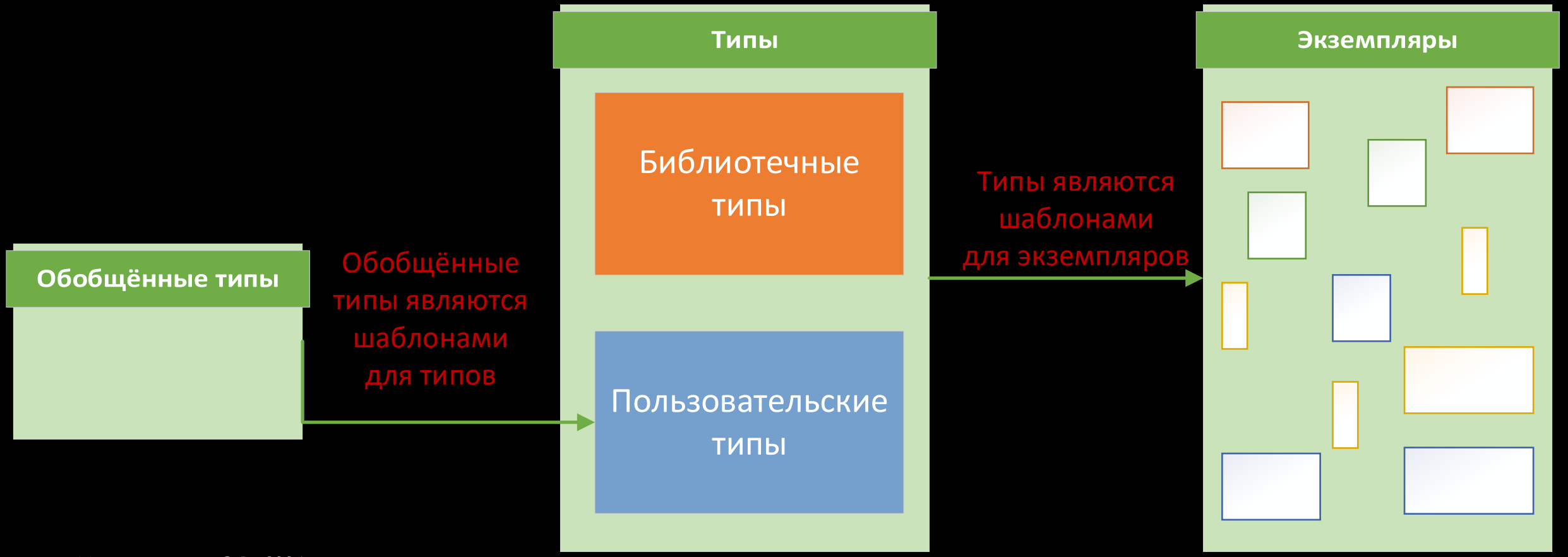
ПЛЮСЫ ИСПОЛЬЗОВАНИЯ ОБОБЩЕНИЙ

- упрощение программного кода
- обеспечение типовой безопасности
- исключение необходимости явного приведения типа при преобразовании типов данных
- повышение производительности

ПРИМЕНИМОСТЬ ОБОБЩЕНИЙ

- классы
- структуры
- интерфейсы
- методы
- делегаты

СХЕМА ТИПОВ С УЧЁТОМ ОБОБЩЕНИЙ



СИНТАКСИС ОБОБЩЁННОГО КЛАССА

Описание класса

```
class ClassIdentifier <T1, T2, …., TN> {
{
  // реализация членов класса, в т.ч. работающих членами T1,…,TN
}
```

Имя обобщённого класса

Имена обобщённых типов, используемых в классе

Инстанцирование класса

```
ClassIdentifier <t1, t2, …, tN> refName =
                    new ClassIdentifier<t1, t2, …, tN>(args);
```

Ссылка на экземпляр типа

Вызов конструктора

Названия конкретных типов, выступающих параметрами

АРГУМЕНТЫ ТИПА И ПАРАМЕТРЫ ТИПА

Объявление обобщённого класса

```
class ClassName<T1, T2>
{
  // члены класса
}
```

Параметры типа

Закрытый (сконструированный) класс

```
ClassName<short, int>
```

Аргументы типа

Параметры типа – «заглушки», на место которых подставляются реальные типы
Аргументы типа – конкретные типы, подставляемые на место параметров типа

