



Arrays

Arrays

Types of Arrays

An Array As an Object

One-Dimensional and Rectangular Arrays

Instantiating a One-Dimensional or Rectangular Array

Accessing Array Elements

Initializing an Array

Jagged Arrays

Comparing Rectangular and Jagged Arrays

The foreach Statement

Array Covariance

Useful Inherited Array Members

Comparing Array Types

Arrays

An array is a set of uniform data elements, represented by a single variable name. The individual elements are accessed using the variable name and one or more indexes between square brackets, as shown here:

Array name Index

↓ ↓

MyArray[4]

Definitions

Let's start with some important definitions having to do with arrays in C#.

Elements: The individual data items of an array are called *elements*. All elements of an array must be of the same type.

Rank/dimensions: Arrays can have any positive number of *dimensions*. The number of dimensions an array has is called its *rank*.

Dimension length: Each dimension of an array has a *length*, which is the number of positions in that direction.

Array length: The total number of elements contained in an array, in *all* dimensions, is called the *length* of the array.

Important Details

Besides the preceding definitions, there are several high-level facts about C# arrays that I should mention before launching into a discussion of arrays.

- Once an array is created, its size is fixed. C# does not support dynamic arrays.
- Array indexes are *0-based*. That is, if the length of a dimension is *n*, the index values range from 0 to *n* - 1. For example, Figure 14-1 shows the dimensions and lengths of two example arrays. Notice that for each dimension, the indexes range from 0 to *length* - 1.

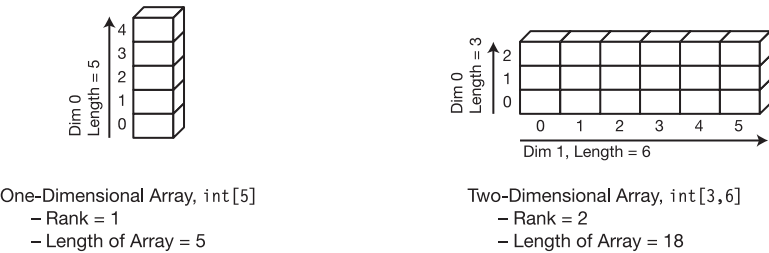


Figure 14-1. Dimensions and sizes

Types of Arrays

C# provides two types of arrays:

- One-dimensional arrays can be thought of as a single line, or *vector*, of elements.
- Multidimensional arrays are composed such that each position in the primary vector is itself an array, called a *sub-array*. Positions in the sub-array vectors can themselves be sub-arrays.

In addition, there are two types of multidimensional arrays, rectangular arrays and jagged arrays, which have the following characteristics:

- Rectangular arrays
 - Are multidimensional arrays where all the sub-arrays in a particular dimension have the same length
 - Always use a single set of square brackets, regardless of the number of dimensions

```
int x = myArray2[4, 6, 1]           // One set of square brackets
```

- Jagged arrays
 - Are multidimensional arrays where each sub-array is an independent array
 - Can have sub-arrays of *different* lengths
 - Use a separate set of square brackets for each dimension of the array

```
jagArray1[2][7][4]                 // Three sets of square brackets
```

Figure 14-2 shows the kinds of arrays available in C#.

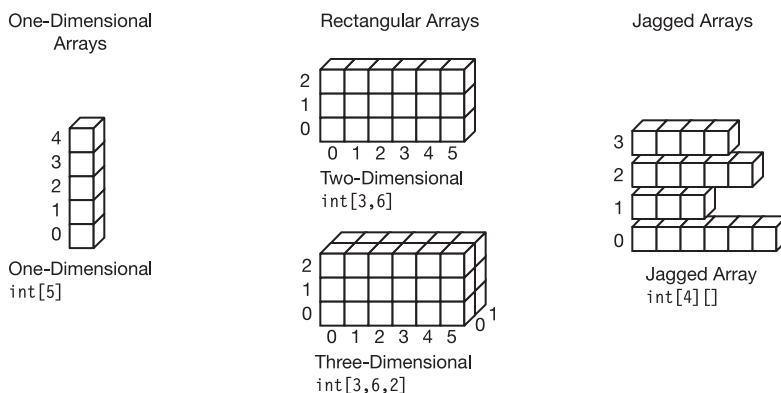


Figure 14-2. One-dimensional, rectangular, and jagged arrays

An Array As an Object

An array instance is an object whose type derives from class `System.Array`. Since arrays are derived from this BCL base class, they inherit a number of useful members from it, such as

- Rank: A property that returns the number of dimensions of the array
- Length: A property that returns the length of the array (the total number of elements in the array)

Arrays are reference types, and as with all reference types, have both a reference to the data and the data object itself. The reference is in either the stack or the heap, and the data object itself will always be in the heap. Figure 14-3 shows the memory configuration and components of an array.

