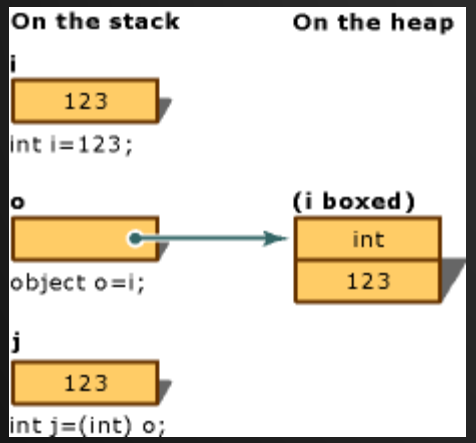
ПИШНИК: BECOME HUMAN

Глава 3: Упаковка и Распаковка Типов. Коллекции

Автор презентации – Сагалов Даниил БПИ-196

Упаковка. Распаковка

Вжух, упаковочка, вжух, распаковочка



Упаковка – процесс приведения значимого к Object или любому другому типу интерфейса, реализуемого данным значимым типом. В процессе упаковки тип значения инкапсулируется внутри экземпляра System.Object и сохраняется в куче общеязыковой средой выполнения (CLR).

Распаковка – процесс, обратный упаковке, извлекает значение из объекта.

Важно: упаковка происходит неявно, а распаковка требует обязательного явного приведения к исходному типу. Всегда нужно помнить, что упаковка и распаковка являются весьма затратными с точки зрения вычислений процессами (при упаковке нужно создать объект и разместить его в куче). При успешной распаковке будут выполнены 3 шага:

- Проверка инвариантности (совпадения) типов. Тип, к которому приводится упакованный объект должен строго совпадать с изначальным типом упакованного значения, иначе возникнет **InvalidCastException**.
- Проверка на null. При попытке распаковки null вы получите **NullReferenceException**.
- Копирование значения из экземпляра в указанную значимую переменную, если два прошлых шага были пройдены успешно.

Примеры Упаковки и Распаковки

При упаковке и распаковке всегда нужно помнить тип упакованного объекта, иначе вы получите **InvalidCastException**. Тем не менее, вы можете узнать тип упакованного объекта при помощи оператора **is** (ошибки возможны только в случае с nullable-типами). Оператор **as** для типов значений неприменим.

Примеры:

```
int int1 = 4;
object container1 = int1; // значение int1 упаковано как Int32
object container2 = (byte)int1; // значение int1 упаковано как byte
short short1 = (short)container1; // Исключение – short != int1
int int2 = 5 + (byte)container1; // Исключение – container1 содержит int, а не byte
int int3 = (int)container2; // Исключение – container2 содержит byte, а не int
int int4 = (int)container1 + 228; // ОК
byte byte1 = (byte)container2; // ОК
```

Задачи

Задание 1

Выберите верные утверждения: (*укажите все верные ответы*)**:**

1) Любой тип значения может быть упакован в Object;
2) Упаковка является явным приведением типов;
3) При распаковке вы можете привести объект только к типу, из которого он был изначально запакован;
4) Вы можете распаковать объект при помощи метода Unbox() класса Object;
5) Распаковка является неявным приведением типов;

Задание 2

Из указанных строк выберите те, написание которых НЕ приведёт к ошибке компиляции или исключению (*укажите все верные ответы*)**:**

```
using System;

public class Program {
    static void Main() {
        int myInt = 255;
        object obj1 = myInt;
        object obj2 = (byte)myInt;
        int resInt1 = 4 + (int)obj2;                    // 1
        int resInt2 = (byte)obj2 + (int)obj1;           // 2
        int resInt3 = obj1 as int;                      // 3
        long resLong = (int)obj1 + 33;                  // 4
        byte resByte = obj2 is byte b ? b : (byte)0;    // 5
    }
}
```

Ответы

Ответ 1: **13**

Ответ 2: **245**

Осторожно! Legacy-Код

В настоящее время коллекции из пространства имён **System.Collections** не рекомендуется использовать, в первую очередь они нужны для обратной совместимости.

Дело в том, что в первых версиях платформы .NET не существовало обобщённых типов, и все коллекции **System.Collections являются нетипизированными**. В дальнейшем данные коллекции были заменены на содержащиеся в пространствах имён **System.Collections.Generic** и **System.Collections.ObjectModel**, что избавило разработчиков от основных недостатков:

- При работе с необобщёнными коллекциями возникает необходимость постоянному приведению от данного типа к **Object** и обратно. Так как компилятор не всегда способен в таком случае осуществить проверку, вероятность поместить не тот тип не в ту коллекцию возрастает.
- При приведении типов к **Object** возникает вторая проблема – возникает постоянная необходимость упаковки и распаковки значимых типов, что негативно отражается на общей производительности программы.

