Тема 15. Обобщения

Парадигмы программирования. C#

План

- Необходимость использования обобщений
- Основы обобщений
- Применение обобщений
- Ограничения обобщений
- Обобщения и приведение типов
- Обобщения и наследование
- Обобщённые методы
- Обобщённые делегаты
- Обобщённые коллекции и массивы
- Обобщения и рефлексия

Необходимость использования обобщений

Часть 1

Пример реализации нетипизированного стека

```
public class Stack
{
    private object[] items;
    public void Push(object item)
    {...}
    public object Pop()
    {...}
}
```

Пример использования нетипизированного стека

```
Stack stack = new Stack();
stack.Push(1);
stack.Push(2);
stack.Push("1");
string number =
(string)stack.Pop();
int number = (int)stack.Pop();
```

Недостатки использования нетипизированных контейнеров

- Низкая производительность
  - Упаковка данных значимых типов при занесении в контейнер
  - Распаковка данных значимых типов при извлечении из контейнера
  - Необходимость явного приведения типа при извлечении из контейнера
- Трудность выявления ошибок

```
Stack stack = new Stack();
stack.Push(1);
// Нет ошибки компиляции, но есть ошибка
// выполнения
string number = (string)stack.Pop();
```

- Решение проблем – разработка множества типизированных контейнеров

Типизированный стек для целых чисел

```
public class IntStack
{
    private int[] items;
    public void Push(int item){...}
    public int Pop(){...}
}
IntStack stack = new IntStack();
stack.Push(1);
int number = stack.Pop();
```

Типизированный стек для строк

```
public class StringStack
{
    private string[] items;
    public void Push(string item){...}
    public string Pop(){...}
}
StringStack stack = new
  StringStack();
stack.Push("1");
string number = stack.Pop();
```

Недостатки реализации множества типизированных контейнеров

- Значительное увеличение затрат на разработку
- Необходимость внесения множества изменений при обнаружении ошибки в реализованном шаблоне контейнера
- Невозможность реализовать типизированные контейнеры для ещё не разработанных типов данных

- Решение всех проблем – применение обобщений (generics)

Основы обобщений

Часть 2

Обобщения

- Обобщение – это шаблон для типа данных
  - Шаблон для класса
  - Шаблон для структуры
- Применение обобщений
  - У типа данных контейнера добавляется параметр обобщённого типа – тип данных элемента
  - Тип данных элемента записывается в угловых скобках «<тип данных элемента>»
  - Описание типа данных контейнера становится описанием множества типов данных
  - Конкретный тип данных выбирается из этого множества при объявлении переменной типа данных контейнера за счёт указания типа данных элемента контейнера

Обобщённый стек

```
public class Stack<T>
{
    private T[] items;
    public void Push(T item)
    {...}
    public T Pop()
    {...}
}

Stack<int> intStack = new Stack<int>();
intStack.Push(1);
intStack.Push(2);
int number = intStack.Pop();

Stack<string> stringStack = new Stack<string>();
stringStack.Push("String 1");
stringStack.Push("String 2");
string str = stringStack.Pop();
```

Сравнение обобщений в C++ и C#

- В классическом C++ компилятор заменяет объявления переменных обобщённых типов на дополнительные объявления типов с заменёнными параметрами
  - Это приводит к созданию множества одинаковых типов данных
- В C# и .NET тоже создаются объявления типов данных на основе параметров обобщённого типа данных
  - Простая подстановка параметра не используется
  - Ранее созданные типы используются повторно

Применение обобщений

Часть 3

Использование обобщённых структур

```
public struct Point<T>
{
    public T X;

    public T Y;
}

Point<int> point;
point.X = 1;
point.Y = 2;

Point<double> precisePoint;
precisePoint.X = 1.123;
precisePoint.Y = 2.234;
```

Связанный список

```
public class LinkedList<K,T>
{
    Node<K,T> head;

    public LinkedList()
    {
        head = new Node<K,T>();
    }
    public void AddHead(K key,T item)
    {
        Node<K,T> newNode = new Node<K,T>(
            key, item, head.NextNode);
        head.NextNode = newNode;
    }
}
```