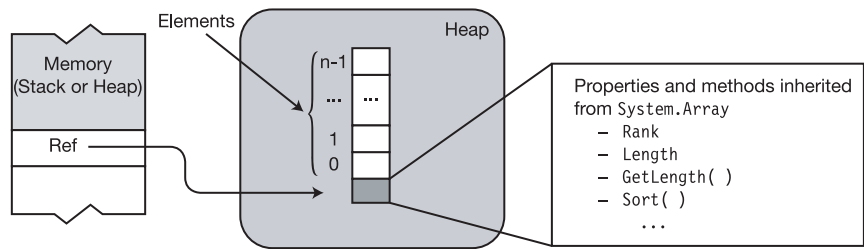


Figure 14-3. Structure of an array

Although an array is always a reference type, the elements of the array can be either value types or reference types.

- An array is called a *value type array* if the elements stored are value types.
- An array is called a *reference type array* if the elements stored in the array are *references* of reference type objects.

Figure 14-4 shows a value type array and a reference type array.

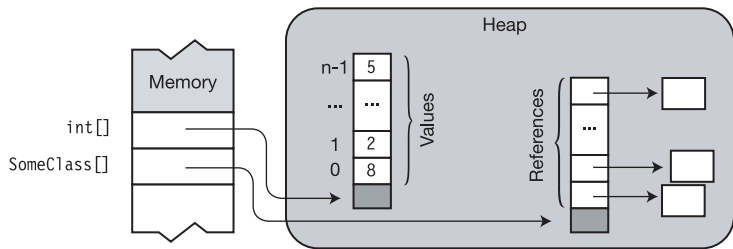


Figure 14-4. Elements can be values or references.

One-Dimensional and Rectangular Arrays

Syntactically, one-dimensional arrays and rectangular arrays are very similar, so they will be treated together. Jagged arrays will be treated separately.

Declaring a One-Dimensional Array or a Rectangular Array

To declare a one-dimensional or rectangular array, use a single set of square brackets between the type and the variable name.

The *rank specifiers* are commas between the brackets. They specify the number of dimensions the array will have. The rank is the number of commas, plus one. For example, no commas indicates a one-dimensional array, one comma indicates a two-dimensional array, and so forth.

The base type, together with the rank specifiers, is the *type* of the array. For example, the following line of code declares a one-dimensional array of longs. The type of the array is `long[]`, which is read as “an array of longs.”

```
Rank specifiers = 1
  ↓
long[ ] secondArray;
  ↑
Array type
```

The following code shows examples of declarations of rectangular arrays. Notice that

- You can have as many rank specifiers as you need.
- You cannot place array dimension lengths in the array type section. The rank is part of the array’s type, but the lengths of the dimensions are *not* part of the type.
- When an array is declared, the *number* of dimensions is fixed. The *length* of the dimensions, however, is not determined until the array is instantiated.

```
Rank specifiers
  ↓
int[, ] firstArray;           // Array type: 3-D array of int
int[, ] arr1;                 // Array type: 2-D array of int
long[, ] arr3;                // Array type: 3-D array of long
  ↑
Array type

long[3,2,6] SecondArray;      // Wrong! Compile error
  ↑
Dimension lengths not allowed!
```

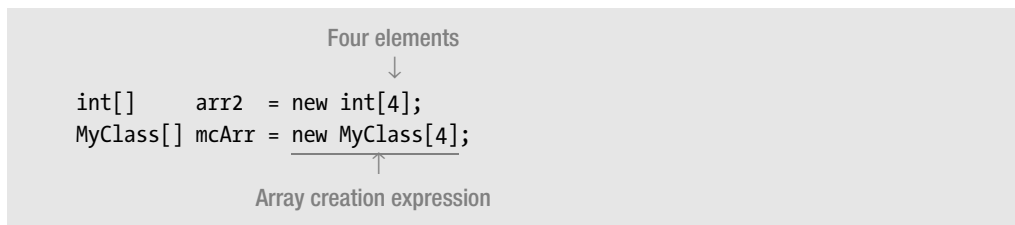
Note Unlike C/C++, the brackets follow the base type, not the variable name.

