5

Массивы

Если вам нужно работать с множественными объектами одного и того же типа, вы можете использовать коллекции и массивы. В C# предусмотрена специальная нотация для объявления и использования массивов.

И здесь "за кулисами" вступает в действие класс `Array`, предоставляющий несколько методов для сортировки и фильтрации элементов внутри массива.

Применяя перечислитель (enumerator), вы можете выполнить итерацию по всем элементам массива.

В настоящей главе речь пойдет о следующих моментах:

- простые массивы;
- многомерные массивы;
- зубчатые (jagged) массивы;
- класс `Array`;
- интерфейсы массивов;
- перечисления.

Простые массивы

Если вам нужно работать с множеством объектов одного и того же типа, вы можете использовать массив. Массив — это структура данных, содержащая множество элементов одного и того же типа.

Объявление массива

Массив объявляется определением типа элементов, содержащихся в нем, за которым следуют квадратные скобки и имя переменной; например, массив, содержащий целочисленные элементы, объявляется так:

```
int[] myArray;
```

Инициализация массива

После объявления массива должна быть выделена память для хранения всех элементов этого массива. Массив — ссылочный тип, поэтому память для него должна распределяться в куче. Это делается инициализацией переменной массива операцией `new` с указанием типа и числа элементов в массиве. Вот как специфицируется размер массива:

```
myArray = new int[4];
```

Типы значений и ссылочные типы описаны в главе 3.

После такого объявления и инициализации переменная `myArray` ссылается на четыре целочисленных значения, распределенные в управляемой куче (рис. 5.1).

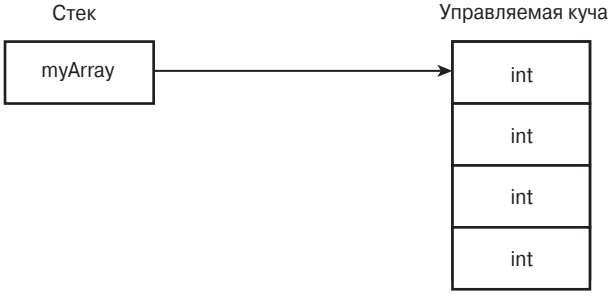

Рис. 5.1. Переменная `myArray` ссылается на четыре целочисленных значения в управляемой куче

Размер массива не может быть изменен после того, как он специфицирован изначально, без копирования всех элементов. Если вы заранее не знаете, сколько элементов должно содержаться в массиве, можете использовать коллекцию. Коллекции описаны в главе 10.

Вместо размещения отдельных строк для объявления и инициализации вы можете объявлять и инициализировать массив в единственной строке:

```
int[] myArray = new int[4];!
```

Также вы можете присвоить значения каждому элементу массива, применив инициализатор массива. Инициализаторы массивов могут быть использованы прямо при объявлении переменной массива, а не после того, как массив объявлен:

```
int[] myArray = new int[4] {4, 7, 11, 2};
```

Если вы инициализируете массив, применяя фигурные скобки, то размер массива также можно не указывать, поскольку компилятор может определить его самостоятельно:

```
int[] myArray = new int[] {4, 7, 11, 2};
```

При использовании компилятора C# существует еще более короткая форма. Применяя фигурные скобки, вы можете написать объявление и инициализацию массива. Код, сгенерированный компилятором в следующем случае, будет таким же, как и в предыдущем:

```
int[] myArray = {4, 7, 11, 2};
```

Обращение к элементам массива

После того, как массив объявлен и инициализирован, вы можете обращаться к его элементам, применяя индексатор (indexer). Массивы поддерживают только целочисленные индексаторы.

В пользовательских классах вы можете также создавать индексаторы, поддерживающие другие типы. О создании таких индексаторов вы можете прочитать в главе 6.

Применяя индексатор для доступа к массиву, вы указываете номер элемента. Индексатор всегда начинается со значения 0 для первого элемента. Самый больший номер, который вы можете передать индексатору, равен общему количеству элементов в массиве минус единица, поскольку нумерация начинается с нуля. В следующем примере массив `myArray` объявляется и инициализируется четырьмя целыми значениями. К элементам можно обращаться со значениями индексатора 0, 1, 2 и 3.

```
int[] myArray = new int[] {4, 7, 11, 2};
int v1 = myArray[0];  // читать первый элемент
int v2 = myArray[1];  // читать второй элемент
myArray[3] = 44;      // изменить четвертый элемент
```

> Если вы используете неверное значение индексатора, которому не соответствует никакой элемент, возбуждается исключение типа `IndexOutOfRangeException`.

Если вы не знаете количества элементов в массиве, то можете использовать свойство `Length`, как в приведенном ниже операторе:

```
for (int i = 0; i < myArray.Length; i++)
{
   Console.WriteLine(myArray[i]);
}
```

Вместо применения оператора `for` для прохождения по всем элементам массива вы можете также использовать оператор `foreach`:

```
foreach (int val in myArray)
{
   Console.WriteLine(val);
}
```

Оператор `foreach` использует интерфейсы `IEnumerable` и `Ienumerator`, о которых мы поговорим в этой главе далее.

