CHAPTER 16

Arrays

In this chapter:

Arrays are mechanisms that allow you to treat several items as a single collection. The Microsoft .NET common language runtime (CLR) supports single-dimensional arrays, multi-dimensional arrays, and jagged arrays (that is, arrays of arrays). All array types are implicitly derived from the `System.Array` abstract class, which itself is derived from `System.Object`. This means that arrays are always reference types that are allocated on the managed heap and that your application's variable or field contains a reference to the array and not the elements of the array itself. The following code makes this clearer.

```
Int32[] myIntegers;          // Declares a reference to an array
myIntegers = new Int32[100]; // Creates an array of 100 Int32s
```

On the first line, `myIntegers` is a variable that's capable of pointing to a single-dimensional array of `Int32`s. Initially, `myIntegers` will be set to `null` because I haven't allocated an array. The second line of code allocates an array of 100 `Int32` values; all of the `Int32`s are initialized to 0. Because arrays are reference types, the memory block required to hold the 100 unboxed `Int32`s is allocated on the managed heap. Actually, in addition to the array's elements, the memory block occupied by an array object also contains a type object pointer, a sync block index, and some additional overhead members as well. The address of this array's memory block is returned and saved in the variable `myIntegers`.

You can also create arrays of reference types.

```
Control[] myControls;         // Declares a reference to an array
myControls = new Control[50]; // Creates an array of 50 Control references
```

On the first line, `myControls` is a variable capable of pointing to a single-dimensional array of `Control` references. Initially, `myControls` will be set to `null` because I haven't allocated an array. The second line allocates an array of 50 `Control` references; all of these references are initialized to `null`. Because `Control` is a reference type, creating the array creates only a bunch of references; the actual objects aren't created at this time. The address of this memory block is returned and saved in the variable `myControls`.

Figure 16-1 shows how arrays of value types and arrays of reference types look in the managed heap.

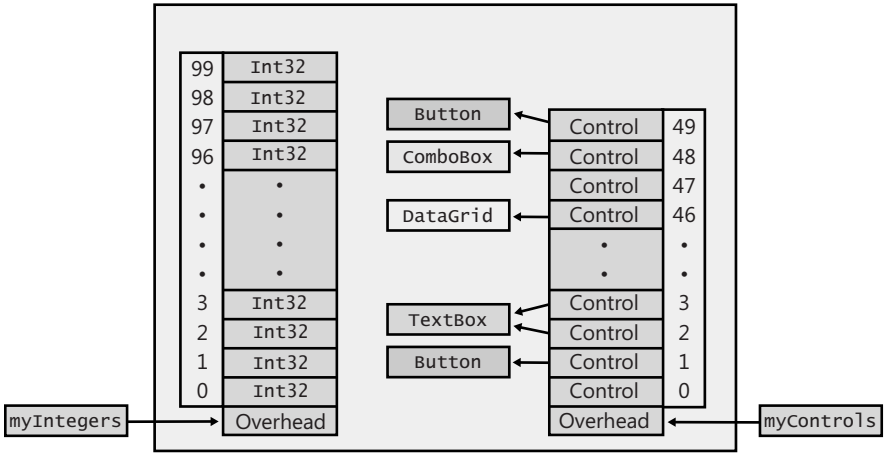

FIGURE 16-1 Arrays of value and reference types in the managed heap.

In the figure, the `Controls` array shows the result after the following lines have executed.

```
myControls[1]  = new Button();
myControls[2]  = new TextBox();
myControls[3]  = myControls[2];  // Two elements refer to the same object.
myControls[46] = new DataGrid();
myControls[48] = new ComboBox();
myControls[49] = new Button();
```

Common Language Specification (CLS) compliance requires all arrays to be zero-based. This allows a method written in C# to create an array and pass the array's reference to code written in another language, such as Microsoft Visual Basic .NET. In addition, because zero-based arrays are, by far, the most common arrays, Microsoft has spent a lot of time optimizing their performance. However, the CLR does support non-zero–based arrays even though their use is discouraged. For those of you who don't care about a slight performance penalty or cross-language portability, I'll demonstrate how to create and use non-zero–based arrays later in this chapter.

Notice in Figure 16-1 that each array has some additional overhead information associated with it. This information contains the rank of the array (number of dimensions), the lower bounds for each dimension of the array (almost always 0), and the length of each dimension. The overhead also contains

the array's element type. I'll mention the methods that allow you to query this overhead information later in this chapter.

