Частное учреждение образования

«Колледж бизнеса и права»

|  |
| --- |
| УТВЕРЖДАЮ  Ведущий методист колледжа  \_\_\_\_\_\_\_\_\_\_\_\_\_Е.В.Паскал  «\_\_\_ » \_\_\_\_\_\_\_\_\_ 20\_\_ |

|  |  |
| --- | --- |
| Специальность: 2-40 01 01 «Программное обеспечение информационных технологий» | Учебная практика по программированию |

**Практическая работа №11**

**Инструкционно-технологическая карта**

Тема:Обобщения

* Обобщенные типы. Обобщенные методы (2 часа)
* Роль параметров обобщенных типов. Ограничения обобщений (2 часа)
* Ковариантность. Контравариантность. Сравнение обобщений и шаблонов С++ (2 часа)

Цель: Закрепить умения и навыки работы с обобщениями. Научиться создавать обобщенные методы и обобщенные типы: делегаты, структуры, классы, интерфейсы и применять их на практике

Время выполнения: 6 часов

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическими сведениями.
2. Выполнить рассмотренные в теоретическом материале примеры.
3. Получить у преподавателя вариант задания и выполнить его.
4. Оформить выполненное задание в отчет.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В C# имеются два отдельных механизма для написания кода, повторно используемого различными типами: *наследование* и *обобщения* (generics). В то время как наследование выражает повторное использование с помощью базового типа, обобщения делают это посредством «шаблона», который содержит «типы-заполнители». Обобщения предназначены для записи кода, который может многократно использоваться различными типами. Обобщения, по сравнению с наследованием, могут *увеличивать безопасность типов*, а также *сокращать количество приведений и упаковок.*

**Обобщенные типы**

Обобщенный тип объявляет *параметры типа* – типы-заполнители, предназначенные для замещения пользователем обобщенного типа, которые передаются как *аргументы типа*. Ниже показан обобщенный тип Stack<T>, предназначенный для реализации стека экземпляров типа T. В Stack<T> объявлен единственный параметр типа T:

public class Stack<T>

{

int position;

T[] data = new T[100];

public void Push(T obj) => data[position++] = obj;

public T Pop() => data[--position];

}

Использовать Stack<T> можно следующим образом:

var stack = new Stack<int>();

stack.Push(5);

stack.Push(10);

int x = stack.Pop(); // x равно 10

int y = stack.Pop(); // y равно 5

Обратите внимание, что в последних двух строках кода приведение вниз не требуется. Это позволяет избежать возможной ошибки во время выполнения и устраняет непроизводительные затраты на упаковку/ распаковку. В результате данный обобщенный стек получает преимущество над необобщенным стеком, в котором вместо T используется тип object.

Класс Stack<int> заменяет параметр типа T аргументом типа int, неявно создавая тип «на лету» (синтез происходит во время выполнения). Фактически Stack<int> имеет показанное ниже определение (подстановки выделены полужирным, а во избежание путаницы вместо имени класса указано ###):

public class ###

{

int position;

int[] data = new int[100];

public void Push(int obj) => data[position++] = obj;

public int Pop() => data[--position];

}

Формально мы говорим, что Stack<T> – это *открытый (open) тип*, а Stack<int> – *закрытый (closed) тип*. Во время выполнения все экземпляры обобщенных типов являются закрытыми – с соответствующей заменой типов-заполнителей.

**Обобщенные методы**

Обобщенный метод объявляет параметры типа внутри сигнатуры метода. С помощью обобщенных методов многие фундаментальные алгоритмы могут быть реализованы единственным универсальным способом. Ниже показан обобщенный метод, который меняет местами содержимое двух переменных любого типа T:

static void Swap<T>(ref T a, ref T b)

{

T temp = a; a = b; b = temp;

}

Метод Swap<T> можно использовать следующим образом:

int x = 5, y = 10;

Swap(ref x, ref y);

Как правило, предоставлять аргументы типа обобщенному методу нет нужды, поскольку компилятор способен вывести их самостоятельно. Если же имеется неоднозначность, то обобщенные методы могут быть вызваны с аргументами типа:

Swap<int>(ref x, ref y);

Внутри обобщенного типа метод не рассматривается как обобщенный до тех пор, пока он сам не вводит параметры типа (посредством синтаксиса с угловыми скобками). Метод Pop() в рассматриваемом обобщенном стеке просто использует существующий параметр типа T, а потому и не классифицируется как обобщенный.

