Массивы и коллекции

Arrays and collections

Массивы и коллекции

Коллекция – это объект содержащий в себе набор значений одного или нескольких типов. Основная задача коллекции хранить значения объектов и обеспечивать доступ к ним. Коллекции позволяют экономить при разработке программ так как предоставляют готовые решения для построения структур данных. Пространство имён **`System.Collections`** содержит классы и интерфейсы для коллекций.

Массив - это коллекция переменных одного типа обращение к которым происходит с помощью одного имени и разных индексов. Хранение данных в массиве организуется таким образом чтобы к ним было легко обращаться. Массивы являются ссылочными типами и наследуют общий класс **`System.Array`**. Выход за границы массива не разрешается в случае такой попытки генерируется **`IndexOutOfRangeException`**.

Массивы

Класс Array

```
public abstract class Array : ICloneable, IList, ICollection, IEnumerable
{
   public static void Clear(Array array, int index, int length );
   public static void Copy(Array sourceArray, int sourceIndex,
      Array destinationArray, int destinationIndex, int length );
   public static int BinarySearch(Array array, Object value );
   public static void Reverse(Array array, int index, int length );
   public static void Sort(Array array, int index, int length );

   public int Length { get; }
   public int Rank { get; }
};
```

Методы и свойства класса Array

- Статический метод **`Clear`** задаёт диапазон элементов массива пустыми значениями.
- Статический метод **`Copy`** копирует диапазон элементов из одного массива в другой массив.
- Статический метод **`BinarySearch`** выполняет поиск значения в отсортированном одномерном массиве используя алгоритм двоичного поиска.
- Статический метод **`Reverse`** изменяет порядок элементов на обратный.
- Статический метод **`Sort`** сортирует элементы в одномерном массиве.

- Свойство **`Length`** возвращает число элементов в массиве.
- Свойство **`Rank`** возвращает размерность массива.

Одномерные массивы

Создание одномерного массива представляет из себя двухступенчатый процесс. Сначала объявляется ссылка на массив, после чего под массив выделяется область памяти и переменной присваивается ссылка на эту область памяти. При объявлении ссылки на массив в отличии от **C++** квадратные скобки находтся за именем типа а не за именем переменной. Обращение к элементам одномерного массива производится с использованием одного целого индекса. Все данные в одномерном массиве должны быть одного типа. Тип данных в одномерном массиве определяются при объявлении массива.

Использование одномерного массива

```
class Program
{
   static void Main()
   {
      int[] numbers = new int[10];
      int average = 0;

      numbers[0]=71; numbers[1]=99;
      numbers[2]=42; numbers[3]=3;
      numbers[4]=18; numbers[5]=45;
      numbers[6]=33; numbers[7]=6;
      numbers[8]=15; numbers[9]=64;

      for(int i=0; i<10;i++)
      {
         average += numbers[i];
      }
      average /= 10;
      Console.WriteLine("Среднее значение в массиве: {0}", average);
   }
}
```

Многомерные массивы

Многомерные массивы отличаются от одномерных тем что они характеризуются двумя или более измерениями. Поэтому обращение к элементам многомерного массива производится с использованием двух и более индексов. Все данные в многомерном массиве должны быть одного типа. Тип данных и количество измерений в многомерном массиве определяются при объявлении массива.

Использование многомерного массива

```
class Program
{
   static void Main()
   {
      int[,] numbers = new int[3,3];
      int average = 0;

      numbers[0,0]=27; numbers[0,1]=88; numbers[0,2]=59;
      numbers[1,0]=64; numbers[1,1]=17; numbers[1,2]=11;
      numbers[2,0]=12; numbers[2,1]=93; numbers[2,2]=8;

      for(int i=0; i<3;i++)
      {
         for(int j=0; j<3; j++)
         {
            average += numbers[i];
         }
      }
      average /= 9;
      Console.WriteLine("Среднее значение в массиве: {0}", average);
   }
}
```

Рваные массивы

Рваные массивы похожи на многомерные массивы в том что они тоже многомерные. Основное отличие заключается в том что рваные массивы это фактически массивы массивов. Отсюда следуют два важных факта. Диапазон допустимых индексов будет зависеть от значений предыдущих индексов. Данные в рваном массиве не обязаны быть одного типа они могут быть разными в зависимости от значений последовательности индексов за исключением последнего.

