

Коллекции и классы- прототипы



Обобщения

- Начиная с версии .NET 2.0, язык программирования C# был расширен поддержкой средства, которое называется **обобщением (*generic*)**.
- Термин *обобщение* означает параметризированный тип.

Пример

```
class MyObj<T>
{
    T obj;

    ссылка 1
    public MyObj(T obj)
    {
        this.obj = obj;
    }

    ссылок 0
    public void objectType()
    {
        Console.WriteLine("Тип объекта: " + typeof(T));
    }
}
```

```
MyObj<string> name = new MyObj<string>("Alex");
name.objectType();
```

```
Тип объекта: System.String
Для продолжения нажмите любую клавишу . . .
```

Свойства обобщений

- **Безопасность** Обобщения автоматически обеспечивают типовую безопасность всех операций.
- **Повторное использование двоичного кода**
- **"Разбухание" кода** Поскольку определение обобщенного класса включается в сборку, создание на его основе конкретных классов специфических типов не приводит к дублированию кода в IL.

Ограничение обобщений

- Указывая параметр типа, можно наложить определенное ограничение на этот параметр.

*class имя_класса<параметр_типа> where
параметр_типа : ограничения { // ...*

`class MyObj<T> where T : MyClass, IMyInterface,
new() { // ...`

Коллекции

- В C# **коллекция** представляет собой совокупность объектов.
- В среде *.NET Framework* имеется немало интерфейсов и классов, в которых определяются и реализуются различные типы коллекций.
- Основной для большинства коллекций является реализация интерфейсов `IEnumerable` и `IEnumerator`. Благодаря такой реализации мы можем перебирать объекты в цикле `foreach`.

Коллекции

- *Коллекции упрощают решение многих задач программирования благодаря тому, что предлагают готовые решения для создания целого ряда типичных, но порой трудоемких для разработки структур данных. Например, в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связанных списков, стеков, очередей и хеш-таблиц. Коллекции являются современным технологическим средством, заслуживающим пристального внимания всех, кто программирует на C#.*
- *Первоначально существовали только классы необобщенных коллекций. Но с внедрением обобщений в версии C# 2.0 среда .NET Framework была дополнена многими новыми обобщенными классами и интерфейсами. Благодаря введению обобщенных коллекций общее количество классов и интерфейсов удвоилось. Вместе с библиотекой распараллеливания задач (TPL) в версии 4.0 среды .NET Framework появился ряд новых классов коллекций, предназначенных для применения в тех случаях, когда доступ к коллекции осуществляется из нескольких потоков. Нетрудно догадаться, что прикладной интерфейс Collections API составляет значительную часть среды .NET Framework.*

Типы коллекций

- **Необобщенные коллекции** Реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также словари, в которых можно хранить пары "ключ-значение". Классы и интерфейсы необобщенных коллекций находятся в пространстве имен **System.Collections**.
- **Специальные коллекции** Оперируют данными конкретного типа или же делают это каким-то особым образом. Специальные коллекции объявляются в пространстве имен **System.Collections.Specialized**.
- **Поразрядная коллекция** В прикладном интерфейсе Collections API определена одна коллекция с поразрядной организацией — это **BitArray**. Коллекция типа **BitArray** поддерживает поразрядные операции объявляется в пространстве имен **System.Collections**.
- **Обобщенные коллекции** Обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связанные списки, стеки, очереди и словари. Обобщенные коллекции объявляются в пространстве имен **System.Collections.Generic**.
- **Параллельные коллекции** Поддерживают многопоточный доступ к коллекции. Это обобщенные коллекции, определенные в пространстве имен **System.Collections.Concurrent**.