Замена на Обобщённые Коллекции

В данной таблице вы найдёте обобщённые коллекции, эквивалентные представленным в **System.Collections**:

Коллекция	Замена	Коллекция	Замена
ArrayList	List<T>	DictionaryEntry	KeyValuePair<TKey, TValue>
CaseInsensitiveComparer	StringComparer.OrdinalIgnoreCase	Hashtable	Dictionary<TKey, TValue>
CaseInsensitiveHashCodeProvider	StringComparer.OrdinalIgnoreCase	Queue	Queue<T>
CollectionBase	Collection<T>	ReadOnlyCollectionBase	ReadOnlyCollection<T>
Comparer	Comparer<T>	SortedList	SortedList<TKey, TValue>
DictionaryBase	Dictionary<TKey,TValue> ИЛИ KeyedCollection<TKey, TItem>	Stack	Stack<T>

```
ArrayList exampleList = new ArrayList();
// ArrayList - массив object, хранит что угодно
exampleList.Add(new DateTime(1985, 10, 2));
exampleList.Add(exampleList[0].GetType());
// Добавляем дипазон значений в конец ArrayList
exampleList.AddRange(new object[] { 2038, "error", 3.14, 42});
for (int i = 0; i < exampleList.Count; i++)
    // Выводим в консоль все объекты типа Int32
    if (exampleList[i] is int integer)
        Console.WriteLine(integer);

// Меняем порядок элементов на противоположный
exampleList.Reverse();
foreach (var item in exampleList)
    Console.Write(item + " ");
// Выводим в консоль индекс первого вхождения объекта типа DateTime
Console.WriteLine("\n" + exampleList.IndexOf(typeof(DateTime)));
// Удаляем первый элемент ArrayList
exampleList.RemoveAt(0);
// Очистка всех элементов
exampleList.Clear();
```

ArrayList

ArrayList – динамический массив object[ ].

ArrayList( ) – создаёт пустой ArrayList с ёмкостью по умолчанию (равной 4)

ArrayList(ICollection col) – создаёт ArrayList, содержащий копии всех элементов указанной коллекции. Ёмкость равна ёмкости исходной коллекции.

ArrayList(Int32 count) – создаёт пустой ArrayList ёмкости count.

void Add(Object item) – добавляет элемент в конец данного ArrayList.

void AddRange(ICollection col) – добавляет элементы col в конец данного ArrayList.

void Clear( ) – удаляет все элементы данного ArrayList.

int Capacity – свойство, возвращает или задаёт вместимость данного ArrayList.

int Count – свойство, возвращает количество реально содержащихся в ArrayList элементов.

ArrayList

void TrimToSize( ) – задаёт значение Count свойству Capacity.

bool Contains(object item) – возвращает true, если item содержится в данном ArrayList, в противном случае false.

ArrayList GetRange(Int32 index, Int32 count) – возвращается ArrayList, содержащий count элементов исходного, начиная с index.

Array ToArray( ) – копирует элементы ArrayList в новый массив Object и возвращает его.

Array ToArray(Type type) – копирует элементы ArrayList в новый массив Object заданного типа и возвращает его.

void Insert(Int32 pos, Object val) – вставляет val на позицию pos.

void InsertRange(Int32 pos, ICollection col) – вставляет элементы коллекции col на позицию pos.

void Remove(Object item) – удаляет первое вхождение item в ArrayList.

void RemoveAt(Int32 index) – удаляет элемент с по индексу index.

void Reverse( ) – меняет порядок элементов на противоположный.

void Sort( ) – сортирует все элементы ArrayList.

void Sort(IComparer comparer) – сортирует ArrayList с помощью указанной функции сравнения.

```
Hashtable exampleHashtable = new Hashtable();
for (char i = 'a'; i <= 'z'; ++i)
    exampleHashtable.Add(i, 1);

foreach (DictionaryEntry entry in exampleHashtable)
    Console.WriteLine($"Key: {entry.Key}, value: {entry.Value}.");

long a;
try {
    a = (long)exampleHashtable['a'];
}
catch (InvalidCastException) {
    Console.WriteLine("Проблемы упаковки наглядно - нельзя распаковать" +
        " значение типа int в значение типа long.");
}
try {
    char[] alphabet = (char[])exampleHashtable.Keys;
}
catch (InvalidCastException){
    Console.WriteLine("Не получится привести KeyCollection к char[].");
}
// Не содержит ключа ы, вернёт false;
Console.WriteLine(exampleHashtable.ContainsKey('ы'));
```

Hashtable

Hashtable – коллекция пар «ключ-значение», упорядоченных по хэш-коду ключа.

int Count – свойство, возвращает количество пар «ключ-значение».