ОТКРЫТЫЕ И ЗАКРЫТЫЕ ОБОБЩЁННЫЕ ТИПЫ

Открыто-сконструированный тип

```
class ClassName<T1, T2>
{
  // члены класса
}
```

Закрыто-сконструированный тип

```
ClassName<short, int>
```

- На этапе выполнения существуют только обобщённые типы/методы, в которых выполнена подстановка параметров; такие типы называют закрытыми или закрытыми сконструированными
- Объявленные обобщённые типы без подстановки аргументов типов называют открытыми

ПРИМЕР. ОТКРЫТЫЙ СКОНСТРУИРОВАННЫЙ ТИП

```
public class NaiveStack<T>
{
    List<T> _items = new List<T>();
    public void Push(T value) => _items.Add(value);
    public T Pop()
    {
        if (_items.Count == 0)
        {
            throw new InvalidOperationException("Stack is empty!!!");
        }
        T elem = _items[_items.Count - 1];
        _items.RemoveAt(_items.Count - 1);
        return elem;
    }
}
```

Параметр типа T объявлен, однако аргумент типа для него не подставлен.

тип NaiveStack<T> – открытый

ПРИМЕР. ЗАКРЫТЫЙ СКОНСТРУИРОВАННЫЙ ТИП

C# не позволяет опустить треугольные скобки при использовании конструктора.

```
NaiveStack<int> intStack = new NaiveStack<int>();
NaiveStack<string> stringStack = new();
// Заполнение стека
        for(int i = 1; i  <= 10; i++)
        {
            intStack.Push(i);
            stringStack.Push(i.ToString());
        }
        Console.WriteLine($"Элемент целочисленного стека: {intStack.Pop()}");
        Console.WriteLine($"Элемент строкового стека: {stringStack.Pop()}");
```

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

Вывод:

```
Number: 5
Line: string5
```

СХЕМА СОЗДАНИЯ ЗАКРЫТОГО ТИПА

Объявление обобщённого класса

```
class ClassName <T1, T2>
{
    T1 _fieldOfT1;
    T2 _fieldOfT2;
}
```

Запускаем создание закрытого типа… где-то в коде

```
ClassName<short, int>
```

создаётся

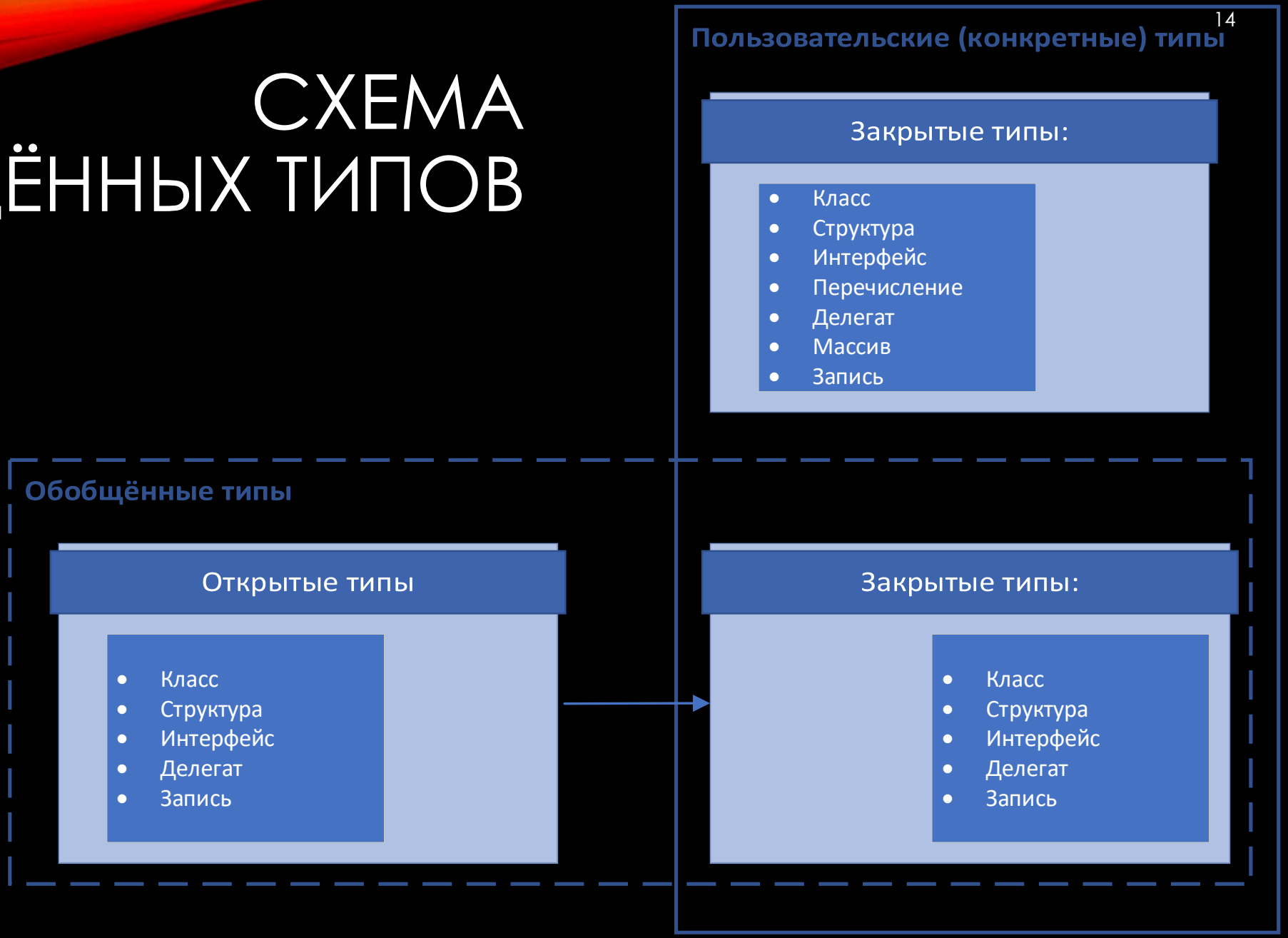
Сконструированный (закрытый) класс

```
class ClassName
<short, int>
{
    short _fieldOfT1;
    int _fieldOfT2;
}
```