Использование ссылочных типов

Вы можете объявлять массивы не только предопределенных типов; можно также объявлять массивы типа пользовательского класса. Начнем с класса `Person` с двумя конструкторами, свойствами `Firstname` и `Lastname`, использующими автоматически реализуемые свойства, и переопределенным методом `ToString()` из класса `Object`:

```
public class Person
{
  public Person()
  {
  }
  public Person(string firstname, string lastname)
  {
     this.Firstname = firstname;
     this.Lastname = lastname;
  }
  public string Firstname { get; set }
  public string Lastname { get; set }
  public override string ToString()
  {
     return String.Format("{0} {1}", FirstName, LastName);
  }
}
```

Объявление массива из двух элементов `Person` подобно объявлению массива целых чисел:

```
Person[] myPersons = new Person[2];
```

Однако следует помнить, что если элементы массива относятся к ссылочному типу, то для каждого из них должна быть выделена память. Если вы обратитесь к элементу массива, для которого память не распределялась, возбуждается исключение `NullReferenceException`.

Всю необходимую информацию об исключениях и ошибках вы найдете в главе 14.

Можно выделить память для каждого элемента, используя индексатор, начинающийся с 0:

```
myPersons[0] = new Person("Ayrton", "Senna");
myPersons[1] = new Person("Michael", "Schumacher");
```

На рис. 5.2 показаны объекты в управляемой куче, относящиеся к массиву `Person`. Переменная `myPersons` сохраняется в стеке. Эта переменная ссылается на массив элементов `Person`, хранящихся в управляемой куче. Этот массив имеет достаточно места для двух ссылок. Каждый элемент массива ссылается на объект `Person`, также находящийся в управляемой куче.

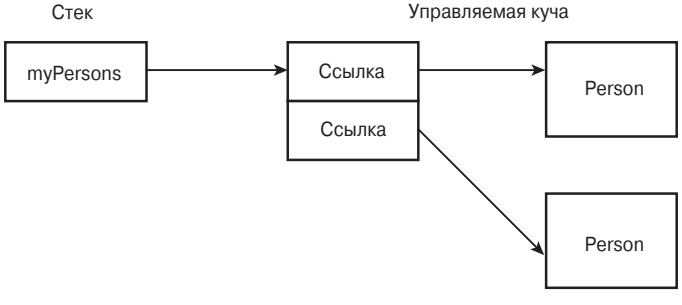

Рис. 5.2. Объекты в управляемой куче, относящиеся к массиву `Person`

Как и в случае с типом `int`, инициализатор массива можно также применять с пользовательским типом:

```
Person[] myPersons = { new Person("Ayrton", "Senna"),
   new Person("Michael", "Schumacher") };
```

Многомерные массивы

Обычные массивы (также известные, как одномерные массивы) индексируются единственным целым числом. Многомерный массив индексируется двумя или более целыми числами.

На рис. 5.3 показано математическое обозначение двумерного массива, имеющего три строки и три столбца. Первая строка содержит значения 1, 2 и 3, а третья – 7, 8 и 9.

$$A = \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 9 \end{bmatrix}$$

Рис. 5.3. Математическое обозначение двумерного массива

Объявление двумерного массива на C# выполняется помещением точки с запятой внутри квадратных скобок. Массив инициализируется указанием размера каждого измерения (также известного, как ранг). Затем к элементам массива можно обращаться с применением двух целых чисел в индексаторе:

```
int[,] twodim = new int[3, 3];
twodim[0, 0] = 1;
twodim[0, 1] = 2;
twodim[0, 2] = 3;
twodim[1, 0] = 4;
twodim[1, 1] = 5;
twodim[1, 2] = 6;
twodim[2, 0] = 7;
twodim[2, 1] = 8;
twodim[2, 2] = 9;
```

После объявления массива изменить его ранг невозможно.

Также вы можете инициализировать двумерный массив, используя индексатор массива, если заранее известно количество элементов. Для инициализации массива применяется одна внешняя пара фигурных скобок, и каждая строка инициализируется с использованием фигурных скобок, расположенных внутри этой внешней пары скобок.

```
int[,] twodim = {
                 {1, 2, 3},
                 {4, 5, 6},
                 {7, 8, 9}
                }
```

При использовании инициализатора массива вы должны инициализировать каждый его элемент. Невозможно опустить инициализацию некоторых значений.

Используя две точки с запятой внутри фигурных скобок, вы можете объявить трехмерный массив:

```
int[,,] threedim = {
                     { { 1, 2 }, { 3, 4 } },
                     { { 5, 6 }, { 7, 8 } },
                     { { 9, 10 }, { 11, 12 } }
                   };
Console.WriteLine(threedim[0, 1, 1]);
```

Зубчатые массивы

Двумерный массив имеет прямоугольную форму (например, размером 3 на 3 элемента). Зубчатый (jagged) массив более гибок в отношении размерности. В таких массивах каждая строка может иметь отличающийся размер.