ICollection Keys – свойство, возвращает коллекцию ключей данного Hashtable.

ICollection Values – свойство, возвращает коллекцию значений данного Hashtable.

void Add(Object key, Object value) – добавляет элемент с указанным ключом и значением в данный HashTable. Если объект с указанным ключом уже добавлен, возникнет **ArgumentException**.

void Remove(Object item) – удаляет элемент с указанным ключом из данного Hashtable.

void Clear( ) – удаляет все элементы данного ArrayList.

bool ContainsKey(Object key) – возвращает true, если ключ содержится в данном Hashtable и false в противном случае. bool Contains(Object key) работает таким же образом.

bool ContainsValue(Object value) – вернёт true, если значение содержится в данном Hashtable и false в противном случае.

```
// Создадим очередь с начальной вместимостью
// 4 элемента и коэффициентом роста 3.0.
Queue exampleQueue = new Queue(4, 3.0f);
for (int i = 1; i < 5; i++)
    exampleQueue.Enqueue(i);

// Выведется одно и то же число 1, Peek()
// не удаляет элемент очереди.
Console.WriteLine(exampleQueue.Peek());
Console.WriteLine(exampleQueue.Dequeue());
```

Queue

Queue – коллекция объектов, работающая по принципу **first-in - first-out**. Для очереди вводится понятие коэффициент роста – число, на которое умножается ёмкость очереди при нехватке места. По умолчанию равен 2.0, при этом очередь всегда увеличивается как минимум на 4 элемента (например, если создать Queue с коэф. Роста 1.0, очередь всегда будет увеличиваться на 4 элемента).

Queue(int capacity, float growthMult) – создаёт очередь заданной ёмкости capacity с коэффициентом роста growthMult.

int Count – свойство, возвращает количество элементов очереди.

void Enqueue(Object item) – добавляет элемент в конец очереди.

Object Dequeue( ) – удаляет и возвращает элемент в начале очереди. Бросает **InvalidOperationException**, если очередь пуста.

Object Peek( ) – возвращает элемент в начале очереди, не удаляя его. Бросает **InvalidOperationException**, если очередь пуста.

void Clear( ) – удаляет все элементы данного объекта Queue.

```
Stack exampleStack = new Stack();
for (int i = 1; i < 5; i++)
    exampleStack.Push(i);

// Выведется одно и то же число 4, Peek()
// не удаляет элемент очереди.
// В отличие от очереди стек первым вернёт
// последний добавленный объект
Console.WriteLine(exampleStack.Peek());
Console.WriteLine(exampleStack.Pop());
```

Stack

Stack – коллекция объектов, упорядоченная по принципу **last-in - first-out**.

Stack(int capacity) – создаёт стек заданной ёмкости capacity или ёмкостью по умолчанию, если его значение больше capacity.

int Count – свойство, возвращает количество элементов очереди.

void Push(Object item) – вставляет объект на верх стека.

Object Pop( ) – удаляет и возвращает элемент в начале стека. Бросает **InvalidOperationException**, если стек пустой.

Object Peek( ) – возвращает элемент в начале стека, не удаляя его. Бросает **InvalidOperationException**, если стек пустой.

void Clear( ) – удаляет все элементы данного объекта Queue.