Как и для одномерных или многомерных массивах выход за границу допустимых значений для рваного массива не допускается разрешается в случае такой попытки генерируется **`IndexOutOfRangeException.`**

Использование рваного массива

```
class Program
{
   static void Main()
   {
      int[][] numbers = new int[3][];
      int average = 0;
      numbers[0] = new int[3];
      numbers[1] = new int[1];
      numbers[2] = new int[2];
      numbers[0][0]=91; numbers[0][1]=25; numbers[0][2]=15;
      numbers[1][0]=36;
      numbers[2][0]=17; numbers[2][1]=11;
           for(int i=0; i<3;i++)
      {
         for(int j=0; j<numbers[i].Length; j++)
         {
            average += numbers[i];
         }
      }
      average /= 6;
      Console.WriteLine("Среднее значение в массиве: {0}", average);
   }
}
```

Коллекции

Интерфейсы коллекций

Коллекции в FCL основанны на интерфейсах. Пространство имён **System.Collections** содержит несколько таких интерфейсов включая **IEnumerable, IEnumerator, ICollection, IList, IDictionary**.

Интерфейс **ICollection** является самым примитивным интерфейсом. Он определяет поведение которое должны поддерживать все коллекции. Например все коллекции должны поддерживать свойство возвращающее количество элементов в коллекции.

Интерфейс **IList** определяет поведение которое должны поддерживать списки то есть коллекции доступ к элементам которой возможен с помощью индекса.

Интерфейс **IDictionary** определяет поведение коллекций хранящих пары ключ-значение.

Интерфейсы **IEnumerable** и **IEnumerator** определяет поведение необходимое для поэлементного доступа к содержимому коллекции с помощью цикла **foreach**.

Интерфейсы IEnumerable и IEnumerator

```
public interface IEnumerable
{
   IEnumerator GetEnumerator();
}
```

```
public interface IEnumerator
{
   bool MoveNext(); // сдвинуть курсор
   object Current {get;} // достать текущий элемент
   void Reset(); // установить курсор перед первым элементом
}
```

Интерфейс ICollection

```
public interface ICollection : IEnumerable
{
   int Count {get;}
   bool IsSynchronized();
   object SybcRoot {get;}
   void CopyTo(Array array, int index);
}
```

Интерфейс IList

```
public interface IList : ICollection, IEnumerable
{
   Object this [int index] {get; set;}
   int Add(Object value);
   void Clear();
   bool Contains(Object value);
   int IndexOf(Object value);
   void Insert(int index, Object value);
   void Remove(Object value);
   void RemoveAt(int index);

   bool IsFixedSize {get;}
   bool IsReadOnly {get;}
}
```

Интерфейс IDictionary

```
public interface IDictionary : ICollection, IEnumerable
{
   object this [Object key] {get; set;}
   ICollection Keys {get;}
   ICollection Values {get;}
   void Add(Object key, Object value);
   void Clear();
   bool Contains(Object key);
   IDictionaryEnumerator GetEnumerator();
   void  Remove(Object key);

   bool IsFixedSize {get;}
   bool IsReadOnly {get;}
}
```

```
public interface IDictionaryEnumerator : IEnumerator
{
   DictionaryEntry Entry {get;}
   object Key {get;}
   object Value {get;}
}
```

Коллекции

Пространство имён **System.Collections** содержит несколько конкретных классов коллекций таких как **Queue**, **Stack**, **ArrayList**, **SortedList** и **HashTable** реализующих несколько из интерфейсов **IEnumerable**, **ICollection**, **IList**, **IDictionary**.

Реализация интерфейсов в конкретных классах очень сильно отличается. Несмотря на это наборы действий которые можно выполнять над этими коллекциями во многом совпадают. Все они поддерживают набор элементов которые можно перечислять и все они поддерживают добавление и удаление элементов.

Класс Queue

```
public class Queue : ICollection, Ienumerable, ICloneable
{
   Object Dequeue();
   void Enqueue(Object);
   Object Peek();
}
```

Методы класса Queue

- Метод **`Deque`** возвращает и удаляет элемент из начала очереди.
- Метод **`Enqueue`** добавляет элемент в конец очереди.
- Метод **`Peek`** возвращает элемент из начала очереди не удаляя его.

Использование очереди

```
static void Main()
{
   Queue someQ = new Queue();

   someQ.Enqueue("Первый добавленный");
   someQ.Enqueue("Второй добавленный ");
   someQ.Enqueue("Третий добавленный ");

   Console.WriteLine("Первый элемент: {0}", someQ.Peek());

   Console.WriteLine("Первый элемент: {0}", someQ.Dequeue());
   Console.WriteLine("Второй элемент: {0}", someQ.Dequeue());
   Console.WriteLine("Третий элемент: {0}", someQ.Dequeue());

   try
   {
      Console.WriteLine("Четвёртый элемент: {0}", someQ.Dequeue());
   }
   catch(InvalidOperationException)
   {
      Console.WriteLine("А четвёртого элемента в очереди нет!");
   }

}
```

Класс Stack

```
public class Stack : ICollection, IEnumerable, ICloneable
{
   Object Peek();
   Object Pop();
   void Push(Object);
}
```

Методы класса Stack

- Метод **`Peek`** возвращает верхний элемент из стека не удаляя его.
- Метод **`Pop`** возвращает и удаляет верхний элемент стека.
- Метод **`Push`** добавляет верхний элемент в стек.