На рис. 5.4 демонстрируется отличие обычного двумерного массива от массива зубчатого. Показанный здесь зубчатый массив содержит три строки, причем первая строка имеет два элемента, вторая — шесть элементов, а третья — три элемента.

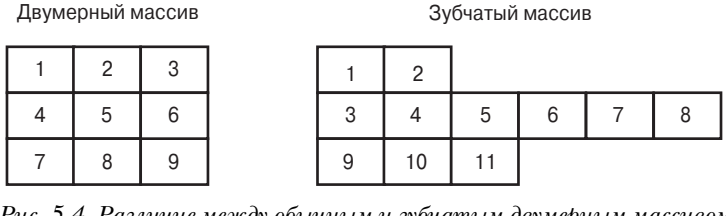

Рис. 5.4. Различие между обычным и зубчатым двумерным массивом

Зубчатый массив объявляется размещением открывающихся и закрывающихся квадратных скобок друг за другом. При инициализации зубчатого массива, устанавливается только размер, определяющий количество строк, в первой паре квадратных скобок. Вторая пара квадратных скобок, определяющая количество элементов внутри строки, остается пустой, поскольку каждая строка может содержать отличающееся количество элементов. Затем для каждой строки может быть указано количество ее элементов:

```
int[][] jagged = new int[3][];
jagged[0] = new int[2] { 1, 2 };
jagged[1] = new int[6] { 3, 4, 5, 6, 7, 8 };
jagged[2] = new int[3] { 9, 10, 11 };
```

Итерация по всем элементам зубчатого массива может осуществляться вложенными циклами `for`. Во внешнем цикле `for` выполняется проход по всем строкам, а во внутренних циклах `for` — проход по каждому элементу строки:

```
for (int row = 0; row < jagged.Length; row++)
{
  for (int element = 0; element < jagged[row].Length; element++)
  {
       Console.WriteLine("строка: {0}, элемент: {1}, значение: {2}",
            row, element, jagged[row][element]);
  }
}
```

Вывод этой итерации отображает строки и каждый элемент внутри строк:

```
строка: 0, элемент: 0, значение: 1
строка: 0, элемент: 1, значение: 2
строка: 1, элемент: 0, значение: 3
строка: 1, элемент: 1, значение: 4
строка: 1, элемент: 2, значение: 5
строка: 1, элемент: 3, значение: 6
строка: 1, элемент: 4, значение: 7
строка: 1, элемент: 5, значение: 8
строка: 2, элемент: 1, значение: 9
строка: 2, элемент: 2, значение: 10
строка: 2, элемент: 3, значение: 11
```

Класс `Array`

Объявление массива с квадратными скобками — это нотация C#, использующая класс `Array`. Такой синтаксис C# приводит к созданию "за кулисами" нового класса, унаследованного от абстрактного базового класса `Array`. Таким образом, методы и свойства, определенные в классе `Array`, можно использовать с любым массивом C#. Например, вы уже применяли свойство `Length` и итерацию по элементам с помощью оператора `foreach`. Делая так, вы используете метод `GetEnumerator()` класса `Array`.

Свойства

Класс `Array` содержит описанные в табл. 5.1 свойства, которые вы можете использовать с каждым экземпляром массива. Есть еще несколько свойств, о которых мы поговорим позднее в настоящей главе.

Таблица 5.1. Некоторые свойства класса `Array`

Свойство	Описание
`Length`	Свойство `Length` возвращает количество элементов в массиве. Если массив является многомерным, то вы получите число элементов по всем измерениям. Если вам нужно знать число элементов внутри измерения, вы можете использовать вместо этого метод `GetLength()`.
`LongLength`	Свойство `Length` возвращает значение типа `int`, а свойство `LongLength` — длину в виде значения `long`.
`Rank`	Свойство `Rank` позволяет получить количество измерений массива.

Создание массивов

Класс `Array` является абстрактным, поэтому вы не можете создать массив, используя его конструктор. Однако вместо применения синтаксиса C# для создания экземпляров массивов также возможно создавать их с помощью статического метода `CreateInstance()`. Это исключительно удобно, когда заранее не известен тип элементов массива, поскольку тип можно передать методу `CreateInstance()` в параметре в виде объекта `Type`.