Элемент связанного списка

```
class Node<K,T>
{
   public K Key { get; set; }
   public T Item { get; set; }
   public Node<K,T> NextNode { get; set; }

   public Node()
   {
      Key      = default(K);
      Item     = default(T);
      NextNode = null;
   }
   public Node(K key,T item,Node<K,T> nextNode)
   {
      Key      = key;
      Item     = item;
      NextNode = nextNode;
   }
}
```

Ключевое слово default

- Возвращает значение по умолчанию для значимых типов данных
  - Для числовых типов данных возвращает 0
  - Для структур создаёт новый экземпляр с элементами, инициализированными в 0
- Возвращает null для ссылочных типов данных

Применение обобщённого связанного списка

```
LinkedList<int,string> list = new
  LinkedList<int,string>();
list.AddHead(123,"AAA");

LinkedList<DateTime,string> list =
  new LinkedList<DateTime,string>();
list.AddHead(DateTime.Now,"AAA");
```

Ограничения обобщений

Часть 4

Пример необходимости ограничений обобщений

```
public class LinkedList<K, T>
{
    T Find(K key)
    {...}
    public T this[K key]
    {
        get { return Find(key); }
    }
}
...

LinkedList<int,string> list = new
  LinkedList<int,string>();

list.AddHead(123,"AAA");
list.AddHead(456,"BBB");
string item = list[456];
Debug.Assert(item == "BBB");
```

Реализация метода Find

```
T Find(K key)
{
    Node<K,T> current = head;
    while(current.NextNode != null)
    {
        // Ошибка компиляции: для типа данных K
        // оператор сравнения может быть
        // не определён
        if(current.Key == key)
            break;
        else
            current = current.NextNode;
    }
    return current.Item;
}
```

Решение проблемы возможного отсутствия оператора сравнения

- Применение интерфейса IComparable

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

- Замена проблемной строки в методе Find

```
if(current.Key.CompareTo(key) == 0)
```

- Для успешной компиляции необходимо явное приведение типа

```
if(((IComparable)(current.Key)).CompareTo(key) == 0)
```

- Получаем исходные проблемы, решаемые с помощью обобщений
  - Необходимость явного приведения типа
  - Необходимость упаковки и распаковки для значимых типов данных

Ограничения наследования

```
public class LinkedList<K, T> where K : IComparable
{
    T Find(K key)
    {
        Node<K, T> current = m_Head;
        while (current.NextNode != null)
        {
            if (current.Key.CompareTo(key) == 0)

                break;
            else

                current = current.NextNode;
        }
        return current.Item;
    }
    ...
}
```

Устранение упаковки/распаковки значимых типов в IComparable

- Добавлен обобщённый интерфейс

```
public interface IComparable<T>
{
    int CompareTo(T other);
    bool Equals(T other);
}
```

- Доработанное объявление связанного списка

```
public class LinkedList<K,T>
    where K : IComparable<K>
{...}
```

Дополнительные возможности ограничения наследования

- Ограничение на наследование от нескольких интерфейсов

```
public class LinkedList<K,T> where
  K : IComparable<K>, IConvertible
{...}
```

- Ограничения на несколько параметров

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

Запрещённые ограничения наследования

- Ограничение на наследование от нескольких базовых классов
- Ограничение на наследование от System.Delegate и System.Array
- Обобщённый тип не может иметь область видимости шире типа, использованного в качестве ограничения
- Ограничение на наследование от параметра

```
public class MyClass<T,U>
    where T : U
{...}
```

Ограничение конструктора

- Для создания объектов параметров обобщённого типа необходимо вызывать конструктор
- Для этого необходимо знать, какой конструктор есть у этого параметра

```
class Node<K, T> where T : new()
{
    public K Key;
    public T Item;
    public Node<K, T> NextNode;
    public Node()
    {
        Key = default(K);
        Item = new T();
        NextNode = null;
    }
}
```

Ограничение на значимый/ссылочный тип данных

- Ограничение на значимый тип

```
public class MyClass<T>
    where T : struct
{...}
```

- Ограничение на ссылочный тип

```
public class MyClass<T>
    where T : class
{...}
```

Обобщения и приведение типов

Часть 5

Неявное приведение типа параметра

```
interface ISomeInterface
{...}
class BaseClass
{...}
class MyClass<T>
    where T : BaseClass, ISomeInterface
{
   void SomeMethod(T t)
   {
      ISomeInterface obj1 = t;
      BaseClass      obj2 = t;
      object         obj3 = t;
   }
}
```