So far, I've shown examples demonstrating how to create single-dimensional arrays. When possible, you should stick with single-dimensional, zero-based arrays, sometimes referred to as *SZ arrays*, or *vectors*. Vectors give the best performance because you can use specific Intermediate Language (IL) instructions—such as `newarr`, `ldelem`, `ldelema`, `ldlen`, and `stelem`—to manipulate them. However, if you prefer to work with multi-dimensional arrays, you can. Here are some examples of multi-dimensional arrays.

```
// Create a two-dimensional array of Doubles.
Double[,] myDoubles = new Double[10, 20];

// Create a three-dimensional array of String references.
String[,,] myStrings = new String[5, 3, 10];
```

The CLR also supports jagged arrays, which are arrays of arrays. Zero-based, single-dimensional jagged arrays have the same performance as normal vectors. However, accessing the elements of a jagged array means that two or more array accesses must occur. Here are some examples of how to create an array of polygons with each polygon consisting of an array of `Point` instances.

```
// Create a single-dimensional array of Point arrays.
Point[][] myPolygons = new Point[3][];

// myPolygons[0] refers to an array of 10 Point instances.
myPolygons[0] = new Point[10];

// myPolygons[1] refers to an array of 20 Point instances.
myPolygons[1] = new Point[20];

// myPolygons[2] refers to an array of 30 Point instances.
myPolygons[2] = new Point[30];

// Display the Points in the first polygon.
for (Int32 x = 0; x < myPolygons[0].Length; x++)
   Console.WriteLine(myPolygons[0][x]);
```

Note The CLR verifies that an index into an array is valid. In other words, you can't create an array with 100 elements in it (numbered 0 through 99) and then try to access the element at index –5 or 100. Doing so will cause a `System.IndexOutOfRangeException` to be thrown. Allowing access to memory outside the range of an array would be a breach of type safety and a potential security hole, and the CLR doesn't allow verifiable code to do this. Usually, the performance degradation associated with index checking is insubstantial because the just-in-time (JIT) compiler normally checks array bounds once before a loop executes instead of at each loop iteration. However, if you're still concerned about the performance hit of the CLR's index checks, you can use unsafe code in C# to access the array. The "Array Internals" section later in this chapter demonstrates how to do this.

Initializing Array Elements

In the previous section, I showed how to create an array object and then I showed how to initialize the elements of the array. C# offers syntax that allows you to do these two operations in one statement. The following shows an example.

```
String[] names = new String[] { "Aidan", "Grant" };
```

The comma-separated set of tokens contained within the braces is called an *array initializer*. Each token can be an arbitrarily complex expression or, in the case of a multi-dimensional array, a nested array initializer. In the preceding example, I used just two simple `String` expressions.

If you are declaring a local variable in a method to refer to the initialized array, then you can use C#'s implicitly typed local variable (`var`) feature to simplify the code a little.

```
// Using C#'s implicitly typed local variable feature:
var names = new String[] { "Aidan", "Grant" };
```

Here, the compiler is inferring that the `names` local variable should be of the `String[]` type because that is the type of the expression on the right of the assignment operator (=).

You can use C#'s implicitly typed array feature to have the compiler infer the type of the array's elements. Notice the following line has no type specified between `new` and `[]`.

```
// Using C#'s implicitly typed local variable and implicitly typed array features:
var names = new[] { "Aidan", "Grant", null };
```

In the preceding line, the compiler examines the types of the expressions being used inside the array to initialize the array's elements, and the compiler chooses the closest base class that all the elements have in common to determine the type of the array. In this example, the compiler sees two `String`s and `null`. Because `null` is implicitly castable to any reference type (including `String`), the compiler infers that it should be creating and initializing an array of `String` references.

If you had this code:

```
// Using C#'s implicitly typed local variable & implicitly typed array features: (error)
var names = new[] { "Aidan", "Grant", 123 };
```

the compiler would issue the message `error CS0826: No best type found for implicitly-typed array.` This is because the base type in common between the two `String`s and the `Int32` is `Object`, which would mean that the compiler would have to create an array of `Object` references and then box the `123` and have the last array element refer to a boxed `Int32` with a value of `123`. The C# compiler team thinks that boxing array elements is too heavy-handed for the compiler to do for you implicitly, and that is why the compiler issues the error.

As an added syntactical bonus when initializing an array, you can write the following.

```
String[] names = { "Aidan", "Grant" };
```

Notice that on the right of the assignment operator (=), only the array initializer expression is given with no new, no type, and no []s. This syntax is nice, but unfortunately, the C# compiler does not allow you to use implicitly typed local variables with this syntax.

```
// This is a local variable now (error)
var names = { "Aidan", "Grant" };
```

If you try to compile the preceding line of code, the compiler issues two messages: `error CS0820: Cannot initialize an implicitly-typed local variable with an array initializer` and `error CS0622: Can only use array initializer expressions to assign to array types. Try using a new expression instead.` Although the compiler could make this work, the C# team thought that the compiler would be doing too much for you here. It would be inferring the type of the array, new'ing the array, initializing the array, and inferring the type of the local variable, too.