В следующем примере демонстрируется создание массива типа `int` размером 5. Первый аргумент метода `CreateInstance()` требует тип элементов, а второй — определяет размер. Вы можете устанавливать значения методом SetValue(), а читать — методом `GetValue()`.

```
Array intArray1 = Array.CreateInstance(typeof(int), 5);
for (int i = 0; i < 5; i++)
{
   intArray1.SetValue(33, i);
}
for (int i = 0; i < 5; i++)
{
   Console.WriteLine(intArray1.GetValue(i));
}
```

Также вы можете привести созданный массив к типу массива, объявленного как `int[]`.

```
int[] intArray2 = (int[])intArray1;
```

Метод `CreateInstance()` имеет множество перегрузок для создания многомерных массивов, а также для создания массивов с индексацией, не начинающейся с 0. В следующем примере создается двумерный массив размером 2×3 элемента. База первого измерения — 1, а второго — 10.

```
int[] lengths = { 2, 3 };
int[] lowerBounds = { 1, 10 };
Array racers = Array.CreateInstance(typeof(Person), lengths, lowerBounds);
```

Устанавливающий элементы массива метод `SetValue()` принимает индексы каждого измерения:

```
racers.SetValue(new Person("Alain", "Prost"), 1, 10);
racers.SetValue(new Person("Emerson", "Fittipaldi"), 1, 11);
racers.SetValue(new Person("Ayrton", "Senna"), 1, 12);
racers.SetValue(new Person("Ralf", "Schumacher"), 2, 10);
racers.SetValue(new Person("Fernando", "Alonso"), 2, 11);
racers.SetValue(new Person("Jenson", "Button"), 2, 12);
```

Хотя массив не базируется на 0, вы можете присваивать его переменной в обычной нотации C#. Следует лишь обращать внимание на то, чтобы не выходить за границы индексов.

```
Person[,] racers2 = (Person[,])racers;
Person first = racers2[1, 10];
Person last = racers2[2, 12];
```

Копирование массивов

Поскольку массивы — это ссылочные типы, присвоение переменной массива другой переменной создает две переменных, ссылающихся на один и тот же массив. Для копирования массивов предусмотрена реализация массивами интерфейса `ICloneable`. Метод `Clone()`, определенный в этом интерфейсе, создает неглубокую (shallow) копию массива.

Если элементы массива относятся к типу значений, как в следующем сегменте кода, то все они копируются, как показано на рис. 5.5.

```
int[] intArray1 = {1, 2};
int[] intArray2 = (int[])intArray1.Clone();
```

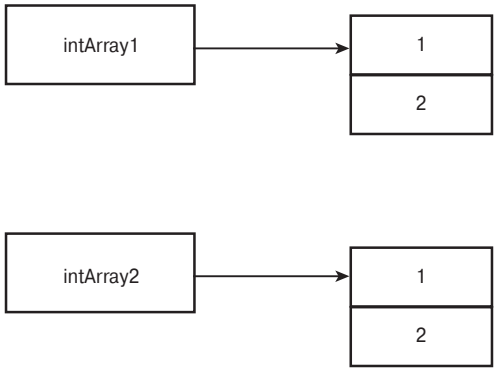

Рис. 5.5. Копирование массива с элементами типа значений

Если массив содержит элементы ссылочных типов, то сами эти элементы не копируются, а копируются лишь ссылки на них. На рис. 5.6 показана переменная `beatles` и переменная `beatlesClone`, созданная методом `Clone()` из `beatles`. Объекты `Person`, на которые ссылаются `beatles` и `beatlesClone`, одни и те же. Если вы измените свойство элемента, относящегося к `beatlesClone`, то тем самым измените объект, относящийся и к `beatles`.

```
Person[] beatles = {
                    new Person("John", "Lennon"),
                    new Person("Paul", "McCartney")
                   };
Person[] beatlesClone = (Person[])beatles.Clone();
```

Вместо использования метода `Clone()` вы также можете применять метод `Array.Copy()`, тоже создающий поверхностную копию. Но есть одно важное отличие между `Clone()` и `Copy()`: `Clone()` создает новый массив, а `Copy()` требует наличия существующего массива той же размерности с достаточным количеством элементов.

Если вам нужно глубокое копирование массива, содержащего ссылочные типы, вам придется выполнить итерацию по объектам исходного массива с созданием новых объектов.

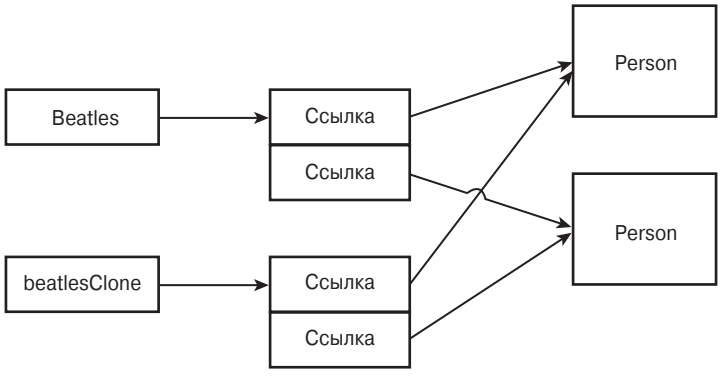

Рис. 5.6. Копирование массива с элементами ссылочного типа

Сортировка

В классе `Array` реализована пузырьковая сортировка элементов массива. Метод `Sort()` требует от элементов реализации интерфейса `IComparable`. Простые типы, такие как `System.String` и `System.Int32`, реализуют `IComparable`, так что вы можете сортировать элементы, относящиеся к этим типам.