Явное приведение типа параметра

```
interface ISomeInterface
{...}
class SomeClass
{...}
class MyClass<T>
{
    void SomeMethod(T t)
    {
        // Допускается приведение к интерфейсу
        ISomeInterface obj1 =
            (ISomeInterface)t;
        // Приведение к классу не допускается
        SomeClass obj2 = (SomeClass)t;
    }
}
```

Приведение параметра к классу дополнительной переменной

```
class SomeClass
{...}

class MyClass<T>
{
    void SomeMethod(T t)
    {
        object temp = t;
        SomeClass obj = (SomeClass)temp;
    }
}
```

Аккуратное приведение типа (операторы is и as)

```
public class MyClass<T>
{
    public void SomeMethod(T t)
    {
        if(t is int)
        {...}

        if(t is LinkedList<int, string>)
        {...}

        string str = t as string;
        if(str != null)
        {...}

        LinkedList<int,string> list =
            t as LinkedList<int, string>;
        if(list != null)
        {...}
    }
}
```

Обобщения и наследование

Часть 6

Наследование от обобщённого типа

- Наследование обычного типа от обобщённого типа

```
public class BaseClass<T>
{...}
public class SubClass : BaseClass<int>
{...}
```

- Наследование обобщённого типа от другого обобщённого типа

```
public class SubClass<T> :
  BaseClass<T>
{...}
```

Наследование обобщённого типа с ограничениями

- Ограничение наследования

```
public class BaseClass<T>  where T : ISomeInterface
{...}
public class SubClass<T> : BaseClass<T>
    where T : ISomeInterface
{...}
```

- Ограничение конструктора

```
public class BaseClass<T>  where T : new()
{
   public T SomeMethod()
   {
      return new T();
   }
}
public class SubClass<T> : BaseClass<T>
    where T : new()
{...}
```

Полиморфические методы

```
public class BaseClass<T>
{
    public virtual T SomeMethod()
    {...}
}
public class SubClass : BaseClass<int>
{
    public override int SomeMethod()
    {...}
}
public class SubClass<T> : BaseClass<T>
{
    public override T SomeMethod()
    {...}
}
```

Интерфейсы, абстрактные классы, абстрактные методы

```
public interface ISomeInterface<T>
{
    T SomeMethod(T t);
}
public abstract class BaseClass<T>
{
    public abstract T SomeMethod(T t);
}

public class SubClass<T> : BaseClass<T>
{
    public override T SomeMethod(T t)
    {...}
}
```

Обобщённые методы

Часть 7

Методы с обобщёнными параметрами

- Внутри обобщённого типа

```
public class MyClass<T>
{
    public void MyMethod<X>(X x)
    {...}
}
```

- Внутри обычного типа

```
public class MyClass
{
    public void MyMethod<T>(T t)
    {...}
}
...
MyClass obj = new MyClass();
obj.MyMethod<int>(3);
// Пример выведения типа
obj.MyMethod(5);
```

Выведение типа не работает для возвращаемых параметров

```
public class MyClass
{
    public T MyMethod<T>()
    {...}
}
MyClass obj = new MyClass();
int number = obj.MyMethod();
```

Ограничения обобщённых параметров метода

```
public class MyClass
{
    public void SomeMethod<T>(T t)
        where T : IComparable<T>
    {...}
}
```

- Недопустима установка ограничения на наследование от класса, это можно делать только на уровне обобщённого типа данных

Обобщённые статические методы

```
public class MyClass<T>
{
    public static T SomeMethod(T t)
    {...}
}
int number =
  MyClass<int>.SomeMethod(3);
```

Ограничения на обобщённые параметры в статических методах

```
public class MyClass
{
    public static T SomeMethod<T>(T t)
        where T : IComparable<T>
    {...}
}
```

Обобщённые операторы

```
public class LinkedList<K, T>
{
    public static LinkedList<K, T> operator +(LinkedList<K, T> lhs,
        LinkedList<K, T> rhs)
    {
        return concatenate(lhs, rhs);
    }
    static LinkedList<K, T> concatenate(LinkedList<K, T> list1,
        LinkedList<K, T> list2)
    {
        LinkedList<K, T> newList = new LinkedList<K, T>();
        Node<K, T> current;
        current = list1.m_Head;
        while (current != null)
        {
            newList.AddHead(current.Key, current.Item);
            current = current.NextNode;
        }
        current = list2.m_Head;
        while (current != null)
        {
            newList.AddHead(current.Key, current.Item);
            current = current.NextNode;
        }
        return newList;
    }
    ...
}
```