Задание 1

В результате выполнения фрагмента программы:

```
using System;
using System.Collections;

class Program {
    static ArrayList ar = new ArrayList(10);
    static void Main() {
        for (int i = 0; i < ar.Capacity - 5; i++) {
            ar.Add(i);
            ar.Add(i);
            ar.Reverse();
        }
        for (int i = 0; i < ar.Capacity; i++) {
            Console.Write(ar[i]);
        }
    }
}
```

на экран будет выведено:

Примечание:
*Если возникнет ошибка компиляции, введите: ****
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---
Если возникнет ошибка исполнения или исключение, введите: +++

Задание 2

В результате выполнения фрагмента программы:

```
using System;
using System.Collections;

class Program {
    static Stack s = new Stack();
    static void Main() {
        s.Push(1);
        for (int i = 0; i < s.Count + 5; i++) {
            s.Push(i);
            s.Push(i);
            Console.Write((int)s.Pop() + (int)s.Pop() +
(int)s.Peek());
        }
    }
}
```

на экран будет выведено:

Примечание:
*Если возникнет ошибка компиляции, введите: ****
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---
Если возникнет ошибка исполнения или исключение, введите: +++

Задачи

Задание 3

В результате выполнения фрагмента программы:

```
using System;
using System.Collections;

class Program {
    static Queue q = new Queue();
    static void Main() {
        q.Enqueue(1);
        for (int i = 0; i < q.Count + 5; i++) {
            q.Enqueue(i);
            q.Enqueue(i);
            Console.Write((int)q.Dequeue() + (int)q.Dequeue() +
(int)q.Peek());
        }
    }
}
```

на экран будет выведено:

Примечание:
*Если возникнет ошибка компиляции, введите: ****
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---
Если возникнет ошибка исполнения или исключение, введите: +++

Задание 4

Выберите верные утверждения о коллекциях: (*укажите все верные ответы*)**:**

1) Класс HashTable находится в пространстве имён System.Collections.Generic;
2) ArrayList хранит все объекты в виде ссылок типа Object;
3) Коллекция Queue работает по принципу «first-in – first-out»;
4) HashTable – неупорядоченная коллекция.
5) Вы можете получить последний в очереди элемент коллекции Queue, не удаляя его при помощи метода Peek();

Ответы

Ответ 1: **4422001133**

Ответ 2: **1357911**

Ответ 3: **12581114**

Ответ 4: **234**

Обобщённые Коллекции

На замену коллекциям System.Collections пришли обобщённые коллекции пространства имён **System.Collections.Generic**, которые избавили программистов от необходимости постоянной упаковки и распаковки. Данные в обобщённых коллекциях хранятся в том типе, который указывался при их создании.

List<T> – обобщённый аналог ArrayList, содержит дополнительный методы, принимающие делегаты, которые аналогичны статическим методам Array (Exists, FindAll, TrueForAll), а так же собственный метод **public int RemoveAll (Predicate<T> match)**, который удаляет все элементы, удовлетворяющие условию и возвращает их количество.

Dictionary<TKey, TValue> – аналог Hashtable, содержит дополнительные методы **bool TryAdd (TKey key, TValue value)** (если ключ уже добавлен, его значение не будет переопределено, не возникнет исключения, а метод вернёт false) и **bool TryGetValue (TKey key, out TValue value)** (задаёт значение по умолчанию типа TValue и возвращает false, если ключ не найден. Выбрасывает **ArgumentNullException**, если key имеет значение null).

Queue<T> и **Stack<T>** – аналоги необобщённых очереди и стеках.

LinkedList<T>

LinkedList<T> – двусвязный список, элементы которого являются объектами **LinkListNode<T>**, который не наследуется. Вы можете вставить один и тот же узел двусвязного списка несколько раз, не создавая дополнительных объектов, а так же использовать null в качестве значения для ссылочных типов. При попытке добавить LinkedListNode<T> одного LinkedList<T> в другой возникнет **InvalidOperationException**.

Каждый объект LinkListNode<T> содержит 4 свойства:

- **LinkedList<T> List** – ссылка на LinkedList, с которым связан данный объект.
- **LinkedListNode<T> Next** – ссылка на следующий узел.
- **LinkedListNode<T> Previous** – ссылка на предыдущий узел.
- **T Value** – значение узла.

LinkedList<T> содержит 3 свойства:

- **int Count** – количество узлов, которое реально хранится
- **LinkedListNode<T> First { get; }** – первый узел данного LinkedList<T>, равен null если список пуст.
- **LinkedListNode<T> Last { get; }** – последний узел данного LinkedList<T>, равен null если список пуст.

Методы LinkedList<T>

void AddAfter (LinkedListNode<T> node, LinkedListNode<T> newNode) – добавляет новый узел после newNode.

LinkedListNode<T> AddAfter (LinkedListNode<T> node, T value) – добавляет новый узел, содержащий указанное значение после узла node. При отсутствии узла node в обоих случаях возникнет **InvalidOperationException**.