В следующем примере программы создается массив `names`, содержащий элементы типа `string`, и этот массив может быть отсортирован.

```
string[] names = {
                    "Christina Aguillera",
                    "Shakira",
                    "Beyonce",
                    "Gwen Stefani"
                 };
Array.Sort(names);
foreach (string name in names)
{
   Console.WriteLine(name);
}
```

Вывод этого кода показывает отсортированное содержимое массива:

```
Beyonce
Christina Aguillera
Gwen Stefani
Shakira
```

Если вы используете с массивом собственные классы, то должны реализовать интерфейс `IComparable`. Этот интерфейс определяет единственный метод `CompareTo()`, который должен возвращать 0, если сравниваемые объекты эквивалентны, значение меньше 0, если данный экземпляр должен следовать перед объектом, переданным в параметре, и значение больше 0, если экземпляр должен следовать за объектом, переданным в параметре.

Изменим класс `Person` так, чтобы он реализовывал интерфейс `IComparable`. Сравнение будет выполняться по значению `lastname`. Поскольку `lastname` имеет тип `string`, а класс `String` уже реализует интерфейс `IComparable`, вы можете положиться на его реализацию метода `CompareTo()`. Если `LastName` совпадают, то сравниваются `FirstName`:

```
public class Person : IComparable
{
  public int CompareTo(object obj)
  {
     Person other = obj as Person;
     int result = this.LastName.CompareTo(other.LastName);
     if (result == 0)
     {
        result = this.FirstName.CompareTo(
        other.FirstName);
     }
     return result;}
     //...
```

Теперь можно отсортировать массив объектов `Person` по значению фамилии:

```
Person[] persons = {
                    new Person("Emerson", "Fittipaldi"),
                    new Person("Niki", "Lauda"),
                    new Person("Ayrton", "Senna"),
                    new Person("Michael", "Schumacher")
                   };
Array.Sort(persons);
foreach (Person p in persons)
{
   Console.WriteLine(p);
}
```

Вывод отсортированного массива элементов `Person` будет выглядеть так:

```
Emerson Fittipaldi
Niki Lauda
Michael Schumacher
Ayrton Senna
```

Если объекты `Person` понадобится отсортировать как-то иначе, либо если у вас нет возможности изменить класс, используемый в качестве элемента массива, вы можете реализовать интерфейс `IComparer`. Этот интерфейс определяет метод `Compare()`. Интерфейс `IComparable` должен быть реализован классом, подлежащим сравнению. Интерфейс `IComparer` независим от сравниваемого класса. Вот почему метод `Compare()` принимает два аргумента, которые подлежат сравнению. Возвращаемое значение подобно тому, что возвращает метод `CompareTo()` интерфейса `IComparable`.

Класс `PersonComparer` реализует интерфейс `IComparer` для сортировки объектов `Person` либо по `firstname`, либо по `lastname`. Перечисление `PersonCompareType` определяет различные опции сортировки, которые доступны в `PersonComparer`: `FirstName` и `LastName`. Способ сравнения определяется конструктором класса `PersonComparer`, в котором устанавливается значение `PersonCompareType`. Метод `Compare()` реализован с оператором `switch` для сравнения либо по `Lastname`, либо по `Firstname`.

```
public class PersonComparer : IComparer
{
  public enum PersonCompareType
  {
     Firstname,
     Lastname
  }
  private PersonCompareType compareType;
```

```
  public PersonComparer(PersonCompareType compareType)
  {
    this.compareType = compareType;
  }
  public int Compare(object x, object y)
  {
    Person p1 = x as Person;
    Person p2 = y as Person;
    switch (compareType)
    {
    case PersonCompareType.Firstname:
      return p1.Firstname.CompareTo(p2.Firstname);
    case PersonCompareType.Lastname:
      return p1.Lastname.CompareTo(p2.Lastname);
    default:
      throw new ArgumentException("недопустимый тип для сравнения");
    }
  }
}
```

Теперь вы можете передать объект `PersonComparer` в качестве второго аргумента метода `Array.Sort()`. Так персоны сортируются по фамилии:

```
Array.Sort(persons,
   new PersonComparer(PersonComparer.PersonCompareType.Firstname));
foreach (Person p in persons)
{
   Console.WriteLine(p);
}
```

Как видим, в результате получится список лиц, отсортированных по имени:

```
Ayrton Senna
Emerson Fittipaldi
Michael Schumacher
Niki Lauda
```

Класс `Array` также предлагает методы `Sort`, требующие в качестве аргумента делегата. Всю информацию об использовании делегатов вы найдете в главе 7.

Интерфейсы `Array` и `Collection`

Класс `Array` реализует интерфейсы `IEnumerable`, `ICollection` и `IList`, предназначенные для перечисления элементов в массиве. Поскольку с пользовательским массивом создается класс, унаследованный от абстрактного класса `Array`, вы можете использовать с переменной массива методы и свойства реализованных интерфейсов.