Использование стека

```
static void Main()
{
   Stack someS = new Stack();

   someS.Push("Первый добавленный");
   someS.Push("Второй добавленный");
   someS.Push("Третий добавленный");

   Console.WriteLine("Первый элемент: {0}", someS.Peek());

   Console.WriteLine("Первый элемент: {0}", someS.Pop());
   Console.WriteLine("Второй элемент: {0}", someS.Pop());
   Console.WriteLine("Третий элемент: {0}", someS.Pop());

   try
   {
      Console.WriteLine("Четвёртый элемент: {0}", someS.Pop());
   }
   catch(InvalidOperationException)
   {
      Console.WriteLine("А четвёртого элемента в стеке нет!");
   }

}
```

Класс ArrayList

```
public class ArrayList : IList, ICollection, IEnumerable, ICloneable
{
   public virtual int Add(Object value);
   public virtual bool Contains(Object item);
   public virtual int IndexOf(Object value);
   public virtual void Insert(int index, Object value);
   public virtual void Remove(Object obj);
   public virtual void RemoveAt(int index);

   public virtual int Capacity { get; set; }
   public virtual int Count { get; }
   public virtual Object Item[ int index ] { get; set; }
}
```

Методы и свойства класса ArrayList

- Метод **`Add`** добавляет элемент в конец динамического массива.
- Метод **`Contains`** определяет содержит ли динамический массив указанное значение.
- Метод **`IndexOf`** возвращает индекс первого вхождения значения.
- Метод **`Insert`** вставляет элемент в динамический массив по указанному индексу.
- Метод **`Remove`** удаляет первое вхождение указанного значения из динамического массива.
- Метод **`RemoveAt`** удаляет элемент с указанным индексом из динамического массива.

- Свойство **`Capacity`** позволяет получать и задавать ёмкость динамического массива.
- Свойство **`Count`** возвращает количество элементов в динамическом массиве.

- Индексатор позволяет по индексу получать и изменять значение.

Использование динамического массива

```
static void Main()
{
   ArrayList someAL = new ArrayList();

   Console.WriteLine("Количество элементов после создания: {0}", someAL.Count);
   Console.WriteLine("Вместимость после создания: {0}", someAL.Capacity);

   someAL.Add("Первый добавленный");
   someAL.Add("Второй добавленный ");
   someAL.Add("Третий добавленный ");

   Console.WriteLine("Количество элементов после добавления: {0}", someAL.Count);
   Console.WriteLine("Вместимость после добавления: {0}", someAL.Capacity);

   for(int i=0; i<someAL.Count; i++)
   {
      Console.WriteLine("{0}ый элемент: {1}",i ,someAL[i]);
   }

}
```

Класс Hashtable

```
public class Hashtable : IDictionary, ICollection, IEnumerable, ICloneable
{
   public virtual void Add(Object key, Object value);
   public virtual void Clear();
   public virtual bool ContainsKey(Object key);
   public virtual bool ContainsValue(Object value);
   public virtual void Remove(Object key);

   public virtual Object Item[Object key] { get; set; }
   public virtual ICollection Keys { get; }
   public virtual ICollection Values { get; }
}
```

Методы и свойства класса HashTable

- Метод **`Add`** добавляет пару ключ-значение в хэш таблицу.
- Метод **`Clear`** удаляет все элементы из хэш таблицы.
- Метод **`ContainsKey`** определяет содержит ли хэш таблица указанный ключ.
- Метод **`ContainsValue`** определяет содержит ли хэш таблица указанное значение.
- Метод **`Remove`** удаляет пару ключ-значение с указанным ключом из хэш таблицы.

- Свойства **`Keys`** и **`Values`** возвращают коллекции ключей и значений для всех пар содержащихся в хэш таблице.

- Индексатор позволяет по ключу получать и изменять значение.

Использование хэш таблицы

```
static void Main()
{
   Hashtable someHT = new Hashtable();

   someHT.Add("Имя", "Иван");
   someHT.Add("Фамилия", "Петров");

   someHT["Должность"]="Разработчик .NET";

   ICollection keysC = someHT.Keys;

   foreach( string str in keysC)
   {
      Console.WriteLine(str + " : " + someHT[str]);
   }

   Console.WriteLine("Отчество: {0}", someHT.["Отчество"]);
}
```