The last thing I'd like to show you is how to use implicitly typed arrays with anonymous types and implicitly typed local variables. Anonymous types and how type identity applies to them are discussed in Chapter 10, "Properties." Examine the following code.

```
// Using C#'s implicitly typed local, implicitly typed array, and anonymous type features:
var kids = new[] {new { Name="Aidan" }, new { Name="Grant" }};

// Sample usage (with another implicitly typed local variable):
foreach (var kid in kids)
   Console.WriteLine(kid.Name);
```

In this example, I am using an array initializer that has two expressions for the array elements. Each expression represents an anonymous type (because no type name is specified after the new operator). Because the two anonymous types have the identical structure (one field called Name of type String), the compiler knows that these two objects are of the exact same type. Now, I use C#'s implicitly typed array feature (no type specified between the new and the []s) so that the compiler will infer the type of the array itself, construct this array object, and initialize its references to the two instances of the one anonymous type.[1] Finally, a reference to this array object is assigned to the kids local variable, the type of which is inferred by the compiler due to C#'s implicitly typed local variable feature.

I show the foreach loop as an example of how to use this array that was just created and initialized with the two anonymous type objects. I have to use an implicitly typed local variable (kid) for the loop, too. When I run this code, I get the following output.

```
Aidan
Grant
```

[1] If you think these sentences are fun to read, you can only imagine how fun they were to write in the first place!

Casting Arrays

For arrays with reference type elements, the CLR allows you to implicitly cast the source array's element type to a target type. For the cast to succeed, both array types must have the same number of dimensions, and an implicit or explicit conversion from the source element type to the target element type must exist. The CLR doesn't allow the casting of arrays with value type elements to any other type. (However, by using the `Array.Copy` method, you can create a new array and populate its elements in order to obtain the desired effect.) The following code demonstrates how array casting works.

```
// Create a two-dimensional FileStream array.
FileStream[,] fs2dim = new FileStream[5, 10];

// Implicit cast to a two-dimensional Object array
Object[,] o2dim = fs2dim;

// Can't cast from two-dimensional array to one-dimensional array
// Compiler error CS0030: Cannot convert type 'object[*,*]' to
// 'System.IO.Stream[]'
Stream[] s1dim = (Stream[]) o2dim;

// Explicit cast to two-dimensional Stream array
Stream[,] s2dim = (Stream[,]) o2dim;

// Explicit cast to two-dimensional String array
// Compiles but throws InvalidCastException at runtime
String[,] st2dim = (String[,]) o2dim;

// Create a one-dimensional Int32 array (value types).
Int32[] i1dim = new Int32[5];

// Can't cast from array of value types to anything else
// Compiler error CS0030: Cannot convert type 'int[]' to 'object[]'
Object[] o1dim = (Object[]) i1dim;

// Create a new array, then use Array.Copy to coerce each element in the
// source array to the desired type in the destination array.
// The following code creates an array of references to boxed Int32s.
Object[] ob1dim = new Object[i1dim.Length];
Array.Copy(i1dim, ob1dim, i1dim.Length);
```

The `Array.Copy` method is not just a method that copies elements from one array to another. The `Copy` method handles overlapping regions of memory correctly, as does C's `memmove` function. C's `memcpy` function, on the other hand, doesn't handle overlapping regions correctly. The `Copy` method can also convert each array element as it is copied if conversion is required. The `Copy` method is capable of performing the following conversions:

- Boxing value type elements to reference type elements, such as copying an `Int32[]` to an `Object[]`.
- Unboxing reference type elements to value type elements, such as copying an `Object[]` to an `Int32[]`.

- Widening CLR primitive value types, such as copying elements from an `Int32[]` to a `Double[]`.
- Downcasting elements when copying between array types that can't be proven to be compatible based on the array's type, such as when casting from an `Object[]` to an `IFormattable[]`. If every object in the `Object[]` implements `IFormattable`, `Copy` will succeed.

Here's another example showing the usefulness of `Copy`.

```
// Define a value type that implements an interface.
internal struct MyValueType : IComparable {
   public Int32 CompareTo(Object obj) {
      ...
   }
}


public static class Program {
   public static void Main() {
      // Create an array of 100 value types.
      MyValueType[] src = new MyValueType[100];

      // Create an array of IComparable references.
      IComparable[] dest = new IComparable[src.Length];

      // Initialize an array of IComparable elements to refer to boxed
      // versions of elements in the source array.
      Array.Copy(src, dest, src.Length);
   }
}
```

As you might imagine, the Framework Class Library (FCL) takes advantage of `Array`'s `Copy` method quite frequently.