`IEnumerable`

`IEnumerable` — это интерфейс, используемый оператором `foreach` для выполнения итерации по массиву. Поскольку это достаточно специализированное средство, мы обсудим его в следующем разделе “Перечисления”.

`ICollection`

Интерфейс `ICollection` наследуется от интерфейса `IEnumerable` и имеет дополнительные свойства и методы, как показано в табл. 5.2. Этот интерфейс в основном используется для получения количества элементов в коллекции и для синхронизации.

Таблица 5.2. Свойства и методы интерфейса `ICollection`

Свойства и методы	Описание
`Count`	Свойство `Count` хранит количество элементов внутри коллекции. Возвращает то же самое значение, что и свойство `Length`.
`IsSynchronized` `SyncRoot`	Свойство `IsSynchronized` определяет, является ли коллекция безопасной в отношении потоков. Для массивов это свойство всегда возвращает `false`. Для синхронизированного доступа свойство `SyncRoot` может использоваться для доступа, безопасного к потокам. Оба эти свойства определены с интерфейсом `ICollection`. Потоки и синхронизация объясняются в главе 19, и в ней вы можете прочитать о том, как реализовать потоковую безопасность коллекций.
`CopyTo()`	Методом `CopyTo()` вы можете копировать элементы массива в другой существующий массив. Этот метод подобен статическому методу `Array.Copy()`.

IList

Интерфейс `IList` порожден от интерфейса `ICollection`, и определяет дополнительные свойства и методы. Главная причина того, что класс `Array` реализует интерфейс `IList`, состоит в том, что этот интерфейс определяет свойство `Item` для доступа к элементам через индексатор. Многие другие члены `IList` реализованы классом `Array` через простое возбуждение исключения `NotSupportedException`, поскольку они не применимы к массивам. Все свойства и методы интерфейса `IList` описаны в табл. 5.3.

Таблица 5.3. Свойства и методы интерфейса `IList`

Свойства и методы	Описание
`Add()`	Метод `Add()` используется для добавления элементов в коллекцию. Для массивов этот метод возбуждает исключение `NotSupportedException`.
`Clear()`	Метод `Clear()` очищает элементы массива. Типы значений устанавливаются в 0, а ссылочные — в `null`.
`Contains()`	С помощью метода `Contains()` вы можете узнать, содержится ли элемент в массиве. Его возвращаемое значение — `true` или `false`. Этот метод выполняет линейный поиск по всем элементам массива, до тех пор, пока не найдет нужный элемент.
`IndexOf()`	Метод `IndexOf()` выполняет линейный поиск по всем элементам массива, подобно тому, как это делает метод `Contains()`. Отличие состоит в том, что `IndexOf()` возвращает индекс первого найденного элемента.

`Insert()`
`Remove()`
`RemoveAt()`
	С коллекциями метод `Insert()` применяется для вставки элементов; методами `Remove()` и `RemoveAt()` элементы могут быть удалены. Для массивов эти методы возбуждают исключение `NotSupportedException`.

`IsFixedSize`	Поскольку массивы всегда имеют фиксированный размер, это свойство всегда возвращает `true`.
`IsReadOnly`	Массивы всегда доступны по чтению/записи, поэтому данное свойство возвращает `false`. В главе 10 вы прочтете о том, как из массивов создаются коллекции, доступные только для чтения.
`Item`	Свойство `Item` позволяет осуществлять доступ к массиву с использованием индекса.

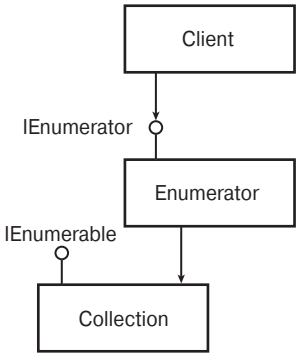

Рис. 5.7. Отношение между клиентом, вызывающим `foreach`, и коллекцией

Перечисления

Применяя оператор `foreach`, вы можете выполнять итерацию по элементам коллекции без необходимости знания количества ее элементов. Оператор `foreach` использует перечислитель (enumerator). На рис. 5.7 показано отношение между клиентом, вызывающим `foreach`, и коллекцией. Массив или коллекция реализует интерфейс `IEnumerable` с методом `GetEnumerator()`. Метод `GetEnumerator()` возвращает перечислитель, реализующий интерфейс `IEnumerable`. Интерфейс `IEnumerable` затем применяется оператором `foreach` для выполнения итерации по элементам коллекции.

Метод `GetEnumerator()` определен в интерфейсе `IEnumerable`. Оператор `foreach` в действительности не нуждается в том, чтобы класс коллекции реализовывал этот интерфейс. Достаточно иметь метод по имени `GetEnumerator()`, который возвращает объект, реализующий интерфейс `IEnumerator`.