Закрытый (сконструированный) тип формируется путём подстановки short на место T1 и int на место T2

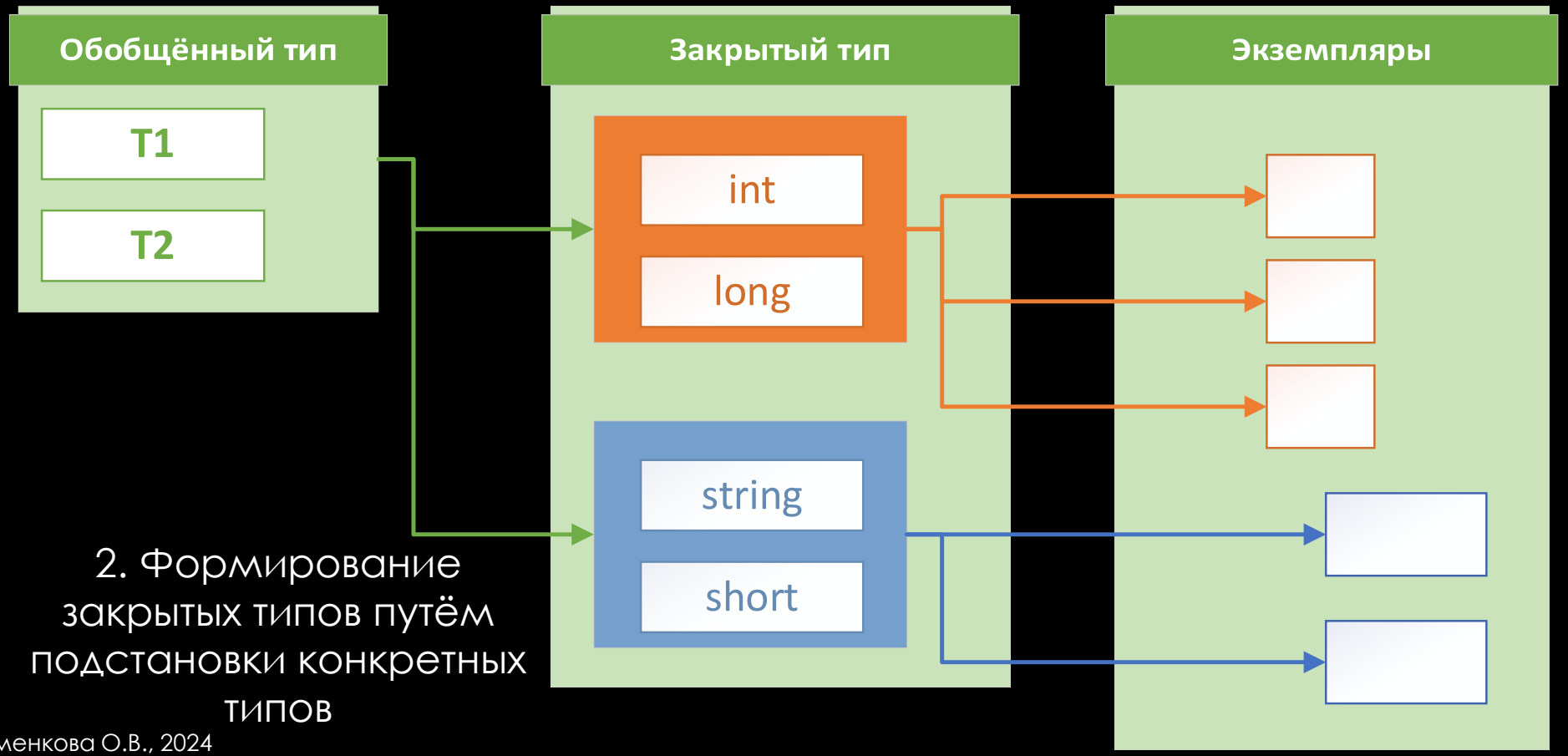
Подстановка аргументов типов на место параметров типов позволяет избежать упаковки/распаковки при выполнении операций