Instantiating a One-Dimensional or Rectangular Array

To instantiate an array, you use an *array creation expression*. An array creation expression consists of the new operator, followed by the base type, followed by a pair of square brackets. The length of each dimension is placed in a comma-separated list between the brackets.

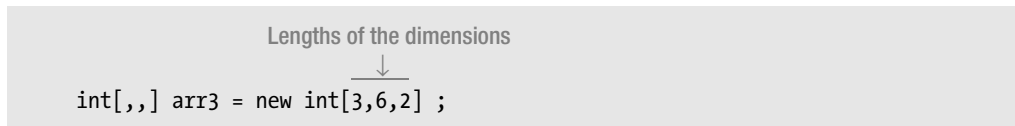
The following are examples of one-dimensional array declarations:

- Array `arr2` is a one-dimensional array of four ints.
- Array `mcArr` is a one-dimensional array of four `MyClass` references.
- Their layouts in memory are shown in Figure 14-5.



The following is an example of a rectangular array. Array `arr3` is a three-dimensional array.

- The length of the array is $3 * 6 * 2 = 36$.
- Its layout in memory is shown in Figure 14-5.



At the time of instantiation, each element is automatically initialized to the default initialization value for the type of the element.

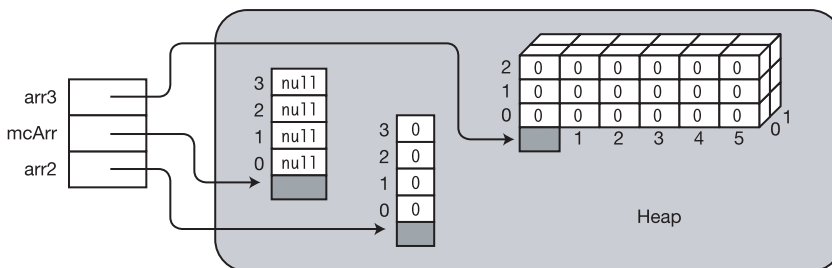


Figure 14-5. Declaring and instantiating arrays

Note Array creation expressions do not contain parentheses—even for reference type arrays.

Accessing Array Elements

An array element is accessed using an integer value as an index into the array.

- Each dimension uses 0-based indexing.
- The index is placed between square brackets following the array name.

The following code shows examples of declaring, writing to, and reading from a one-dimensional and a two-dimensional array:

```
int[] intArr1 = new int[15];           // 1-D example
intArr1[2]    = 10;                    // Write to element 2 of the array.
int var1      = intArr1[2];            // Read from element 2 of the array.

int[,] intArr2 = new int[5,10];        // 2-D example
intArr2[2,3]   = 7;                    // Write to the array.
int var2       = intArr2[2,3];         // Read from the array.
```

The following code shows the full process of creating and accessing a one-dimensional array:

```
int[] myIntArray;                      // Declare the array.

myIntArray = new int[4];                // Instantiate the array.

for( int i=0; i<4; i++ )                // Set the values.
    myIntArray[i] = i*10;

// Read and display the values of each element.
for( int i=0; i<4; i++ )
    Console.WriteLine("Value of element {0} = {1}", i, myIntArray[i]);
```

This code produces the following output:

```
Value of element 0 is 0
Value of element 1 is 10
Value of element 2 is 20
Value of element 3 is 30
```

Initializing an Array

Array elements are always initialized. If they are not explicitly initialized, the system will automatically initialize them to default values.

Automatic Initialization

When any type of array is created, each of the elements is automatically initialized to the default value for the type. The default values for the predefined types are 0 for integer types, 0.0 for floating point types, false for Booleans, and null for reference types.

