Коллекции

Во многих приложениях требуется создавать группы связанных объектов и управлять этими группами. Существует два способа группировки объектов: создать **массив** объектов и создать **коллекцию**.

Массивы удобнее всего использовать для создания и работы с фиксированным числом строго типизированных объектов.

Коллекции предоставляют более гибкий способ работы с группами объектов. В отличие от массивов, группа объектов в классе может динамически возрастать и сокращаться в соответствии с потребностями приложения.

Некоторые коллекции допускают назначение ключа любому объекту, который добавляется в коллекцию, чтобы в дальнейшем можно было быстро извлечь связанный с ключом объект из коллекции.

Коллекция является классом, поэтому необходимо объявлять новую коллекцию перед добавлением в неё элементов.

Многие типовые коллекции предоставляются платформой .NET Framework. Каждый тип коллекции предназначен для определенной цели.

Коллекции основаны на интерфейсах ICollection, IList, IDictionary или на их универсальных аналогах.

Интерфейсы IList и IDictionary являются производными от интерфейса ICollection.

Поэтому все коллекции прямо или косвенно основаны на интерфейсе ICollection.

Типы коллекции — это распространенные виды коллекций данных, такие как хэш-таблицы, очереди, стеки, контейнеры, словари и списки.

Каждый тип коллекции предназначен для определенной цели.

Существуют следующие типовые группы классов коллекций:

- Классы System.Collections (устаревшая)
- Классы System.Collections.Generic
- Классы System.Collections.Concurrent

Классы System.Collections

Классы в пространстве имен System.Collections не хранят элементы в виде конкретно типизированных объектов, а хранят их как объекты типа Object.

В следующей таблице перечислены некоторые из часто используемых классов в пространстве имен System.Collections:

Класс	Описание
ArrayList	Представляет массив объектов, размер которого динамически увеличивается по мере необходимости.
Hashtable	Предоставляет коллекцию пар ключ/значение, которые упорядочены по хэш-коду ключа.
Queue	Предоставляет коллекцию объектов, которая обслуживается по принципу "первым пришел — первым вышел" (FIFO).
Stack	Представляет коллекцию объектов, которая обслуживается по принципу "последним пришел — первым вышел"

Лучше не использовать типы в пространстве имен System.Collections, если явным образом не нужна платформа .NET Framework версии 1.1.

System.Collections - это устаревшие группы классов.

Везде, где это возможно, следует использовать универсальные коллекции пространства имен System.Collections.Generic или пространства имен System.Collections.Concurrent вместо устаревших типов пространства имен System.Collections, так как они обладают большей типобезопасностью и содержат некоторые улучшения.

Классы System.Collections.Generic

Если коллекция содержит элементы только одного типа данных, можно использовать один из классов в пространстве имен System.Collections.Generic.

Универсальная (Generic) коллекция обеспечивает безопасность типов, так что другие типы данных не могут быть в нее добавлены.

При извлечении элемента из универсальной коллекции нет необходимости определять или преобразовывать его тип данных.

В следующей таблице перечислены некоторые из часто используемых классов пространства имен System.Collections.Generic:

Класс	Описание
Dictionary<TKey, TValue>	Предоставляет коллекцию пар ключ/значение, которые упорядочены по ключу.
List<T>	Представляет список объектов, доступных по индексу. Предоставляет методы для поиска по списку, выполнения сортировки и изменения списка.
Queue<T>	Предоставляет коллекцию объектов, которая обслуживается по принципу "первым пришел — первым вышел" (FIFO).
SortedList<TKey, TValue>	Представляет коллекцию пар ключ/значение, упорядоченных по ключу на основе реализации IComparer<T>.
Stack<T>	Представляет коллекцию объектов, которая обслуживается по принципу "последним пришел — первым вышел" (LIFO).

В следующих примерах используется универсальный класс List<T>.

Класс List<T> является универсальным эквивалентом класса ArrayList.

Он реализует универсальный интерфейс IList<T> с помощью массива, размер которого динамически увеличивается по мере необходимости.

Доступ к элементам этой коллекции осуществляется с помощью целочисленного индекса. Индексы в этой коллекции начинаются с нуля.