Интерфейс `IEnumerator`

Оператор `foreach` использует методы и свойства интерфейса `IEnumerator` для итерации по всем элементам коллекции. Свойства и методы этого интерфейса перечислены в табл. 5.4.

Таблица 5.4. Свойства и методы интерфейса `IEnumerator`

Свойства и методы	Описание
`MoveNext()`	Метод `MoveNext()` переходит на следующий элемент коллекции и возвращает `true`, если таковой имеется. Если же коллекция не содержит более элементов, возвращается значение `false`.
`Current`	Свойство `Current` возвращает элемент, на котором в данный момент позиционирован курсор.
`Reset()`	Метод `Reset()` позиционирует курсор в начало коллекции. Многие перечислители возбуждают исключение `NotSupportedException`.

Оператор `foreach`

Оператор `foreach` не преобразуется к оператору `foreach` в коде IL. Вместо этого компилятор C# преобразует оператор `foreach` в методы и свойства интерфейса `IEnumerable`. Ниже приведен простой пример оператора `foreach` для итерации по всем элементам массива персон и отображения их одну за другой.

```
foreach (Person p in persons)
{
   Console.WriteLine(p);
}
```

Оператор `foreach` преобразуется в следующий сегмент кода. Сначала вызывается метод `GetEnumerator()` для получения перечислителя для массива. Внутри цикла `while` — до тех пор, пока `MoveNext()` возвращает `true` — элементы массива доступны через свойство `Current`:

```
IEnumerator enumerator = persons.GetEnumerator();
while (enumerator.MoveNext())
{
  Person p = (Person)enumerator.Current;
  Console.WriteLine(p);
}
```

Оператор `yield`

C# 1.0 позволяет легко выполнять итерации по коллекциям с помощью оператора `foreach`. В C# 1.0 приходилось выполнить немалую работу, чтобы получить перечислитель. В C# 2.0 добавлен оператор `yield` для легкого создания перечислителей.

`Yield return` возвращает один элемент коллекции и перемещает текущую позицию на следующий элемент, а `yield break` прекращает итерацию.

В следующем примере демонстрируется реализация простой коллекции с применением оператора `yield return`. Класс `HelloCollection` содержит метод `GetEnumerator()`. Реализация метода `GetEnumerator()` содержит два оператора `yield return`, где возвращаются строки `"Hello"` и `"World"`.

```
using System;
using System.Collections;
namespace Wrox.ProCSharp.Arrays
{
  public class HelloCollection
  {
    public IEnumerator GetEnumerator()
    {
      yield return "Hello";
      yield return "World";
    }
  }
```

Метод или свойство, содержащее операторы `yield`, также известно как блок итератора. Блок итератора должен быть объявлен для возврата интерфейса `IEnumerator` или `IEnumerable`. Этот блок может содержать множество операторов `yield return` или `yield break`; оператор `return` не допускается.

Теперь возможно выполнить итерацию по коллекции с использованием оператора `foreach`:

```
  public class Program
  {
    HelloCollection helloCollection = new HelloCollection();
    foreach (string s in helloCollection)
    {
      Console.WriteLine(s);
    }
  }
}
```

С блоком итератора компилятор генерирует `yield type`, включая конечный автомат, как показано в следующем фрагменте кода. `yield type` реализует свойства и методы интерфейсов `IEnumerator` и `IDisposable`. В примере вы можете увидеть `yield type` как внутренний класс `Enumerator`. Метод `GetEnumerator()` внешнего класса создает экземпляр и возвращает `yield type`. Внутри `yield type` переменная `state` определяет текущую позицию итерации и изменяется каждый раз, когда вызывается метод `MoveNext()`. Метод `MoveNext()` инкапсулирует код блока итератора и устанавливает значение текущей переменной таким образом, что свойство `Current` возвращает объект, зависящий от позиции.

```
public class HelloCollection
{
 public IEnumerator GetEnumerator()
 {
  Enumerator enumerator = new Enumerator();
  return enumerator;
 }
 public class Enumerator : IEnumerator, IDisposable
 {
  private int state;
  private object current;
  public Enumerator(int state)
  {
    this.state = state;
  }
  bool System.Collections.IEnumerator.MoveNext()
  {
    switch (state)
    {
       case 0:
         current = "Hello";
         state = 1;
         return true;
       case 1:
         current = "World";
         state = 2;
         return true;
       case 2:
         break;
     }
    return false;
  }
  void System.Collections.IEnumerator.Reset()
  {
    throw new NotSupportedException();
  }
  object System.Collections.IEnumerator.Current
  {
    get
    {
       return current;
    }
  }
  void IDisposable.Dispose()
  {
  }
 }
}
```