Методы и типы – единственные конструкции, в которых могут вводиться параметры типа. Свойства, индексаторы, события, поля, конструкторы, операции и так далее не могут объявлять параметры типа, хотя способны пользоваться любыми параметрами типа, которые уже объявлены во включающем типе. В примере с обобщенным стеком можно было бы написать индексатор, который возвращает обобщенный элемент:

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

Аналогично конструкторы также могут пользоваться существующими параметрами типа, но не могут их вводить.

**Роль параметров обобщенных типов**

Параметры типа могут быть введены в объявлениях классов, структур, интерфейсов, делегатов и методов. Можно указывать несколько параметров типа, разделяя их запятыми:

class Dictionary<TKey, TValue> {...}

Вот как он инстанцируется:

var myDict = new Dictionary<int, string>();

Имена обобщенных типов и методов могут быть перегружены при условии, что количество параметров типа у них различается. Например, показанные ниже три имени типов не конфликтуют друг с другом:

class A { }

class A<T> { }

class A<T1, T2> { }

По соглашению обобщенные типы и методы с единственным параметром типа обычно именуют его как T, если назначение параметра очевидно. В случае нескольких параметров типа каждый такой параметр имеет более описательное имя (с префиксом T).

**Оператор typeof и несвязанные обобщенные типы**

Во время выполнения открытых обобщенных типов не существует: они закрываются как часть компиляции. Однако во время выполнения возможно существование несвязанного (unbound) обобщенного типа — исключительно как объекта Type. Единственным способом указания несвязанного обобщенного типа в C# является применение оператора typeof :

class A<T> { }

class A<T1, T2> { }

Type a1 = typeof(A<>); // несвязанный тип

Type a2 = typeof(A<,>); // указывает на 2 аргумента типа

Console.Write(a2.GetGenericArguments().Count()); // 2

Оператор typeof можно использовать также для указания закрытого типа:

Type a3 = typeof(A<int, int>);

или открытого типа (который закрыт во время выполнения):

class B<T> { void X() { Type t = typeof(T); } }

**Обобщенное значение по умолчанию**

Ключевое слово default может применяться для получения значения параметра типа обобщения по умолчанию. Значением по умолчанию для ссылочного типа является null, а для типа-значения — результат побитового обнуления полей в этом типе:

static void Zap<T>(T[] array)

{

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

array[i] = default(T);

}

Начиная с версии C# 7.1, аргумент типа можно не указывать в случаях, когда компилятор способен его вывести самостоятельно:

array[i] = default;

**Ограничения обобщений**

По умолчанию параметр типа может быть замещен любым типом. Чтобы затребовать более специфичные аргументы типа, к параметру типа можно применить ограничения . Существуют восемь видов ограничений:

where T : base-class // ограничение базового класса

where T : interface // ограничение интерфейса

where T : class // ограничение ссылочного типа

where T : class? // (см. ссылочные типы, допускающие значение null)

where T : struct // ограничения типа-значения

where T : unmanaged // ограничения неуправляемого типа

where T : new() // ограничения конструктора без параметров

where U : T // неприкрытое ограничение типа

where T : notnull // тип-значение или ссылочный тип, которые не

// могут быть null

В следующем примере GenericClass<T,U> требует, чтобы тип T был производным от класса SomeClass (или идентичен ему) и реализовывал интерфейс Interface1, а тип U имел конструктор без параметров:

class SomeClass { }

interface Interface1 { }

class GenericClass<T, U> where T : SomeClass, Interface1

where U : new()

{ ... }

Ограничения могут применяться везде, где определены параметры типа, как в методах, так и в определениях типов.