* Для добавления перед узлом существует метод **AddBefore**.

void AddFirst (LinkedListNode<T> node) – добавляет node в качестве первого узла. Если node принадлежит другому LinkedList<T>, возникнет **InvalidOperationException**.

LinkedListNode<T> AddFirst (T value) – создаёт узел, содержащий value, добавляет его первым и возвращает его.

* Для добавления последнего элемента существует метод **AddLast**.

void Remove (LinkedListNode<T> node) – удаляет указанный узел. Выбрасывает **InvalidOperationException**, если узел не найден.

bool Remove (T value) – удаляет указанное значение, если это возможно и возвращает true, в противном случае вернёт false.

void RemoveFirst( ) / void RemoveLast( ) – удаляет первый/последний узел LinkedList<T>. Выбрасывает **InvalidOperationException**, если данный двусвязный список пуст.

HashSet<T>

HashSet<T> – неупорядоченная коллекция объектов, объекты которой не повторяются. HashSet предоставляет набор математических операций, таких как сложение и вычитание множеств. Свойство Count позволяет получить количество элементов HashSet.

Из наиболее интересных методов HashSet<T>:

void ExceptWith(System.Collections.Generic.IEnumerable<T> other) – вычитает все элементы реализующей интерфейс коллекции из данного HashSet.

void IntersectWith (IEnumerable<T> other) – пересекает заданное с реализующей интерфейс коллекцией, оставляя в полученной коллекции только их пересечение.

bool IsProperSubsetOf (IEnumerable<T> other) – возвращает true, если данный HashSet строго включается в реализующую интерфейс коллекцию и false в противном случае. **IsSubsetOf** – аналог, проверяющий нестрогое включение.

HashSet<T>

bool IsSupersetOf (IEnumerable<T> other) – возвращает true, если данный HashSet строго включает в себя реализующую интерфейс коллекцию и false в противном случае. IsSupersetOf – аналог, проверяющий нестрогое включение.

bool Overlaps (IEnumerable<T> other) – возвращает true, если в двух коллекциях содержится хотя бы два общих элемента.

int RemoveWhere (Predicate<T> match) – удаляет все элементы коллекции, удовлетворяющие предикату match и возвращает их количество.

void SymmetricExceptWith (IEnumerable<T> other) – изменяет коллекцию так, что в итоговой будут содержаться либо элементы только данного множества, либо только переданного, но не их пересечение.

void UnionWith (IEnumerable<T> other) – объединяет данный HashSet с указанной коллекцией.

Задачи

Задание 1

В результате выполнения фрагмента программы:

```
using System;

public class Program {
    static void Main() {
        Dictionary<char, int> alphabet = new Dictionary<char, int>();
        for (char i = 'a'; i < 'Z'; ++i)
            alphabet.Add(i, 1);

        Console.Write(alphabet.Count);
        for (char i = 'z'; i >= 'a' * 1.2; --i) {
            if (!alphabet.ContainsKey(i)) {
                alphabet.Add(i, 1);
                Console.Write(i);
            }
        }
    }
}
```

на экран будет выведено:

Задание 2

Выберите верные утверждения: (*укажите все верные ответы*)**:**

1) Класс LinkedListNode<T> может иметь наследников;
2) Класс SortedList<T> содержит метод Overlaps(IEnumerable<T> other) для проверки наличия общих элементов в двух коллекциях;
3) К элементам коллекции Stack<T> нельзя обращаться по индексу;
4) Пару «ключ-значение» коллекции Dictionary<TKey, TValue> можно получить в виде объекта специального типа;
5) Свойство Capacity коллекции LinkedList<T> возвращает количество реально хранящихся в коллекции элементов;

Задание 3

В результате выполнения фрагмента программы:

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

public class Program {
    static void Main() {
        List<int> numbers = new List<int>(new int[] { 3, 100, 4, 200, 5, 300 });
        int trigger = 0;

        for (int i = 0; i < numbers.Count; i++) {
            if (numbers[i] / 100 > 0) {
                ++trigger;
                numbers.AddRange(new int[] { 1, 0 });
                if (trigger == 3)
                    numbers.Add(100);
            }
        }
        List<int> res = numbers.FindAll(number => {
            if (number % 10 == 1)
                return true;
            return false;
            });
        Console.Write(res[0] + res.Count + "" + res[res.Count - 1]);
    }
}
```

на экран будет выведено:

Ответы

Ответ 1: **0zyxwvu**

Ответ 2: **34**

Ответ 3: **51**