CHAPTER 16

Indexers

Indexers allow an object to be treated as an array. They are declared in the same way as properties, except that the `this` keyword is used instead of a name and their accessors take parameters. In the following example, the indexer corresponds to an object array called `data`, so the type of the indexer is set to `object`.

```
class MyArray
{
  object[] data = new object[10];
  public object this[int i]
  {
    get { return data[i]; }
    set { data[i] = value; }
  }
}
```

The `get` accessor returns the specified element from the object array, and the `set` accessor inserts the value into the specified element. With the indexer in place, an instance of this class can be created and used as an array, both to get and set the elements.

M. Olsson, *C# 8 Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-5577-3_16

```
static void Main()
{
  MyArray a = new MyArray();
  a[5] = "Hello World";
  object o = a[5]; // Hello World
}
```

Indexer Parameters

The parameter list of an indexer is similar to that of a method, except that it must have at least one parameter and the `ref` or `out` modifiers are not allowed. For example, if there is a two-dimensional array, the column and row indexes can be passed as separate parameters.

```
class MyArray
{
  object[,] data = new object[10, 10];
  public object this[int i, int j]
  {
    get { return data[i, j]; }
    set { data[i, j] = value; }
  }
}
```

The index parameter does not have to be of an integer type. An object can just as well be passed as the index parameter. The `get` accessor can then be used to return the index position where the passed object is located.

```
class MyArray
{
  object[] data = new object[10];
  public int this[object o]
  {
    get { return System.Array.IndexOf(data, o); }
  }
}
```

Indexer Overloading

Both of these functionalities can be provided by overloading the indexer. The type and number of arguments will then determine which indexer gets called.

```
class MyArray
{
  object[] data = new object[10];
  public int this[object o]
  {
    get { return System.Array.IndexOf(data, o); }
  }

  public object this[int i]
  {
    get { return data[i]; }
    set { data[i] = value; }
  }
}
```

Keep in mind that in a real program a range check should be included in the accessors, so as to avoid exceptions caused by trying to go beyond the length of the array.

```
public object this[int i]
{
  get {
    return (i >= 0 && i < data.Length) ? data[i] : null;
  }
  set {
    if (i >= 0 && i < data.Length)
      data[i] = value;
  }
}
```

Ranges and Indexes

C# 8.0 introduced two new operators for slicing collections such as arrays. The range operator (`x..y`) specifies the start and end index for a range of elements. The result of such an operation can be used directly in a loop or stored in the System.Range type.

```
int[] b = { 1, 2, 3, 4, 5 };
foreach (int n in b[1..3]) {
  System.Console.Write(n); // "23"
}

System.Range range = 0..3; // 1st to 3rd
foreach (int n in b[range]) {
  System.Console.Write(n); // "123"
}
```

The second operator introduced in C# 8.0 is named the hat operator (^). It is used as a prefix to count indexes starting from the end of the array. An index can be stored using the System.Index type.

```
string s = "welcome";
System.Index first = 0;
System.Index last = ^1;

System.Console.Write($"{s[first]}, {s[last]}"); // "w, e"
```

Both of these operators can be combined in the same expression as seen in the next example. Note that either the start or end point for the range operator can be left out to include all remaining elements.

```
string s = "welcome";
System.Console.Write(s[^4..]); // "come"
```