*Ограничение базового класса* указывает, что параметр типа должен быть подклассом заданного класса (или совпадать с ним); *ограничение интерфейса* указывает, что параметр типа должен реализовывать данный интерфейс. Такие ограничения позволяют экземплярам параметра типа быть неявно преобразуемыми в этот класс или интерфейс.

*Ограничение class* и *ограничение struct* указывают, что T должен быть ссылочным типом или типом-значением (не допускающим null) соответственно. *Ограничение unmanaged* является более сильной версией ограничения struct: тип T должен быть простым типом-значением или структурой, которая (рекурсивно) не имеет ссылочных типов. *Ограничение конструктора без параметров* требует, чтобы тип T имел открытый конструктор без параметров и позволял вызывать операцию new() для T:

static void Initialize<T>(T[] array) where T : new()

{

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

array[i] = new T();

}

*Неприкрытое ограничение типа* требует, чтобы один параметр типа был производным от другого параметра типа (или совпадал с ним).

**Подклассы обобщенных типов**

Подклассы для обобщенного класса можно создавать точно так же, как в случае необобщенного класса. Подкласс может оставлять параметры типа базового класса открытыми, как показано в следующем примере:

class Stack<T> {...}

class SpecialStack<T> : Stack<T> {...}

Либо же подкласс может закрыть параметры обобщенного типа с помощью конкретного типа:

class IntStack : Stack<int> {...}

Подкласс может также вводить новые аргументы типа:

class List<T> {...}

class KeyedList<T, TKey> : List<T> {...}

**Самоссылающиеся обобщенные объявления**

При закрытии аргумента типа тип может указывать в качестве конкретного типа самого себя:

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

public class Balloon : IEquatable<Balloon>

{

public bool Equals(Balloon b) { ... }

}

Следующий код также корректен:

class Foo<T> where T : IComparable<T> { ... }

class Bar<T> where T : Bar<T> { ... }

**Статические данные**

Статические данные являются уникальными для каждого закрытого типа:

class Bob<T> { public static int Count; }

...

Console.WriteLine(++Bob<int>.Count); // 1

Console.WriteLine(++Bob<int>.Count); // 2

Console.WriteLine(++Bob<string>.Count); // 1

Console.WriteLine(++Bob<object>.Count); // 1

**Ковариантность**

Ковариантность и контравариантность – сложные концепции. Мотивация, лежащая в основе их введения в язык C#, заключалась в том, чтобы позволить обобщенным инерфейсам и обобщениям (в частности, определенным в .NET, таким как IEnumerable<T>) работать более предсказуемым образом.

Предполагая, что тип A может быть преобразован в B, тип X имеет ковариантный параметр типа, если X<A> преобразуется в X<B>.

Согласно понятию вариантности в C#, «преобразуется/подается преобразованию» означает возможность преобразования через неявное ссылочное преобразование – такое, как когда A является подклассом B или A реализует B. Сюда не входят числовые преобразования, упаковывающие преобразования и пользовательские преобразования.

Например, тип IFoo<T> имеет ковариантный тип T, если корректен следующий код:

IFoo<string> s = ...;

IFoo<object> b = s;

Интерфейсы (и делегаты) допускают ковариантные параметры типа. В целях иллюстрации предположим, что класс Stack<T>, который был представлен выше, реализует показанный ниже интерфейс:

public interface IPoppable<out T> { T Pop(); }

Модификатор out для T указывает, что тип T используется только в выходных позициях (например, в возвращаемых типах методов) и помечает параметр типа как ковариантный, разрешая написание такого кода:

// полагая, что Bear является подклассом Animal:

var bears = new Stack<Bear>();

bears.Push(new Bear());

// так как bears реализукт IPoppable<Bear>,

// его можно конвертировать в IPoppable<Animal>:

IPoppable<Animal> animals = bears; // корректно

Animal a = animals.Pop();

Приведение bears к animals разрешено компилятором – в силу того, что параметр типа в интерфейсе является ковариантным.

Интерфейсы IEnumerator<T> и IEnumerable<T> помечены как имеющие ковариантный тип T. Это позволяет, например, приводить IEnumerable<string> к IEnumerable<object>.