Для очень больших объектов List<T>, можно увеличить максимальную емкость до 2 миллиарда элементов в системе путем установки обновления 64 (**sp2**) **enabled** атрибут **gcAllowVeryLargeObjects** элемент конфигурации **true** в среде выполнения.

Делая выбор между классами List<T> и ArrayList, предлагающими сходные функциональные возможности, следует помнить, что класс List<T> в большинстве случаев обрабатывается быстрее и является типобезопасным.

Если в качестве типа T класса List<T> используется ссылочный тип, оба класса действуют идентичным образом.

Однако если в качестве типа T используется тип значений, необходимо принять во внимание особенности, связанные с реализацией и упаковкой.

Если в качестве типа T используется тип значений, компилятор генерирует реализацию класса List<T> специально для этого типа значений.

Это означает, что элемент списка List<T> не требует упаковки перед его использованием, и после создания примерно 500 элементов списка объем памяти, сэкономленной за счет отказа от их упаковки, превысит объем памяти, используемой для генерации реализации класса.

Необходимо убедиться, что тип значений, используемый в качестве типа **T**, реализует универсальный интерфейс **IEquatable<T>**. В противном случае такие методы, как **Contains**, должны будут вызывать метод **Object.Equals(Object)**, который осуществляет упаковку соответствующего элемента списка. Если тип значений реализует интерфейс **IComparable** и исходный код принадлежит вам, следует также реализовать универсальный интерфейс **IComparable<T>**, чтобы избежать упаковки элементов списка методами **BinarySearch** и **Sort**. Если исходный код принадлежит не вам, передайте объект **IComparer<T>** в методы **BinarySearch** и **Sort**.

Массив является одним из многочисленных вариантов хранения набора данных, используемых C#.

Вариант выбора зависит от нескольких факторов, например от планируемого способа управления или доступа к элементам. Например, *список* работает, как правило, быстрее массива при добавлении элемента в начало или в середину коллекции.

В следующем примере показано использование класса List<T>. Обратите внимание, что в отличие от класса Array, элементы могут вставляться в середину списка.

В следующих примерах показаны списки, который содержат только текстовые элементы.

```
public class TestCollections
 {
   public static void TestList()
    {
       System.Collections.Generic.List<string> sandwich =
                          new System.Collections.Generic.List<string>();

       sandwich.Add("bacon");
       sandwich.Add("tomato");
       sandwich.Insert(1, "lettuce");

       foreach (string ingredient in sandwich)
        {
           System.Console.WriteLine(ingredient);
        }
    }
 }
```

```
var salmons = new List<string>(); // Create a list of strings.

salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.

foreach (var salmon in salmons)
 {
   Console.Write(salmon + " ");
 }

// Output: chinook coho pink sockeye
```

Если содержимое коллекции известно заранее, можно использовать инициализатор коллекции для инициализации коллекции. Следующий пример аналогичен предыдущему за исключением того, что инициализатор коллекции используется для добавления элементов в коллекцию.

```
// Create a list of strings by using a collection initializer.

var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}

// Output:
chinook coho pink sockeye
```

Для выполнения итерации по коллекции можно использовать оператор for вместо оператора foreach.

Это выполняется путем доступа к элементам коллекции по позиции индекса.

Индекс элементов начинается с 0 и заканчивается на числе, равном количеству элементов за вычетом 1

```
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

 for (var index = 0; index < salmons.Count; index++)
 {
   Console.Write(salmons[index] + " ");
 }

// Output: chinook coho pink sockeye
```

Следующий пример удаляет элемент из коллекции путем указания объекта для удаления.

```
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

 salmons.Remove("coho");       // Remove an element from the list

 foreach (var salmon in salmons)
 {
   Console.Write(salmon + " ");
 }

// Output: chinook pink sockeye
```

В следующем примере для удаления элементов из универсального списка используется метод RemoveAt. Используется оператор for в порядке убывания, потому что в результате работы метода RemoveAt элементы, следующие за удаленным элементом, должны иметь меньшее значение индекса.

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

for (var index = numbers.Count - 1; index >= 0; index--)
{
   if (numbers[index] % 2 == 1)
    {
      // Remove the element by specifying the zero-based index in the list.
      numbers.RemoveAt(index);
    }
}

// A lambda expression is placed in the ForEach method  of the List(T) object.
numbers.ForEach( number => Console.Write(number + " "));

// Output: 0 2 4 6 8
```

Для типа элементов в List<T> можно также определить собственный класс. В следующем примере используется класс Galaxy, который используется объектом List<T>.