Необобщенные коллекции

- **ICollection**: является основой для всех необобщенных коллекций, определяет основные методы и свойства для всех необобщенных коллекций (например, метод `CopyTo` и свойство `Count`). Данный интерфейс унаследован от интерфейса `IEnumerable`, благодаря чему базовый интерфейс также реализуется всеми классами необобщенных коллекций
- **IList**: позволяет получать элементы коллекции по порядку. Также определяет ряд методов для манипуляции элементами.
- **IComparer**: определяет метод `int Compare(object x, object y)` для сравнения двух объектов
- **IDictionary**: определяет поведение коллекции, при котором она должна хранить объекты в виде пар ключ-значение
- **IDictionaryEnumerator**: определяет методы и свойства для перечислителя словаря
- **IEqualityComparer**: определяет два метода `Equals` и `GetHashCode`, с помощью которых два объекта сравниваются на предмет равенства
- **IStructuralComparer**: определяет метод `Compare` для структурного сравнения двух объектов: при таком сравнении сравниваются не ссылки на объекты, а непосредственное содержимое объектов
- **IStructuralEquatable**: позволяет провести структурное равенство двух объектов. Как и в случае с интерфейсом `IStructuralComparer` сравнивается содержимое двух объектов

Классы необобщенных коллекций

- **ArrayList**: класс простой коллекции объектов. Реализует интерфейсы `IList`, `ICollection`, `IEnumerable`
- **BitArray**: класс коллекции, содержащей массив битовых значений. Реализует интерфейсы `ICollection`, `IEnumerable`
- **Hashtable**: класс коллекции, представляющей хэш-таблицу и хранящий набор пар "ключ-значение"
- **Queue**: класс очереди объектов, работающей по алгоритму FIFO ("первый вошел - первый вышел"). Реализует интерфейсы `ICollection`, `IEnumerable`
- **SortedList**: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы `ICollection`, `IDictionary`, `IEnumerable`
- **Stack**: класс стека

ArrayList

- Предназначен для хранения разнотипных объектов(int, bool, string, decimal)

```
ArrayList list = new ArrayList();  
    list.Add(2.3); // заносим в список  
объект типа double  
    list.Add(55); // заносим в список  
объект типа int  
    list.AddRange(new string[] { "Hello",  
"world" }); // заносим в список строковый  
массив  
  
    // перебор значений  
foreach (object o in list)  
{  
    Console.WriteLine(o);  
}  
  
// удаляем первый элемент  
list.RemoveAt(0);  
// переворачиваем список  
list.Reverse();  
// получение элемента по индексу  
Console.WriteLine(list[0]);  
// перебор значений  
for (int i = 0; i < list.Count; i++)  
{  
    Console.WriteLine(list[i]);  
}
```

Обобщенные коллекции

Классы обобщенных коллекций находятся в пространстве имен **System.Collections.Generic**.

Рассмотрим основные интерфейсы обобщенных коллекций:

- **IEnumerable<T>**: определяет метод `GetEnumerator`, с помощью которого можно получать элементы любой коллекции
- **IEnumerator<T>**: определяет методы, с помощью которых потом можно получить содержимое коллекции по очереди
- **ICollection<T>**: представляет ряд общих свойств и методов для всех необобщенных коллекций (например, методы `CopyTo`, `Add`, `Remove`, `Contains`, свойство `Count`)
- **IList<T>**: предоставляет функционал для создания последовательных списков
- **IComparer<T>**: определяет метод `Compare` для сравнения двух однотипных объектов
- **IDictionary<TKey, TValue>**: определяет поведение коллекции, при котором она должна хранить объекты в виде пар ключ-значение
- **IEqualityComparer<T>**: определяет методы, с помощью которых два однотипных объекта сравниваются на предмет равенства