In some situations, it is useful to cast an array from one type to another. This kind of functionality is called *array covariance*. When you take advantage of array covariance, you should be aware of an associated performance penalty. Let's say you have the following code.

```
String[] sa = new String[100];
Object[] oa = sa;  // oa refers to an array of String elements
oa[5] = "Jeff";    // Perf hit: CLR checks oa's element type for String; OK
oa[3] = 5;         // Perf hit: CLR checks oa's element type for Int32; throws
                   // ArrayTypeMismatchException
```

In the preceding code, the `oa` variable is typed as an `Object[]`; however, it really refers to a `String[]`. The compiler will allow you to write code that attempts to put a 5 into an array element because 5 is an `Int32`, which is derived from `Object`. Of course, the CLR must ensure type safety, and when assigning to an array element, the CLR must ensure that the assignment is legal. So the CLR must check at run time whether the array contains `Int32` elements. In this case, it doesn't, and the assignment cannot be allowed; the CLR will throw an `ArrayTypeMismatchException`.

Note If you just need to make a copy of some array elements to another array, `System.Buffer`'s `BlockCopy` method executes faster than `Array`'s `Copy` method. However, `Buffer`'s `BlockCopy` supports only primitive types; it does not offer the same casting abilities as `Array`'s `Copy` method. The `Int32` parameters are expressed as byte offsets within the array, not as element indexes. `BlockCopy` is really designed for copying data that is bitwise-compatible from one array type to another blittable array type, such as copying a `Byte[]` containing Unicode characters (in the proper byte order) to a `Char[]`. This method allows programmers to partially make up for the lack of the ability to treat an array as a block of memory of any type.

If you need to reliably copy a set of array elements from one array to another array, you should use `System.Array`'s `ConstrainedCopy` method. This method guarantees that the copy operation will either complete or throw an exception without destroying any data within the destination array. This allows `ConstrainedCopy` to be used in a constrained execution region (CER). In order to offer this guarantee, `ConstrainedCopy` requires that the source array's element type be the same as or derived from the destination array's element type. In addition, it will not perform any boxing, unboxing, or downcasting.

All Arrays Are Implicitly Derived from System.Array

When you declare an array variable like this:

```
FileStream[] fsArray;
```

then the CLR automatically creates a `FileStream[]` type for the AppDomain. This type will be implicitly derived from the `System.Array` type, and therefore, all of the instance methods and properties defined on the `System.Array` type will be inherited by the `FileStream[]` type, allowing these methods and properties to be called using the `fsArray` variable. This makes working with arrays extremely convenient because there are many helpful instance methods and properties defined by `System.Array`, such as `Clone`, `CopyTo`, `GetLength`, `GetLongLength`, `GetLowerBound`, `GetUpperBound`, `Length`, `Rank`, and others.

The `System.Array` type also exposes a large number of extremely useful static methods that operate on arrays. These methods all take a reference to an array as a parameter. Some of the useful static methods are `AsReadOnly`, `BinarySearch`, `Clear`, `ConstrainedCopy`, `ConvertAll`, `Copy`, `Exists`, `Find`, `FindAll`, `FindIndex`, `FindLast`, `FindLastIndex`, `ForEach`, `IndexOf`, `LastIndexOf`, `Resize`, `Reverse`, `Sort`, and `TrueForAll`. There are many overloads for each of these methods. In fact, many of the methods provide generic overloads for compile-time type safety as well as good performance. I encourage you to examine the SDK documentation to get an understanding of how useful and powerful these methods are.

All Arrays Implicitly Implement IEnumerable, ICollection, and IList

There are many methods that operate on various collection objects because the methods are declared with parameters such as `IEnumerable`, `ICollection`, and `IList`. It is possible to pass arrays to these methods because `System.Array` also implements these three interfaces. `System.Array` implements these non-generic interfaces because they treat all elements as `System.Object`. However, it would be nice to have `System.Array` implement the generic equivalent of these interfaces, providing better compile-time type safety as well as better performance.

The CLR team didn't want `System.Array` to implement `IEnumerable<T>`, `ICollection<T>`, and `IList<T>`, though, because of issues related to multi-dimensional arrays and non-zero–based arrays. Defining these interfaces on `System.Array` would have enabled these interfaces for all array types. Instead, the CLR performs a little trick: when a single-dimensional, zero–lower bound array type is created, the CLR automatically makes the array type implement `IEnumerable<T>`, `ICollection<T>`, and `IList<T>` (where `T` is the array's element type) and also implements the three interfaces for all of the array type's base types as long as they are reference types. The following hierarchy diagram helps make this clear.

```
Object
   Array (non-generic IEnumerable, ICollection, IList)
      Object[]           (IEnumerable, ICollection, IList of Object)
         String[]        (IEnumerable, ICollection, IList of String)
         Stream[]        (IEnumerable, ICollection, IList of Stream)
            FileStream[] (IEnumerable, ICollection, IList of FileStream)
         .
         .      (other arrays of reference types)
         .
```