Теперь, используя оператор `yield return`, легко реализовать класс, позволяющий выполнять итерацию по коллекции различными способами. Класс `MusicTitles` позволяет итерацию по наименованиям способом по умолчанию — методом `GetEnumerator()`, в обратном порядке — методом `Reverse()`, и итерацию по подмножеству методом `Subset()`.

```
public class MusicTitles
{
 string[] names = {
     "Tubular Bells", "Hergest Ridge",
     "Ommadawn", "Platinum" };
```

```
public IEnumerator GetEnumerator()
{
 for (int i = 0; i < 4; i++)
 {
    yield return names[i];
 }
}
public IEnumerable Reverse()
{
 for (int i = 3; i >= 0; i—)
 {
    yield return names[i];
 }
}
public IEnumerable Subset(int index, int length)
{
 for (int i = index; i < index + length; i++)
 {
    yield return names[i];
 }
}
}
```

Клиентский код для того, чтобы выполнить итерацию по массиву строк, сначала использует метод `GetEnumerator()`, который вам не нужно писать в коде, поскольку он используется по умолчанию. Затем заголовки итерируются в обратном порядке и, наконец, выполняется итерация по подмножеству посредством передачи индекса и количества элементов для итерации методу `Subset()`:

```
MusicTitles titles = new MusicTitles();
foreach (string title in titles)
{
   Console.WriteLine(title);
}
Console.WriteLine();
Console.WriteLine("reverse");
foreach (string title in titles.Reverse())
{
   Console.WriteLine(title);
}
Console.WriteLine();
Console.WriteLine("subset");
foreach (string title in titles.Subset(2, 2))
{
   Console.WriteLine(title);
}
```

С помощью оператора `yield` вы можете также делать и более сложные вещи, например, возвращать перечислитель из `yield return`.

В игре "крестики-нолики" у вас есть девять полей, где игроки-соперники расставляют крестики или нолики. Эти ходы имитируются классом `GameMoves`. Метод `Cross()` и `Circle()` — это блоки итератора для создания типов итераторов. Переменные `cross` и `circle` устанавливаются в `Cross()` и `Circle()` внутри конструктора класса `GameMoves`. Для установки этих полей методы не вызываются, но устанавливаются в типы итераторов, определенные в блоках итераторов. Внутри блока итератора `Cross()` информация о ходах записывается на консоль и номер хода увеличивается. Если номер хода больше 9, итератор завершается с помощью `yield break`; в противном случае на каждой итерации возвращается объект перечислителя `circle`. Блок итератора `Circle()` очень похож на блок итератора `Cross()`, только он возвращает на каждой итерации объект перечислителя `cross`.

```
public class GameMoves
{
 private IEnumerator cross;
 private IEnumerator circle;
 public GameMoves()
 {
  cross = Cross();
  circle = Circle();
 }
 private int move = 0;
 public IEnumerator Cross()
 {
  while (true)
  {
     Console.WriteLine("Крестик, ход {0}", move);
     move++;
     if (move > 9)
       yield break;
     yield return circle;
  }
 }
 public IEnumerator Circle()
 {
  while (true)
  {
     Console.WriteLine("Нолик, ход {0}", move);
     move++;
     if (move > 9)
       yield break;
     yield return cross;
  }
 }
}
```

В клиентской программе вы можете использовать класс `GameMoves`, как показано ниже. Первый ход выполняется установкой перечислителя в тип перечислителя, возвращенный `game.Cross()`. Метод `enumerator.MoveNext` вызывает одну итерацию, определенную блоком итератора, возвращающим другой перечислитель. Возвращенное значение можно получить через свойство `Current` и затем оно устанавливается в переменную `enumerator` для следующего шага цикла:

```
GameMoves game = new GameMoves();
IEnumerator enumerator = game.Cross();
while (enumerator.MoveNext())
{
   enumerator = (IEnumerator)enumerator.Current;
}
```

Вывод этой программы показывает все ходы игроков до самого последнего:

```
Крестик, ход 0
Нолик, ход 1
Крестик, ход 2
Нолик, ход 3
Крестик, ход 4
Нолик, ход 5
Крестик, ход 6
Нолик, ход 7
Крестик, ход 8
```

Резюме

В этой главе вы ознакомились с нотацией C# для создания и использования простых, многомерных и зубчатых массивов. Класс `Array` используется "за кулисами" массивов C#, и таким образом вы можете вызывать свойства и методы этого класса с переменными массива.

Вы увидели, как сортировать элементы массива с использованием интерфейсов `IComparable` и `IComparer`. Мы описали реализацию средств интерфейсов `IEnumerable`, `ICollection` и `IList`, как они реализованы в классе `Array`, и, наконец, вы убедились в преимуществах оператора `yield`.

Двигаясь дальше, в следующей главе мы сосредоточим внимание на операциях и приведениях, и там же вы узнаете о создании специализированных индексаторов. Глава 7 предоставляет информацию о делегатах и событиях. Некоторые методы класса `Array` используют делегаты в качестве параметров. Глава 10 посвящена классам коллекций, которые уже упомянуты в этой главе. Классы коллекций предлагают большую гибкость в отношении размера, и там же вы прочтете о других контейнерах, таких как словари и связные списки.