Классы обобщенных коллекций

- Эти интерфейсы реализуются следующими классами коллекций в пространстве имен `System.Collections.Generic`:
- **List<T>**: класс, представляющий последовательный список. Реализует интерфейсы `ICollection<T>`, `ICollection<T>`, `IEnumerable<T>`
- **Dictionary<TKey, TValue>**: класс коллекции, хранящей наборы пар "ключ-значение". Реализует интерфейсы `ICollection<T>`, `IEnumerable<T>`, `IDictionary<TKey, TValue>`
- **LinkedList<T>**: класс двухсвязанного списка. Реализует интерфейсы `ICollection<T>` и `IEnumerable<T>`
- **Queue<T>**: класс очереди объектов, работающей по алгоритму FIFO ("первый вошел - первый вышел"). Реализует интерфейсы `ICollection`, `IEnumerable<T>`
- **SortedSet<T>**: класс отсортированной коллекции однотипных объектов. Реализует интерфейсы `ICollection<T>`, `ISet<T>`, `IEnumerable<T>`
- **SortedList<TKey, TValue>**: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы `ICollection<T>`, `IEnumerable<T>`, `IDictionary<TKey, TValue>`
- **SortedDictionary<TKey, TValue>**: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. В общем похож на класс `SortedList<TKey, TValue>`, основные отличия состоят лишь в использовании памяти и в скорости вставки и удаления
- **Stack<T>**: класс стека однотипных объектов. Реализует интерфейсы `ICollection<T>` и `IEnumerable<T>`

Список List<T>

Среди его методов можно выделить следующие:

- **void Add(T item):** добавление нового элемента в список
- **void AddRange(ICollection collection):** добавление с список коллекции или массива
- **int BinarySearch(T item):** бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.
- **int IndexOf(T item):** возвращает индекс первого вхождения элемента в списке
- **void Insert(int index, T item):** вставляет элемент item в списке на позицию index
- **bool Remove(T item):** удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true
- **void RemoveAt(int index):** удаление элемента по указанному индексу index
- **void Sort():** сортировка списка

ДВУХСВЯЗНЫЙ СПИСОК

LinkedList<T>

Класс LinkedList<T> представляет двухсвязный список, в котором каждый элемент хранит ссылку одновременно на следующий и на предыдущий элемент.

- Если в простом списке List<T> каждый элемент представляет объект типа T, то в LinkedList<T> каждый узел представляет объект класса LinkedListNode<T>. Этот класс имеет следующие свойства:
- **Value**: само значение узла, представленное типом T
- **Next**: ссылка на следующий элемент типа LinkedListNode<T> в списке. Если следующий элемент отсутствует, то имеет значение null
- **Previous**: ссылка на предыдущий элемент типа LinkedListNode<T> в списке. Если предыдущий элемент отсутствует, то имеет значение null

ДВУХСВЯЗНЫЙ СПИСОК

LinkedList<T>

Используя методы класса `LinkedList<T>`, можно обращаться к различным элементам, как в конце, так и в начале списка:

- **`AddAfter(LinkedListNode<T> node, T value)`**: вставляет в список новый узел со значением `value` после узла `node`.
- **`AddBefore(LinkedListNode<T> node, T value)`**: вставляет в список новый узел со значением `value` перед узлом `node`.
- **`AddFirst(T value)`**: вставляет новый узел со значением `value` в начало списка
- **`AddLast(T value)`**: вставляет новый узел со значением `value` в конец списка
- **`RemoveFirst()`**: удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным
- **`RemoveLast()`**: удаляет последний узел из списка

ДВУХСВЯЗНЫЙ СПИСОК

LinkedList<T>

```
LinkedList<int> numbers = new  
LinkedList<int>();
```

numbers.AddLast(1); // вставляем
узел со значением 1 на последнее
место

numbers.AddFirst(2); //
вставляем узел со значением 2 на
первое место

numbers.AddAfter(numbers.Last,
3); // вставляем после
последнего узла новый узел со
значением 3

```
LinkedList<Person> persons = new  
LinkedList<Person>();
```

// добавляем персона в
список и получим объект
LinkedListNode<Person>, в котором
хранится имя Tom

```
LinkedListNode<Person> tom =  
persons.AddLast(new Person() { Name  
= "Tom" });  
persons.AddLast(new Person() { Name  
= "John" });  
persons.AddFirst(new Person() {  
Name = "Bill" });
```

```
Console.WriteLine(tom.Previous.Valu  
e.Name); // получаем узел перед  
ТОМОМ и его значение
```

```
Console.WriteLine(tom.Next.Value.Na  
me); // получаем узел после тома и  
его значение
```