So, for example, if you have the following line of code:

```
FileStream[] fsArray;
```

then when the CLR creates the `FileStream[]` type, it will cause this type to automatically implement the `IEnumerable<FileStream>`, `ICollection<FileStream>`, and `IList<FileStream>` interfaces. Furthermore, the `FileStream[]` type will also implement the interfaces for the base types: `IEnumerable<Stream>`, `IEnumerable<Object>`, `ICollection<Stream>`, `ICollection<Object>`, `IList<Stream>`, and `IList<Object>`. Because all of these interfaces are automatically implemented by the CLR, the `fsArray` variable could be used wherever any of these interfaces exist. For example, the `fsArray` variable could be passed to methods that have any of the following prototypes.

```
void M1(IList<FileStream> fsList) { … }
void M2(ICollection<Stream> sCollection) { … }
void M3(IEnumerable<Object> oEnumerable) { … }
```

Note that if the array contains value type elements, the array type will not implement the interfaces for the element's base types. For example, if you have the following line of code:

```
DateTime[] dtArray; // An array of value types
```

then the `DateTime[]` type will implement `IEnumerable<DateTime>`, `ICollection<DateTime>`, and `IList<DateTime>` only; it will not implement versions of these interfaces that are generic over `System.ValueType` or `System.Object`. This means that the `dtArray` variable cannot be passed as an argument to the `M3` method shown earlier. The reason for this is because arrays of value types are laid out in memory differently than arrays of reference types. Array memory layout was discussed earlier in this chapter.

Passing and Returning Arrays

When passing an array as an argument to a method, you are really passing a reference to that array. Therefore, the called method is able to modify the elements in the array. If you don't want to allow this, you must make a copy of the array and pass the copy into the method. Note that the `Array.Copy` method performs a shallow copy, and therefore, if the array's elements are reference types, the new array refers to the already existing objects.

Similarly, some methods return a reference to an array. If the method constructs and initializes the array, returning a reference to the array is fine. But if the method wants to return a reference to an internal array maintained by a field, you must decide if you want the method's caller to have direct access to this array and its elements. If you do, just return the array's reference. But most often, you won't want the method's caller to have such access, so the method should construct a new array and call `Array.Copy`, returning a reference to the new array. Again, be aware that `Array.Copy` makes a shallow copy of the original array.

If you define a method that returns a reference to an array, and if that array has no elements in it, your method can return either `null` or a reference to an array with zero elements in it. When you're implementing this kind of method, Microsoft strongly recommends that you implement the method by having it return a zero-length array because doing so simplifies the code that a developer calling the method must write. For example, this easy-to-understand code runs correctly even if there are no appointments to iterate over.

```
// This code is easier to write and understand.
Appointment[] appointments = GetAppointmentsForToday();
for (Int32 a = 0; a < appointments.Length; a++) {
   ...
}
```

The following code also runs correctly if there are no appointments to iterate over. However, this code is slightly more difficult to write and understand.

```
// This code is harder to write and understand.
Appointment[] appointments = GetAppointmentsForToday();
if (appointments != null) {
```

```
   for (Int32 a = 0, a < appointments.Length; a++) {
      // Do something with appointments[a]
   }
}
```

If you design your methods to return arrays with zero elements instead of `null`, callers of your methods will have an easier time working with them. By the way, you should do the same for fields. If your type has a field that's a reference to an array, you should consider having the field refer to an array even if the array has no elements in it.

Creating Non-Zero Lower Bound Arrays

Earlier I mentioned that it's possible to create and work with arrays that have non-zero lower bounds. You can dynamically create your own arrays by calling `Array`'s static `CreateInstance` method. Several overloads of this method exist, allowing you to specify the type of the elements in the array, the number of dimensions in the array, the lower bounds of each dimension, and the number of elements in each dimension. `CreateInstance` allocates memory for the array, saves the parameter information in the overhead portion of the array's memory block, and returns a reference to the array. If the array has two or more dimensions, you can cast the reference returned from `CreateInstance` to an `ElementType[]` variable (where `ElementType` is some type name), making it easier for you to access the elements in the array. If the array has just one dimension, in C#, you have to use `Array`'s `GetValue` and `SetValue` methods to access the elements of the array.