Компилятор генерирует ошибку, если ковариантный параметр типа встречается во *входной позиции* (скажем, в параметре метода или в записываемом свойстве). Цель такого ограничения – гарантировать безопасность типов на этапе компиляции. Например, оно предотвращает добавление к этому интерфейсу метода Push(T), который пользователи могли бы неправильно применять для внешне безобидной операции внесения в стек объекта, представляющего верблюда, в реализацию IPoppable<Animal> (вспомните, что базовым типом в нашем примере является стек медведей). Чтобы можно было определить метод Push(T), параметр типа T в действительности должен быть контравариантным.

В языке C# ковариантность (и контравариантность) поддерживается только для элементов со ссылочными, но не упаковывающими преобразованиями. Таким образом, если имеется метод, который принимает параметр типа IPoppable<object>, то его можно вызывать с IPoppable<string>, но не с IPoppable<int>.

**Контравариантность**

Как было отмечено ранее, если предположить, что A разрешает неявное ссылочное преобразование в B, то тип X имеет ковариантный параметр типа, когда X<A> допускает ссылочное преобразование в X<B>. Тип будет контравариантным, если возможно преобразование в обратном направлении – из X<B> в X<A>. Контравариантность поддерживается интерфейсами и делегатами, когда параметр типа встречается только во входных позициях, обозначаемых с помощью модификатора in. Продолжая предыдущий пример, если класс Stack<T> реализует интерфейс

public interface IPushable<in T> { void Push(T obj); }

то вполне корректно поступать так:

IPushable<Animal> animals = new Stack<Animal>();

IPushable<Bear> bears = animals; // корректно

bears.Push(new Bear());

Зеркально отражая ковариантность, компилятор сообщит об ошибке, если вы попытаетесь использовать контравариантный параметр типа в выходной позиции (например, в качестве возвращаемого значения или в читаемом свойстве).

**Сравнение обобщений С# и шаблонов C++**

Обобщения C# в использовании похожи на шаблоны C++, но работают они совершенно по-другому. В обоих случаях должен осуществляться синтез между поставщиком и потребителем, при котором типы-заполнители заполняются потребителем. Однако в ситуации с обобщениями C# типы поставщика (т.е. открытые типы вроде List<T>) могут быть скомпилированы в библиотеку (такую как mscorlib. dll). Дело в том, что собственно синтез между поставщиком и потребителем, который создает закрытые типы, в действительности не происходит вплоть до времени выполнения. Для шаблонов C++ такой синтез производится на этапе компиляции. Это значит, что в C++ развертывать библиотеки шаблонов как сборки .dll не получится – они существуют только в виде исходного кода. Вдобавок также затрудняется динамическое инспектирование параметризированных типов, не говоря уже об их создании на лету.

Чтобы лучше понять, почему сказанное справедливо, взглянем на метод Мах в C# еще раз:

static Т Max<Т>(Т а, Т b) where Т : IСоmрагаble<Т>

=> a.CompareTo(Ь) > 0 ? а : b;

Почему бы ни реализовать этот метод следующим образом:

static Т Мах<Т>(Т а, Т Ь)=>(a > b ? a : b);// Ошибка на этапе компиляции

Причина в том, что метод Мах должен быть скомпилирован один раз, но работать для всех возможных значений Т. Компиляция не может пройти успешно ввиду отсутствия единого смысла операции > для всех значений Т — на самом деле, операция > может быть доступна далеко не в каждом типе Т. Для сравнения ниже показан код того же метода Мах, написанный с применением шаблонов C++. Этот код будет компилироваться отдельно для каждого значения Т, пользуясь семантикой > для конкретного типа Т и приводя к ошибке на этапе компиляции, если отдельный тип Т не поддерживает операцию >:

template<class Т> Т Мах(Т а, Т Ь)

{

return а > b ? а : b;

}

ВАРИАНТЫ ЗАДАНИЙ

**Задание 1**

Разработать обобщенный класс, позволяющий работать с данными на примере стека/очереди. Продемонстрировать его работу.

