Collection types have been around in various forms since the dawn of programming. I’m sure you remember the linked list exercises when you were learning to write programs. In this chapter, I’ll give a brief overview of arrays but won’t go into much detail, as arrays have not changed much between the various .NET releases.

However, I’ll spend more time explaining the major generic collection interfaces and iterators along with what sorts of cool things you can do with them. Traditionally, creating enumerators for collection types has been tedious and annoying. Iterators make this task a breeze, while making your code a lot more readable in the process.

**Introduction to Arrays**

C# arrays, as well as arrays in the CLR, are highly evolved from C/C++ arrays. In C/C++, you typically access an array by offsetting a pointer that points to the beginning of a contiguous range of items in a memory block somewhere. C/C++ arrays have no built-in range checking, which is the root of more bugs than you can shake a stick at. C# and the CLR solve this problem elegantly by making the array type a built-in, implicit type to the runtime.

When you declare a type—whether it’s a class or struct—the runtime reserves the right to silently generate an array type based upon that new type. The array type that it generates is a reference type—thus, all array instances are of class type. The reference type that it generates is derived from `System.Array`, and ultimately from `System.Object`. Therefore, you can treat all C# arrays polymorphically through a reference to `System.Array`. Of course, that means that each array, no matter what concrete type of array it is, implements all of the methods and properties of `System.Array`.

The way that you declare an array within C# is similar to C/C++, except the designers of the language took the liberty to make the syntax a tad more intuitive in their minds, in that the square brackets in the declaration follow the type and not the array variable name. The following example shows three ways to create an array:

```csharp
using System;

public class EntryPoint
{
    static void Main()
    {
        int[] array1 = new int[10];
        for (int i = 0; i < array1.Length; ++i)
        {
            array1[i] = i*2;
        }
    }
}
```
The longhand way to create an array instance and fill it with initial values is shown where array1 is initialized. Items are indexed using an indexer that is typically greater than or equal to 0. You may know that arrays in the CLR can have a user-defined lower bound. However, in C#, the lower bound is always 0 in order to meet the CLS restriction that arrays have a 0 lower bound. The initialization techniques used for array2 and array3 show a shorter notation for doing the same thing. Notice that in most cases, you must first allocate the array instances on the heap using the new operator. The same thing happens with the array3 instance, but the compiler does it for you in order to facilitate the notational shorthand. It’s interesting to note that an array of type object—thus, System.Object[]—is itself of type System.Object.

One of the conveniences of .NET arrays is that they are range-checked. Therefore, if you step off the end of one of them, thus going out of bounds, the runtime will throw an IndexOutOfRangeException instead of changing random memory, as in native C/C++. So you can say goodbye to those lurking, hard-to-find range bugs, because the CLR won’t allow them to lurk too long, and they definitely won’t go unnoticed for long periods of time anymore.

Lastly, notice that you can conveniently iterate through the items in the array using the C# foreach statement. This works because System.Array implements IEnumerable. I have more to say about IEnumerable and its cousin IEnumerator later on, in the section titled “IEnumerable<T>, IEnumerator<T>, IEnumerable, and IEnumerator.”

Microsoft Visual Studio Professional 2010

Note The compiler actually uses an optimization while compiling the foreach loop which I mention in Chapter 13. Instead of actually converting the array to an IEnumerable instance and then calling IEnumerable.GetEnumerator, it instead simply calls any public GetEnumerator method that matches the required signature. This optimization is a form of what is known as duck typing, which I speak more about in Chapter 17. Even though foreach uses this optimization, it is always a good idea to implement IEnumerable and IEnumerable<T> because a consumer may wish to iterate over your collection without using foreach such as with LINQ.

Implicitly Typed Arrays

C# 3.0 introduced an abbreviated way of initializing arrays when the type of the array can be inferred at runtime. Let’s have a look at the new syntax in the following code snippet:

```csharp
using System;

public class EntryPoint
{
    static void Main()
    {
        // A conventional array
        int[] conventionalArray = new int[] { 1, 2, 3 };
```