Here's some code that demonstrates how to dynamically create a two-dimensional array of `System.Decimal` values. The first dimension represents calendar years from 2005 to 2009 inclusive, and the second dimension represents quarters from 1 to 4 inclusive. The code iterates over all the elements in the dynamic array. I could have hard-coded the array's bounds into the code, which would have given better performance, but I decided to use `System.Array`'s `GetLowerBound` and `GetUpperBound` methods to demonstrate their use.

```
using System;

public static class DynamicArrays {
   public static void Main() {
      // I want a two-dimensional array [2005..2009][1..4].
      Int32[] lowerBounds = { 2005, 1 };
      Int32[] lengths     = {    5, 4 };
      Decimal[,] quarterlyRevenue = (Decimal[,])
         Array.CreateInstance(typeof(Decimal), lengths, lowerBounds);

      Console.WriteLine("{0,4}  {1,9}  {2,9}  {3,9}  {4,9}",
         "Year", "Q1", "Q2", "Q3", "Q4");
      Int32 firstYear    = quarterlyRevenue.GetLowerBound(0);
      Int32 lastYear     = quarterlyRevenue.GetUpperBound(0);
      Int32 firstQuarter = quarterlyRevenue.GetLowerBound(1);
      Int32 lastQuarter  = quarterlyRevenue.GetUpperBound(1);
```

```
      for (Int32 year = firstYear; year <= lastYear; year++) {
         Console.Write(year + "  ");
         for (Int32 quarter = firstQuarter; quarter <= lastQuarter; quarter++) {
            Console.Write("{0,9:C}  ", quarterlyRevenue[year, quarter]);
         }
         Console.WriteLine();
      }
   }
}
```

If you compile and run this code, you get the following output.

```
Year         Q1         Q2         Q3         Q4
2005      $0.00      $0.00      $0.00      $0.00
2006      $0.00      $0.00      $0.00      $0.00
2007      $0.00      $0.00      $0.00      $0.00
2008      $0.00      $0.00      $0.00      $0.00
2009      $0.00      $0.00      $0.00      $0.00
```

Array Internals

Internally, the CLR actually supports two different kinds of arrays:

- Single-dimensional arrays with a lower bound of 0. These arrays are sometimes called SZ (for single-dimensional, zero-based) arrays or vectors.
- Single-dimensional and multi-dimensional arrays with an unknown lower bound.

You can actually see the different kinds of arrays by executing the following code (the output is shown in the code's comments).

```
using System;

public sealed class Program {
      public static void Main() {
      Array a;

      // Create a 1-dim, 0-based array, with no elements in it
      a = new String[0];
      Console.WriteLine(a.GetType());   // "System.String[]"

      // Create a 1-dim, 0-based array, with no elements in it
      a = Array.CreateInstance(typeof(String),
         new Int32[] { 0 }, new Int32[] { 0 });
      Console.WriteLine(a.GetType());   // "System.String[]"

      // Create a 1-dim, 1-based array, with no elements in it
      a = Array.CreateInstance(typeof(String),
         new Int32[] { 0 }, new Int32[] { 1 });
      Console.WriteLine(a.GetType());   // "System.String[*]"  <-- INTERESTING!

      Console.WriteLine();
```

```
      // Create a 2-dim, 0-based array, with no elements in it
      a = new String[0, 0];
      Console.WriteLine(a.GetType());   // "System.String[,]"

      // Create a 2-dim, 0-based array, with no elements in it
      a = Array.CreateInstance(typeof(String),
         new Int32[] { 0, 0 }, new Int32[] { 0, 0 });
      Console.WriteLine(a.GetType());   // "System.String[,]"

      // Create a 2-dim, 1-based array, with no elements in it
      a = Array.CreateInstance(typeof(String),
         new Int32[] { 0, 0 }, new Int32[] { 1, 1 });
      Console.WriteLine(a.GetType());   // "System.String[,]"
   }
}
```

Next to each `Console.WriteLine` is a comment that indicates the output. For the single-dimensional arrays, the zero-based arrays display a type name of `System.String[]`, whereas the 1-based array displays a type name of `System.String[*]`. The `*` indicates that the CLR knows that this array is not zero-based. Note that C# does not allow you to declare a variable of type `String[*]`, and therefore it is not possible to use C# syntax to access a single-dimensional, non-zero–based array. Although you can call `Array`'s `GetValue` and `SetValue` methods to access the elements of the array, this access will be slow due to the overhead of the method call.

For multi-dimensional arrays, the zero-based and 1-based arrays all display the same type name: `System.String[,]`. The CLR treats all multi-dimensional arrays as though they are not zero-based at run time. This would make you think that the type name should display as `System.String[*,*]`; however, the CLR doesn't use the *s for multi-dimensional arrays because they would always be present, and the asterisks would just confuse most developers.