For example, the following code creates an array and initializes its four elements to the value 0. Figure 14-6 illustrates the layout in memory.

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

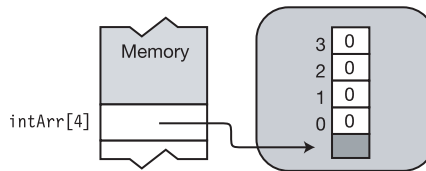


Figure 14-6. Automatic initialization of a one-dimensional array

Explicit Initialization of One-Dimensional Arrays

For a one-dimensional array, you can set explicit initial values by including an *initialization list* immediately after the array creation expression of an array instantiation.

- The initialization values must be separated by commas and enclosed in a set of curly braces.
- Notice, however, that nothing separates the array creation expression and the initialization list. That is, there is no equals sign or other connecting operator.

For example, the following code creates an array and initializes its four elements to the values between the curly braces. Figure 14-7 illustrates the layout in memory.

```
int[] intArr = new int[4] { 10, 20, 30, 40 };
```

Initialization list
↓
↑
No connecting operator

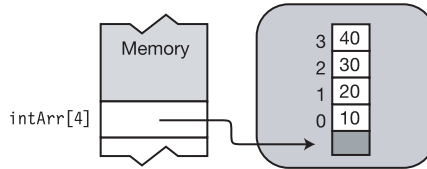


Figure 14-7. *Explicit initialization of a one-dimensional array*

Explicit Initialization of Rectangular Arrays

To explicitly initialize a rectangular array:

- Each *vector of initial values* must be enclosed in curly braces.
- Each *dimension* must also be nested and enclosed in curly braces.
- In addition to the initial values, the initialization lists and components of each dimension must also be separated by commas.

For example, the following code shows the declaration of a two-dimensional array with an initialization list. Figure 14-8 illustrates the layout in memory.

Initialization lists separated by commas
↓ ↓

```
int[,] intArray2 = new int[3,2] { {10, 1}, {2, 10}, {11, 9} } ;
```

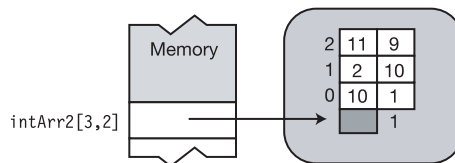


Figure 14-8. *Initializing a rectangular array*

Syntax Points for Initializing Rectangular Arrays

Rectangular arrays are initialized with nested, comma-separated initialization lists. The initialization lists are nested in curly braces. This can sometimes be confusing, so to get the nesting, grouping, and commas right, the following tips can be helpful:

- Commas are used as *separators* between all *elements* and *groups*.
- Commas are *not* placed between left curly braces.
- Commas are *not* placed before a right curly brace.
- Read the rank specifications from left to right, designating the last number as “elements” and all the others as “groups.”

For example, read the following declaration as “intArray has four groups of three groups of two elements.”

```

                                Initialization lists, nested and separated by commas
int[,] intArray = new int[4,3,2] {
                                ↓           ↓           ↓
                                { {8, 6}, {5, 2}, {12, 9} },
                                { {6, 4}, {13, 9}, {18, 4} },
                                { {7, 2}, {1, 13}, {9, 3} },
                                { {4, 6}, {3, 2}, {23, 8} }
                                };

```

Shortcut Syntax

When combining declaration, array creation, and initialization in a single statement, the array creation expression part of the syntax can be left out. This shortcut syntax is shown in Figure 14-9.

```

int[] arr1 = new int[3] {10, 20, 30};
int[] arr1 = {10, 20, 30}; } Equivalent

int[,] arr = new int[2,3] {{0, 1, 2}, {10, 11, 12}};
int[,] arr = {{0, 1, 2}, {10, 11, 12}}; } Equivalent

```

Figure 14-9. *Shortcut for array declaration, creation, and initialization*

Putting It All Together

Now that you have all the pieces, they can be put together in a full example. The following code creates, initializes, and uses a rectangular array.

```

// Declare, create, and initialize the array.
int[,] arr = new int[2,3] {{0, 1, 2}, {10, 11, 12}};

// Print the values.
for( int i=0; i<2; i++ )
    for( int j=0; j<3; j++ )
        Console.WriteLine("Element [{0},{1}] is {2}", i, j, arr[i,j]);

```

This code produces the following output:

```

Element [0,0] is 0
Element [0,1] is 1
Element [0,2] is 2
Element [1,0] is 10
Element [1,1] is 11
Element [1,2] is 12

```

Jagged Arrays

A jagged array is an array of arrays. Unlike rectangular arrays, the sub-arrays of a jagged array can have different numbers of elements.