СХЕМА ОБОБЩЁННЫХ ТИПОВ



СОЗДАНИЕ ЭКЗЕМПЛЯРОВ ИЗ ОБОБЩЁННЫХ ТИПОВ

1. Объявление параметра типа



2. Формирование закрытых типов путём подстановки конкретных типов

3. Создание экземпляров закрытых типов.

ЗАКРЫТЫЕ КЛАССЫ СО СТАТИЧЕСКИМИ ЧЛЕНАМИ

```
class ClassName<T1, T2>
{
    static T1 _field1;
    T2 _field2;
}
```

```
...
var first = new ClassName<short, int> ();
...
```

```
...
var second = new ClassName<int, long> ();
...
```

```
class ClassName<short, int>
{
    static short _field1;
    int _field2;
}
```

```
class ClassName<int, long>
{
    static int _field1;
    long _field2;
}
```

Различные закрытые классы независимы друг от друга. У каждого из них будет свой независимый набор статических полей

ДОПУСТИМЫЕ ОПЕРАЦИИ НАД ПАРАМЕТРАМИ ТИПОВ

- Обобщённые типы по умолчанию требуют, чтобы все выполняемые операции были допустимы для любых подставляемых аргументов на этапе компиляции
- 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

Информацию о допустимых операциях для обобщённых типов указывают при помощи ограничений

ОГРАНИЧЕНИЯ ПАРАМЕТРОВ ТИПОВ (1)

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

? – добавление этого символа позволяет использовать типы, допускающие null

ОГРАНИЧЕНИЯ ПАРАМЕТРОВ ТИПОВ (2)

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

Несовместимые друг с другом ограничения параметров типов:

- class, struct, unmanaged, notnull, default

ПОРЯДОК ОГРАНИЧЕНИЙ ПАРАМЕТРОВ ТИПОВ

Первичное ограничение
(0 или 1)

- **class**
- **struct**
- **unmanaged**
- **notnull**
- **ClassName**
- **default**

Вторичные ограничения
(0 или более)

- **InterfaceName1, InterfaceName2, …**

Ограничение на конструктор
(0 или 1)

- **new()**

Неверный порядок ограничений приводит к ошибке компиляции

СИНТАКСИС ОГРАНИЧЕНИЙ ПАРАМЕТРОВ ТИПОВ

Для указания ограничений используется контекстно-ключевое слово **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()
{ }
```

ОГРАНИЧЕНИЯ НА КОНСТРУКТОР WHERE T: NEW()

Смысл ограничения: безошибочное создание объекта типа T внутри обобщённого типа

Ограничение не позволяет использовать в качестве обобщенного типа класс, в котором отсутствует конструктор без параметров или умалчиваемый конструктор

```
class DemoGenericClass<T> where T : new()
{
    T internalObject;
    public DemoGenericClass() => internalObject = new T();
}
```

Умалчиваемый конструктор