Accessing the elements of a single-dimensional, zero-based array is slightly faster than accessing the elements of a non-zero–based, single-dimensional array or a multi-dimensional array. There are several reasons for this. First, there are specific IL instructions—such as `newarr`, `ldelem`, `ldelema`, `ldlen`, and `stelem`—to manipulate single-dimensional, zero-based arrays, and these special IL instructions cause the JIT compiler to emit optimized code. For example, the JIT compiler will emit code that assumes that the array is zero-based, and this means that an offset doesn't have to be subtracted from the specified index when accessing an element. Second, in common situations, the JIT compiler is able to hoist the index range–checking code out of the loop, causing it to execute just once. For example, look at the following commonly written code.

```
using System;

public static class Program {
   public static void Main() {
      Int32[] a = new Int32[5];
      for(Int32 index = 0; index < a.Length; index++) {
         // Do something with a[index]
      }
   }
}
```

The first thing to notice about this code is the call to the array's `Length` property in the `for` loop's test expression. Because `Length` is a property, querying the length actually represents a method call. However, the JIT compiler knows that `Length` is a property on the `Array` class, and the JIT compiler will actually generate code that calls the property just once and stores the result in a temporary variable that will be checked with each iteration of the loop. The result is that the JITted code is fast. In fact, some developers have underestimated the abilities of the JIT compiler and have tried to write "clever code" in an attempt to help the JIT compiler. However, any clever attempts that you come up with will almost certainly impact performance negatively and make your code harder to read, reducing its maintainability. You are better off leaving the call to the array's `Length` property in the preceding code instead of attempting to cache it in a local variable yourself.

The second thing to notice about the preceding code is that the JIT compiler knows that the `for` loop is accessing array elements 0 through `Length - 1`. So the JIT compiler produces code that, at run time, tests that all array accesses will be within the array's valid range. Specifically, the JIT compiler produces code to check if `(0 >= a.GetLowerBound(0)) && ((Length - 1) <= a.GetUpperBound(0))`. This check occurs just before the loop. If the check is good, the JIT compiler will not generate code inside the loop to verify that each array access is within the valid range. This allows array access within the loop to be very fast.

Unfortunately, as I alluded to earlier in this chapter, accessing elements of a non-zero–based single-dimensional array or of a multi-dimensional array is much slower than a single-dimensional, zero-based array. For these array types, the JIT compiler doesn't hoist index checking outside of loops, so each array access validates the specified indexes. In addition, the JIT compiler adds code to subtract the array's lower bounds from the specified index, which also slows the code down, even if you're using a multi-dimensional array that happens to be zero-based. So if performance is a concern to you, you might want to consider using an array of arrays (a jagged array) instead of a rectangular array.

C# and the CLR also allow you to access an array by using unsafe (non-verifiable) code, which is, in effect, a technique that allows you to turn off the index bounds checking when accessing an array. Note that this unsafe array manipulation technique is usable with arrays whose elements are `SByte`, `Byte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Char`, `Single`, `Double`, `Decimal`, `Boolean`, an enumerated type, or a value type structure whose fields are any of the aforementioned types.

This is a very powerful feature that should be used with extreme caution because it allows you to perform direct memory accesses. If these memory accesses are outside the bounds of the array, an exception will not be thrown; instead, you will be corrupting memory, violating type safety, and possibly opening a security hole! For this reason, the assembly containing the unsafe code must either be granted full trust or at least have the Security Permission with Skip Verification turned on.

The following C# code demonstrates three techniques (safe, jagged, and unsafe), for accessing a two-dimensional array.

```
using System;
using System.Diagnostics;

public static class Program {
   private const Int32 c_numElements = 10000;
```

```
   public static void Main() {
      // Declare a two-dimensional array
      Int32[,] a2Dim = new Int32[c_numElements, c_numElements];

      // Declare a two-dimensional array as a jagged array (a vector of vectors)
      Int32[][] aJagged = new Int32[c_numElements][];
      for (Int32 x = 0; x < c_numElements; x++)
         aJagged[x] = new Int32[c_numElements];

      // 1: Access all elements of the array using the usual, safe technique
      Safe2DimArrayAccess(a2Dim);

      // 2: Access all elements of the array using the jagged array technique
      SafeJaggedArrayAccess(aJagged);

      // 3: Access all elements of the array using the unsafe technique
      Unsafe2DimArrayAccess(a2Dim);
   }

   private static Int32 Safe2DimArrayAccess(Int32[,] a) {
      Int32 sum = 0;
      for (Int32 x = 0; x < c_numElements; x++) {
         for (Int32 y = 0; y < c_numElements; y++) {
            sum += a[x, y];
         }
      }
      return sum;
   }

   private static Int32 SafeJaggedArrayAccess(Int32[][] a) {
      Int32 sum = 0;
      for (Int32 x = 0; x < c_numElements; x++) {
         for (Int32 y = 0; y < c_numElements; y++) {
            sum += a[x][y];
         }
      }
      return sum;
   }

   private static unsafe Int32 Unsafe2DimArrayAccess(Int32[,] a) {
      Int32 sum = 0;
      fixed (Int32* pi = a) {
         for (Int32 x = 0; x < c_numElements; x++) {
            Int32 baseOfDim = x * c_numElements;
            for (Int32 y = 0; y < c_numElements; y++) {
               sum += pi[baseOfDim + y];
            }
         }
      }
      return sum;
   }
}
```

The `Unsafe2DimArrayAccess` method is marked with the `unsafe` modifier, which is required to use C#'s `fixed` statement. To compile this code, you'll have to specify the `/unsafe` switch when invoking the C# compiler or select the Allow Unsafe Code check box on the Build tab of the Project Properties pane in Microsoft Visual Studio.