Обобщённые делегаты

Часть 8

Делегат может указывать на метод с обобщённым параметром

```
public class MyClass<T>
{
    public delegate void
        GenericDelegate(T t);

    public void SomeMethod(T t)
    {...}
}
...
MyClass<int> obj = new MyClass<int>();
MyClass<int>.GenericDelegate del;

del = new MyClass<int>.GenericDelegate(
    obj.SomeMethod);
del(3);
```

Делегат с собственным обобщённым параметров

```
public class MyClass<T>
{
    public delegate void
        GenericDelegate<X>(
        T t, X x);
}
```

Обобщённое событие

```
public delegate void GenericEventHandler<S, A>(S sender, A args);

public class MyPublisher
{
    public event GenericEventHandler<MyPublisher,EventArgs> MyEvent;
    public void FireEvent()
    {
        MyEvent(this,EventArgs.Empty);
    }
}

public class MySubscriber<A>
{
    public void SomeMethod(MyPublisher sender,A args)
    {...}
}

MyPublisher publisher = new MyPublisher();
MySubscriber<EventArgs> subscriber = new MySubscriber<EventArgs>();
publisher.MyEvent += subscriber.SomeMethod;
```

Обобщённые коллекции и массивы

Часть 9

Добавленные статические элементы в тип System.Array

- IList<T> AsReadOnly<T>(T[] array);
- int BinarySearch<T>(T[] array, T value);
- int BinarySearch<T>(T[] array, T value,
  IComparer<T> comparer);
- U[] ConvertAll<T,U>(T[] array,
  Converter<T,U> converter);
- bool Exists<T>(T[] array,Predicate<T> match);
- T Find<T>(T[] array,Predicate<T> match);
- T[] FindAll<T>(T[] array, Predicate<T> match);
- int FindIndex<T>(T[] array, Predicate<T> match);
- void ForEach<T>(T[] array, Action<T> action);
- int IndexOf<T>(T[] array, T value);
- void Sort<K,V>(K[] keys, V[] items,
  IComparer<K> comparer);
- void Sort<T>(T[] array,Comparison<T> comparison);

Обобщённые коллекции

System.Collections.Generic	System.Collections
Comparer<T>	Comparer
Dictionary<K,T>	HashTable
LinkedList<T>	-
List<T>	ArrayList
Queue<T>	Queue
SortedDictionary<K,T>	SortedList
Stack<T>	Stack
ICollection<T>	ICollection
IComparable<T>	System.IComparable
IDictionary<K,T>	IDictionary
IEnumerable<T>	IEnumerable
IEnumerator<T>	IEnumerator
IList<T>	IList

Обобщения и рефлексия

Часть 10

Особенности рефлексии при работе с обобщениями

- Обобщённый тип может находиться в двух состояниях
  - Неограниченном (типы параметров не указаны)
  - Ограниченном(указан типы параметров)
- Получение информации о неограниченном типе данных

```
public class MyClass<T>
{}
Type unboundedType = typeof(MyClass<>);
Type unboundedList = typeof(LinkedList<,>);
```

- Получение информации об ограниченном типе данных

```
LinkedList<int,string> list = new
  LinkedList<int,string>();

Type type1 = typeof(LinkedList<int,string>);
Type type2 = list.GetType();
Debug.Assert(type1 == type2);
```

Элементы Type, предназначенные для обобщённых типов

- bool ContainsGenericParameters {get;}
- int GenericParameterPosition {get;}
- bool HasGenericArguments {get;}
- bool IsGenericParameter {get;}
- bool IsGenericTypeDefinition {get;}
- Type BindGenericParameters(Type[] typeArgs);
- Type[] GetGenericArguments();
- Type GetGenericTypeDefinition();

Обобщения и атрибуты

- Не допускается создавать обобщённые атрибуты

```
// Ошибка компиляции
public class SomeAttribute<T> :
  Attribute
{...}
```

- Для параметров обобщённых типов допускается использовать специальные атрибуты

```
[AttributeUsage(AttributeTargets.
GenericParameter)]
public class SomeAttribute : Attribute
{...}
```

Заключение

- Спасибо за внимание!