```
class Number
{
    public int X { get; set; }
}
```

```
DemoGenericClass<Number> demo = new DemoGenericClass<Number>();
DemoGenericClass<BadNumber> d1 = new DemoGenericClass<BadNumber>();
```

```
class BadNumber
{
    public int X { get; set; }
    public BadNumber(int x) { X = x; }
}
```

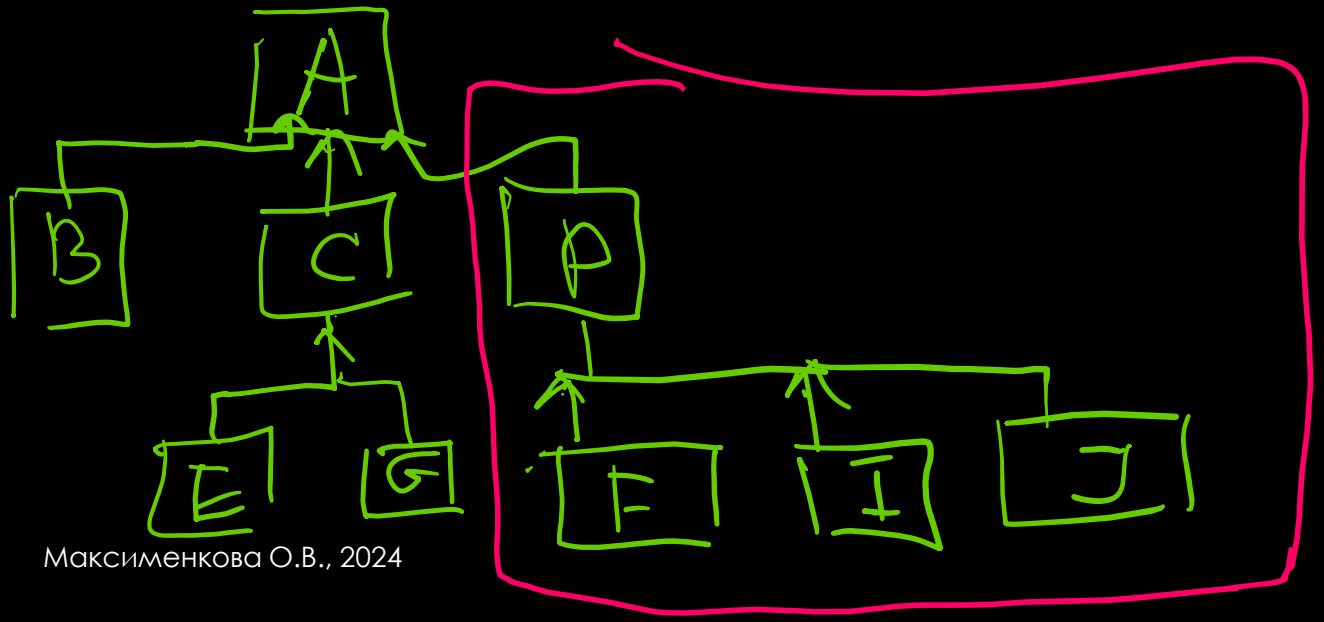
Нет конструктора без параметров

Исходный код (https://replit.com/@olgamaksimenkova/GenericsWhereNew)

ОГРАНИЧЕНИЯ НА РОДИТЕЛЬСКИЙ КЛАСС WHERE T : BASENAME

Смысл ограничения: лимитировать множество классов, которые могут использоваться в качестве аргумента обобщённого типа

Ограничение не позволяет использовать в качестве обобщенного типа класс, не входящий в иерархию от BaseName класса. Попытка подсунуть класс из другой иерархии приводит к ошибке компиляции




```
class DemoGenericClass<T> where T : D
{
    //…
}
```

ОГРАНИЧЕНИЯ НА РОДИТЕЛЬСКИЙ КЛАСС. ПРИМЕР

```
class A { }

class B : A { }

class C : A { }
class D : B { }
```

Иерархия классов

```
class DemoGenericClass<T1, T2> where T1 : T2
{
    //
}
```

Ограничение на отношения типов

```
// Создание экземпляра класса: GenClass<T1, T2> where T1:T2.
DemoGenericClass<B, A> obj = new DemoGenericClass<B, A>(); // Допустимо.
DemoGenericClass<D, A> ex = new DemoGenericClass<D, A>(); // Допустимо.
// Класс A наследник класса B.
// DemoGenericClass<A, B> obj = new DemoGenericClass<A, B>(); // Ошибка компиляции.