**Задание 2**

Общее задание. Создать заданный в варианте обобщенный класс. Определить в классе индексаторы, конструкторы, деструктор, необходимые функции и заданные перегруженные операции. В методах класса для работы с аргументами используйте ref‐ и out‐параметры и операторы is или as, foreach. Используйте обработку исключительных ситуаций.

Индивидуальное задание (по вариантам)

|  |  |
| --- | --- |
| Вариант | Задание |
| 1 | Класс – одномерный массив. Дополнительно перегрузить следующие операции: \* – умножение массивов; int() – преобразование; ==, != – проверка на равенство, неравенство; <=, >= – сравнение |
| 2 | Класс – одномерный массив. Дополнительно перегрузить следующие операции: == – проверка на равенство; != – проверка на неравенство; ++ – инкремент массивов; ‐‐– декремент массивов |
| 3 | Класс – множество Сset. Дополнительно перегрузить следующие операции: + – добавить элемент в множество (типа set + item); \* – пересечение множеств, ==, != – проверка множеств на равенство, неравенство |
| 4 | Класс – очередь Сqueue. Дополнительно перегрузить следующие операции: + – добавить элемент; ‐ – извлечь элемент; bool () – проверка, пустая ли очередь; < – копирование одной очереди в другую с сортировкой в убывающем порядке; >– копирование одной очереди в другую с сортировкой в возрастающем порядке |
| 5 | Класс – множество Сset. Дополнительно перегрузить следующие операции: + – объединение множеств; <=, >= – сравнение множеств; int ()– мощность множества; &– пересечение |
| 6 | Класс – однонаправленный список list. Дополнительно перегрузить следующие операции: + – объединить два списка; ‐ – удалить элемент; ==, != – проверка на равенство, неравенство; ~– проверка, пустой ли список |
| 7 | Класс – матрица. Дополнительно перегрузить следующие операции: + – сложение матриц; () – доступ по индексу M(с, i); >, < – сравнение матриц; == – копирование матриц, != – проверка соответствия размеров матриц |
| 8 | Класс – время CVector. Дополнительно перегрузить следующие операции: + – добавление элемента; ++, ‐‐ – увеличение и уменьшения всех элементов вектора на один (СVector++); !=, == – проверка на неравенство, равенство |
| 9 | Класс – время CVector. Дополнительно перегрузить следующие операции: ‐ – разность двух векторов; + – сложение элементов вектора с целым; !=, == - проверка на неравенство, равенство |
| 10 | Класс – стек Сstack. Дополнительно перегрузить следующие операции: + – добавить элемент в стек; ‐ – извлечь элемент из стека; ~ – проверка, пустой ли стек; > и < – копирование одного стека в другой с сортировкой в возрастающем порядке |

ИНФОРМАЦИОННЫЕ ИСТОЧНИКИ

1. Албахари, Дж. C# 7.0. Справочник. Полное описание языка / Дж. Албахари, Б. Албахари. – СПб. : ООО «Альфа-книга», 2018. – 1024 с. : ил.
2. Прайс, Марк. C# 8 и .NET Core. Разработка и оптимизация / Марк Прайс. – СПб.: Питер, 2021. – 816 с.: ил.
3. Стиллмен, Эндрю. Изучаем C# / Эндрю Стиллмен, Дженнифер Грин. – 3-е изд. – СПб.: Питер, 2020. – 816 с.: ил.
4. Троелсен, Э. Язык программирования C# 7 и платформы .NET и .NET Core / Э. Троелсен, Ф. Джепикс. – 8-е изд. – СПб. : ООО «Диалектика», 2018. – 1328 с.: ил.
5. Шарп, Дж. Microsoft Visual C#. Подробное руководство / Дж. Шарп. – 8-е изд. – СПб.: Питер, 2017. – 848 с.: ил.

Преподаватель Е.В. Багласова

|  |
| --- |
| Рассмотрено на заседании цикловой  комиссии программного обеспечения информационных технологий №10  Протокол №\_\_\_\_\_от \_\_\_\_\_\_\_\_\_\_\_\_\_\_  Председатель ЦК \_\_\_\_\_\_В.Ю.Михалевич |
|  |