```
private void IterateThroughList()
{
 var theGalaxies = new List<Galaxy>
 { new Galaxy() { Name="Tadpole", MegaLightYears=400},
   new Galaxy() { Name="Pinwheel", MegaLightYears=25},
   new Galaxy() { Name="Milky Way", MegaLightYears=0},
   new Galaxy() { Name="Andromeda", MegaLightYears=3}
 };

 foreach (Galaxy theGalaxy in theGalaxies)
    { Console.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears); }
}
// Output:
// Tadpole 400
// Pinwheel 25
// Milky Way 0
// Andromeda 3

  public class Galaxy
   {
      public string Name { get; set; }
      public int MegaLightYears { get; set; }
    }
```

Универсальная коллекция Dictionary<TKey, TValue> позволяет получить доступ к элементам коллекции с помощью ключа каждого элемента. Каждый элемент, добавляемый словарь, состоит из **значения** и связанного с ним **ключа**.

Извлечение значения по его ключу происходит быстро, поскольку класс Dictionary реализован как хэш-таблица.

В следующем примере создается коллекция Dictionary и последовательно перебирается словарь с помощью оператора foreach.

```
public class Element
{
  public string Symbol { get; set; }
  public string Name { get; set; }
  public int AtomicNumber { get; set; }
}
```

```
private Dictionary<string, Element> BuildDictionary()
{
   var elements = new Dictionary<string, Element>();
   AddToDictionary(elements, "K", "Potassium", 19);
   AddToDictionary(elements, "Ca", "Calcium", 20);
   AddToDictionary(elements, "Sc", "Scandium", 21);
   AddToDictionary(elements, "Ti", "Titanium", 22);
   return elements;
}

private void AddToDictionary(Dictionary<string, Element> elements,
                        string symbol, string name, int atomicNumber)
{
  Element theElement = new Element();

  theElement.Symbol = symbol;
  theElement.Name = name;
  theElement.AtomicNumber = atomicNumber;
  elements.Add(key: theElement.Symbol, value: theElement);
}
```

```
private void IterateThruDictionary()
 {
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
      Element theElement = kvp.Value;
      Console.WriteLine("key: " + kvp.Key);
      Console.WriteLine("values: " + theElement.Symbol + " " +
          theElement.Name + " " + theElement.AtomicNumber);
    }
}
```

Можно заменить методы BuildDictionary и AddToDictionary следующим методом, используя инициализатор коллекции.

```
private Dictionary<string, Element> BuildDictionary()
{
  var elements = new Dictionary<string, Element>();
  AddToDictionary(elements, "K", "Potassium", 19);
  AddToDictionary(elements, "Ca", "Calcium", 20);
  AddToDictionary(elements, "Sc", "Scandium", 21);
  AddToDictionary(elements, "Ti", "Titanium", 22);
  return elements;
 }
```

```
private void AddToDictionary(Dictionary<string, Element> elements,
                string symbol, string name, int atomicNumber)
{
  Element theElement = new Element();

  theElement.Symbol = symbol;
  theElement.Name = name;
  theElement.AtomicNumber = atomicNumber;
  elements.Add(key: theElement.Symbol, value: theElement);
}
```

```
private Dictionary<string, Element> BuildDictionary2()
  {
   return new Dictionary<string, Element>
   {
    {"K", new Element()  { Symbol="K", Name="Potassium", AtomicNumber=19}},
    {"Ca", new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
    {"Sc", new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
    {"Ti", new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
   };
}
```

Следующем пример - быстрый поиск элемента по ключу. Доступ к элементу в коллекции elements осуществляется с помощью кода elements[symbol].

```
private void FindInDictionary(string symbol)
{
 Dictionary<string, Element> elements = BuildDictionary();

   if (elements.ContainsKey(symbol) == false)
    { Console.WriteLine(symbol + " not found"); }
   else
    {
      Element theElement = elements[symbol];
      Console.WriteLine("found: " + theElement.Name);
    }
}
```

В следующем примере вместо этого используется метод TryGetValue для быстрого поиска элемента по ключу.