Obviously, the unsafe technique has a time and place when it can best be used by your own code, but beware that there are three serious downsides to using this technique:

- The code that manipulates the array elements is more complicated to read and write than that which manipulates the elements using the other techniques because you are using C#'s `fixed` statement and performing memory-address calculations.
- If you make a mistake in the calculation, you are accessing memory that is not part of the array. This can result in an incorrect calculation, corruption of memory, a type-safety violation, and a potential security hole.
- Due to the potential problems, the CLR forbids unsafe code from running in reduced-security environments (like Microsoft Silverlight).

Unsafe Array Access and Fixed-Size Array

Unsafe array access is very powerful because it allows you to access:

- Elements within a managed array object that resides on the heap (as the previous section demonstrated).
- Elements within an array that resides on an unmanaged heap. The `SecureString` example in Chapter 14, "Chars, Strings, and Working with Text," demonstrated using unsafe array access on an array returned from calling the `System.Runtime.InteropServices.Marshal` class's `SecureStringToCoTaskMemUnicode` method.
- Elements within an array that resides on the thread's stack.

In cases in which performance is extremely critical, you could avoid allocating a managed array object on the heap and instead allocate the array on the thread's stack by using C#'s `stackalloc` statement (which works a lot like C's `alloca` function). The `stackalloc` statement can be used to create a single-dimensional, zero-based array of value type elements only, and the value type must not contain any reference type fields. Really, you should think of this as allocating a block of memory that you can manipulate by using unsafe pointers, and therefore, you cannot pass the address of this memory buffer to the vast majority of FCL methods. Of course, the stack-allocated memory (array) will automatically be freed when the method returns; this is where we get the performance improvement. Using this feature also requires you to specify the `/unsafe` switch to the C# compiler.

The `StackallocDemo` method in the following code shows an example of how to use C#'s `stackalloc` statement.

```
using System;

public static class Program {
   public static void Main() {
      StackallocDemo();
      InlineArrayDemo();
   }

   private static void StackallocDemo() {
      unsafe {
         const Int32 width = 20;
         Char* pc = stackalloc Char[width]; // Allocates array on stack

         String s = "Jeffrey Richter";      // 15 characters

         for (Int32 index = 0; index < width; index++) {
            pc[width - index - 1] =
               (index < s.Length) ? s[index] : '.';
         }

         // The following line displays ".....rethciR yerffeJ"
         Console.WriteLine(new String(pc, 0, width));
      }
   }

   private static void InlineArrayDemo() {
      unsafe {
         CharArray ca;                 // Allocates array on stack
         Int32 widthInBytes = sizeof(CharArray);
         Int32 width = widthInBytes / 2;

         String s = "Jeffrey Richter"; // 15 characters

         for (Int32 index = 0; index < width; index++) {
            ca.Characters[width - index - 1] =
               (index < s.Length) ? s[index] : '.';
         }

         // The following line displays ".....rethciR yerffeJ"
         Console.WriteLine(new String(ca.Characters, 0, width));
      }
   }
}

internal unsafe struct CharArray {
   // This array is embedded inline inside the structure
   public fixed Char Characters[20];
}
```

Normally, because arrays are reference types, an array field defined in a structure is really just a pointer or reference to an array; the array itself lives outside of the structure's memory. However, it is possible to embed an array directly inside a structure as shown by the `CharArray` structure in the preceding code. To embed an array directly inside a structure, there are several requirements:

- The type must be a structure (value type); you cannot embed an array inside a class (reference type).
- The field or its defining structure must be marked with the `unsafe` keyword.
- The array field must be marked with the `fixed` keyword.
- The array must be single-dimensional and zero-based.
- The array's element type must be one of the following types: `Boolean`, `Char`, `SByte`, `Byte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, or `Double`.

Inline arrays are typically used for scenarios that involve interoperating with unmanaged code where the unmanaged data structure also has an inline array. However, inline arrays can be used in other scenarios as well. The `InlineArrayDemo` method in the code shown earlier offers an example of how to use an inline array. The `InlineArrayDemo` method performs the same function as the `StackallocDemo` method; it just does it in a different way.