// C не наследник класса B.
// DemoGenericClass<C, B> obj2 = new DemoGenericClass<C, B>(); // Ошибка компиляции.

// Класс B не наследник класса C.
// DemoGenericClass<B, C> obj3 = new DemoGenericClass<B, C>(); // Ошибка компиляции.
```

ОГРАНИЧЕНИЕ НА ИНТЕРФЕЙС WHERE T : INTERFACENAME

Смысл ограничения: гарантировать наличие у типизирующего параметра определённых методов или его принадлежность к конкретному типу интерфейса

Ограничение не позволяет использовать в качестве типизирующих параметров типы, не реализующие все члены соответствующего интерфейса. При отсутствии у типа реализации хотя бы одного интерфейса происходит ошибка компиляции

ОГРАНИЧЕНИЕ НА ИНТЕРФЕЙС. ПРИМЕР (1)

```
class Point : IComparable<Point>
{
    public double X { get; set; }
    public double Y { get; set; }

    public Point(double x, double y) => (X, Y) = (x, y);
    public int CompareTo(Point anotherPoint)
    {
        if (anotherPoint == null) return 1;
        if (this.DistanceToZero() == anotherPoint.DistanceToZero()) return 0;

        if(this.DistanceToZero() > anotherPoint.DistanceToZero()) return -1;

        return 1;
    }
    private double DistanceToZero() => Math.Sqrt(X * X + Y * Y);

    public override string ToString() => $"({X};{Y})";
}
```

Класс реализует интерфейс

Реализован метод CompareTo()

ОГРАНИЧЕНИЕ НА ИНТЕРФЕЙС. ПРИМЕР (2)

```
class Pair<T> where T : IComparable<T> {
    T _first;
    T _second;

    public Pair(T x, T y)
    {
        if(x.CompareTo(y) > 0)  {
            _first = x;
            _second = y;
        }
        else  {
            _first = y;
            _second = x;
        }
    }
    public override string ToString()
    {
        return $"{_first} {_second}" ;
    }
}
```

Типы, не реализующие интерфейс Icomparable<T> не подойдут

Мы сравниваем объекты общего типа, нужна гарантия наличия этой реализации

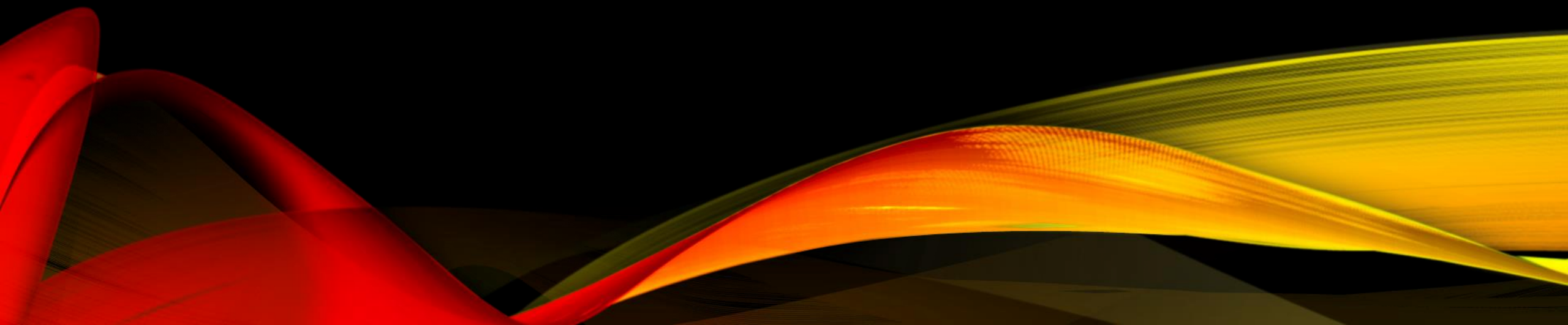
```
Point point1 = new Point(1, 1);
Point point2 = new Point(2, 2);
Pair<Point> pp = new Pair<Point>(point2, point1);
Pair<Point> pp2 = new Pair<Point>(point1, point2);
Console.WriteLine(pp);
Console.WriteLine(pp2);
```

Вывод:

```
(1;1) (2;2)
(1;1) (2;2)
```

Исходный код (https://replit.com/@olgamaksimenkova/GenericWhereInterface)

НАСЛЕДОВАНИЕ ОБОБЩЕННЫХ ТИПОВ



СИНТАКСИС НАСЛЕДОВАНИЯ ОБОБЩЁННЫХ КЛАССОВ

Базовый класс

Идентификатор базового класса

```
class BaseClassId<T1, T2, ..., TN>
{
  ...
}
```

Класс наследник

Идентификатор класса наследника

```
class DerivedClassId<T1, T2, ..., TN> : BaseClassId<T1, T2, ..., TN>
{
  ...
}
```

НАСЛЕДОВАНИЕ ОБОБЩЁННЫХ ТИПОВ (1)

Обобщённые типы могут наследоваться как от необобщённых типов, так и от других открытых и закрытых сконструированных:

```
// Открытый → закрытый сконструированный:
public class GenericDerived2<T> : GenericBase<int> { }

// Открытый сконструированный → необобщённый:
public class GenericDerived3<T> : NonGenericBase { }
```

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

// Открытый → открытый:
public class GenericDerived1<U> : GenericBase<U> { }
```

НАСЛЕДОВАНИЕ ОБОБЩЁННЫХ ТИПОВ (2)

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

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

// Необобщённый → закрытый сконструированный:
public class NonGenericDerived : GenericBase<byte> { }
```

Наследование от обобщённых типов с несколькими параметрами требует 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> { }
```

ТРЕБОВАНИЯ К НАСЛЕДНИКУ ОБОБЩЁННОГО КЛАССА

- Обобщённые параметры <T1, T2, …, TN> базового класса передаются в класс наследник
  - Наследник обобщённого класса – тоже обобщённый класс
  - Класс наследник может дополнять перечень обобщённых параметров базового типа
- Все конструкторы базового класса, получающие параметры обобщённого типа необходимо реализовать в наследнике с прямым вызовом конструкторов базового класса

ПРИМЕР 1. НАСЛЕДОВАНИЕ ОБОБЩЁННОГО ТИПА

```
class BaseClass<T> {
    public T BaseClassState { get; set; }
    public BaseClass(T obj) => BaseClassState = obj;
}
```

В базовом классе присутствует конструктор с параметром T

```
class InheritedClass<T> : BaseClass<T> {
    T InheritedClassState { get; set; }

    public InheritedClass(T objBC, T objIC) : base(objBC) => InheritedClassState = objIC;
    public override string ToString() {
        return InheritedClassState.ToString();
    }
}
```

Обязательно переопределяем в наследнике и передаём в родительский конструктор параметр!

```
InheritedClass<int> obj = new InheritedClass<int>(5, 6);
Console.WriteLine($"inherited class state:: {obj}");
```


Исходный код (https://replit.com/@olgamaksimenkova/GenericInheritConstructor)

ПРИМЕР 2. НАСЛЕДОВАНИЕ ОБОБЩЁННОГО ТИПА

```
class BaseClass<T1, T2, T3> {
    public T1 FieldT1 { get; private set; }
    public T2 FieldT2 { get; private set; }
    public T3 FieldT3 { get; private set; }

    public BaseClass(T1 op1, T2 op2, T3 op3) => (FieldT1, FieldT2, FieldT3) = (op1, op2, op3);
}
```

Три обобщённых параметра

```
class InheritedClass<T1, T2, T3, T4> : BaseClass<T1, T2, T3> {
    public T4 FieldT4 { get; private set; }

    public InheritedClass(T1 op1, T2 op2, T3 op3, T4 op4) : base(op1, op2, op3) => FieldT4 = op4;
}
```

Наследник добавляет еще один обобщённый параметр

```
InheritedClass<int, double, char, bool> obj =
        new InheritedClass<int, double, char, bool>(15, 3.14, 'H', true);

Console.WriteLine(obj.FieldT1);
Console.WriteLine(obj.FieldT2);
Console.WriteLine(obj.FieldT3);
Console.WriteLine(obj.FieldT4);
```

Исходный код (https://replit.com/@olgamaksimenkova/GenericInheritAddType)

ИСПОЛЬЗОВАННАЯ И РЕКОМЕНДОВАННАЯ ЛИТЕРАТУРА

- https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters
- https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-methods