```
private void FindInDictionary2(string symbol)
{
   Dictionary<string, Element> elements = BuildDictionary();

   Element theElement = null;

   if (elements.TryGetValue(symbol, out theElement) == false)
         Console.WriteLine(symbol + " not found");
   else
         Console.WriteLine("found: " + theElement.Name);
}
```

Классы System.Collections.Concurrent

Классы пространства имен System.Collections.Concurrent следует использовать вместо соответствующих типов пространств имен System.Collections.Generic и System.Collections, если несколько потоков параллельно обращаются к такой коллекции.

Определение пользовательской коллекции

Можно определить коллекцию путем реализации интерфейса IEnumerable<T> или IEnumerable.

Хотя можно определить пользовательскую коллекцию, обычно лучше использовать коллекции, входящие в платформу .NET Framework, которые выше.

В следующем примере определяется пользовательская коллекция с именем AllColors. Этот класс реализует интерфейс IEnumerable, который требует, чтобы метод GetEnumerator был реализован.

Метод GetEnumerator возвращает экземпляр класса ColorEnumerator. ColorEnumerator реализует интерфейс IEnumerator, который требует, чтобы были реализованы свойство Current, метод MoveNext и метод Reset.

```
private void ListColors()
{
  var colors = new AllColors();

  foreach (Color theColor in colors)
  {
     Console.Write(theColor.Name + " ");
  }
  Console.WriteLine();
}


// Output: red blue green

// Collection class.

public class AllColors : System.Collections.IEnumerable
{
 Color[] _colors =
  {
     new Color() { Name = "red" },
     new Color() { Name = "blue" },
     new Color() { Name = "green" }
  };
```

```
public System.Collections.IEnumerator GetEnumerator()
{
 return new ColorEnumerator(_colors);
 // Вместо создания пользовательских счетчиков(enumerator)
 // Вы можете использовать GetEnumerator массива
 // return _colors.GetEnumerator();
}
```

```
// Пользовательский enumerator.
private class ColorEnumerator : System.Collections.Ienumerator
 {
   private Color[] _colors;
   private int _position = -1;
   public ColorEnumerator(Color[] colors) { _colors = colors; }

   object System.Collections.IEnumerator.Current
    {
      get { return _colors[_position]; }
     }

    bool System.Collections.IEnumerator.MoveNext()
    {
      _position++;
      return (_position < _colors.Length);
    }

    void System.Collections.IEnumerator.Reset()
     {
      _position = -1;
     }
 }
}

// Element class.
 public class Color { public string Name { get; set; } }
```

Итераторы

Итератор используется для выполнения пользовательских итераций по коллекции. Итератор можно использовать для прохода по коллекции, такой как список и массив.

Итератор может быть методом или методом доступа get. Итератор использует оператор yield return для возврата каждого элемента коллекции по одному за раз.

Итератор вызывается с помощью оператора foreach. Каждая итерация цикла foreach вызывает итератор.

При достижении оператора yield return в итераторе возвращается выражение, и текущее расположение в коде **запоминается**.

При следующем вызове итератора выполнение **возобновляется с этого места**.

В следующем примере используется метод-итератор.

Метод-итератор содержит оператор yield return, который находится внутри цикла for .

В методе ListEvenNumbers каждая итерация тела оператора foreach создает вызов метода-итератора, который переходит к следующему оператору yield return.

```
private void ListEvenNumbers()
{
   foreach (int number in EvenSequence(5, 18))
    {
       Console.Write(number.ToString() + " ");
    }
   Console.WriteLine();
}
// Output: 6 8 10 12 14 16 18

 private static IEnumerable<int> EvenSequence( int firstNumber, int lastNumber)
 {

   for (var number = firstNumber; number <= lastNumber; number++)
    {
       if (number % 2 == 0) { yield return number; }
    }
}
```

Еще один пример:

```
static void Main()
{
  foreach (int number in SomeNumbers())
   {
     Console.Write(number.ToString() + " ");
   }
  Console.ReadKey();
}

// Output: 3 5 8

public static System.Collections.IEnumerable SomeNumbers()
{
  yield return 3;
  yield return 5;
  yield return 8;
}
```