Очередь Queue<T>

- У класса Queue<T> можно отметить следующие методы:
- **Dequeue**: извлекает и возвращает первый элемент очереди
- **Enqueue**: добавляет элемент в конец очереди
- **Peek**: просто возвращает первый элемент из начала очереди без его удаления

Очередь Queue<T>

- `Queue<int> numbers = new Queue<int>();`
-
- `numbers.Enqueue(3); // очередь 3`
- `numbers.Enqueue(5); // очередь 3, 5`
- `numbers.Enqueue(8); // очередь 3, 5, 8`
-
- `// получаем первый элемент очереди`
- `int queueElement = numbers.Dequeue();`
`//теперь очередь 5, 8`
- `Console.WriteLine(queueElement);`

Коллекция Stack<T>

- В классе Stack можно выделить два основных метода, которые позволяют управлять элементами:
- **Push**: добавляет элемент в стек на первое место
- **Pop**: извлекает и возвращает первый элемент из стека
- **Peek**: просто возвращает первый элемент из стека без его удаления

Коллекция Stack<T>

```
Stack<int> numbers = new Stack<int>();
```

```
numbers.Push(3); // в стеке 3
```

```
numbers.Push(5); // в стеке 5, 3
```

```
numbers.Push(8); // в стеке 8, 5, 3
```

```
// так как вверху стека будет находиться  
число 8, то оно и извлекается
```

```
int stackElement = numbers.Pop(); // в стеке 5, 3
```

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

Коллекция Dictionary<T, V>

- Словарь хранит объекты, которые представляют пару ключ-значение. Каждый такой объект является объектом класса **KeyValuePair<TKey, TValue>**. Благодаря свойствам **Key** и **Value**, которые есть у данного класса, мы можем получить ключ и значение элемента в словаре.
- *Класс словарей также, как и другие коллекции, предоставляет методы **Add** и **Remove** для добавления и удаления элементов. Только в случае словарей в метод **Add** передаются два параметра: ключ и значение. А метод **Remove** удаляет не по индексу, а по ключу.*

Коллекция Dictionary<T, V>

```
Dictionary<int, string> countries = new Dictionary<int, string>(5);  
countries.Add(1, "Russia");  
countries.Add(3, "Great Britain");  
countries.Add(2, "USA");  
countries.Add(4, "France");  
countries.Add(5, "China");
```

```
foreach (KeyValuePair<int, string> keyValue in countries)  
{  
    Console.WriteLine(keyValue.Key + " - " + keyValue.Value);  
}
```

Коллекция Dictionary<T, V>

```
Dictionary<int, string> countries =  
    new Dictionary<int, string>(5);  
countries.Add(1, "Russia");  
countries.Add(3, "Great Britain");  
countries.Add(2, "USA");  
countries.Add(4, "France");  
countries.Add(5, "China");  
foreach (KeyValuePair<int, string>  
    keyValue in countries)  
{  
    Console.WriteLine(keyValue.Key  
        + " - " + keyValue.Value);  
}
```

```
// получение элемента по ключу  
string country = countries[4];  
// изменение объекта  
countries[4] = "Spain";  
// удаление по ключу  
countries.Remove(2);
```


Коллекция Dictionary<T, V>

- Dictionary<char, Person> people = new Dictionary<char, Person>();
- people.Add('b', new Person() { Name = "Bill" });
- people.Add('t', new Person() { Name = "Tom" });
- people.Add('j', new Person() { Name = "John" });
-
- foreach (KeyValuePair<char, Person> keyValue in people)
- {
- // keyValue.Value представляет класс Person
- Console.WriteLine(keyValue.Key + " - " + keyValue.Value.Name);
- }