For example, the following code declares a two-dimensional jagged array. The array's layout in memory is shown in Figure 14-10.

- The length of the first dimension is 3.
- The declaration can be read as “jagArr is an array of three arrays of ints.”
- Notice that the figure shows *four* array objects—one for the top-level array, and three for the sub-arrays.

```
int[][] jagArr = new int[3][]; // Declare and create top-level array.
...                          // Declare and create sub-arrays.
```

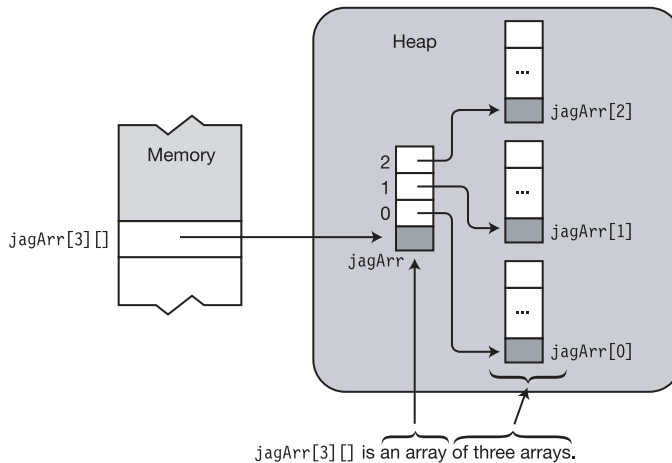


Figure 14-10. A jagged array is an array of arrays.

Declaring a Jagged Array

The declaration syntax for jagged arrays requires a separate set of square brackets for each dimension. The number of sets of square brackets in the declaration of the array variable determines the rank of the array.

- A jagged array can be of any number of dimensions greater than one.
- As with rectangular arrays, dimension lengths cannot be included in the array type section of the declaration.

Rank specifiers
↓
`int[][] SomeArr; // Rank = 2`
`int[][][] OtherArr; // Rank = 3`
↑ ↑
Array type Array name

Shortcut Instantiation

You can combine the jagged array declaration with the creation of the first-level array using an array creation expression, such as in the following declaration. The result is shown in Figure 14-11.

Three sub-arrays
↓
`int[][] jagArr = new int[3][];`

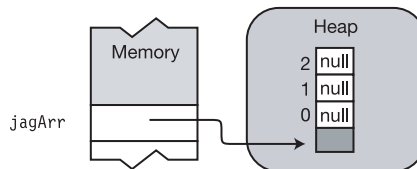


Figure 14-11. *Shortcut first-level instantiation*

You cannot instantiate more than the first-level array in the declaration statement.

Allowed
↓
`int[][] jagArr = new int[3][4];` // Wrong! Compile error
↑
Not allowed

Instantiating a Jagged Array

Unlike other types of arrays, you cannot fully instantiate a jagged array in a single step. Since a jagged array is an array of independent arrays—each array must be created separately. Instantiating a full jagged array requires the following steps:

1. First, instantiate the top-level array.
2. Next, instantiate each sub-array separately, assigning the reference to the newly created array to the appropriate element of its containing array.

For example, the following code shows the declaration, instantiation, and initialization of a two-dimensional jagged array. Notice in the code that the reference to each sub-array is assigned to an element in the top-level array. The progression of steps 1 through 4 in the code correspond to the numbered representations in Figure 14-12.

```
int[][] Arr = new int[3][];           // 1. Instantiate top level

Arr[0] = new int[] {10, 20, 30};      // 2. Instantiate sub-array
Arr[1] = new int[] {40, 50, 60, 70};  // 3. Instantiate sub-array
Arr[2] = new int[] {80, 90, 100, 110, 120}; // 4. Instantiate sub-array
```

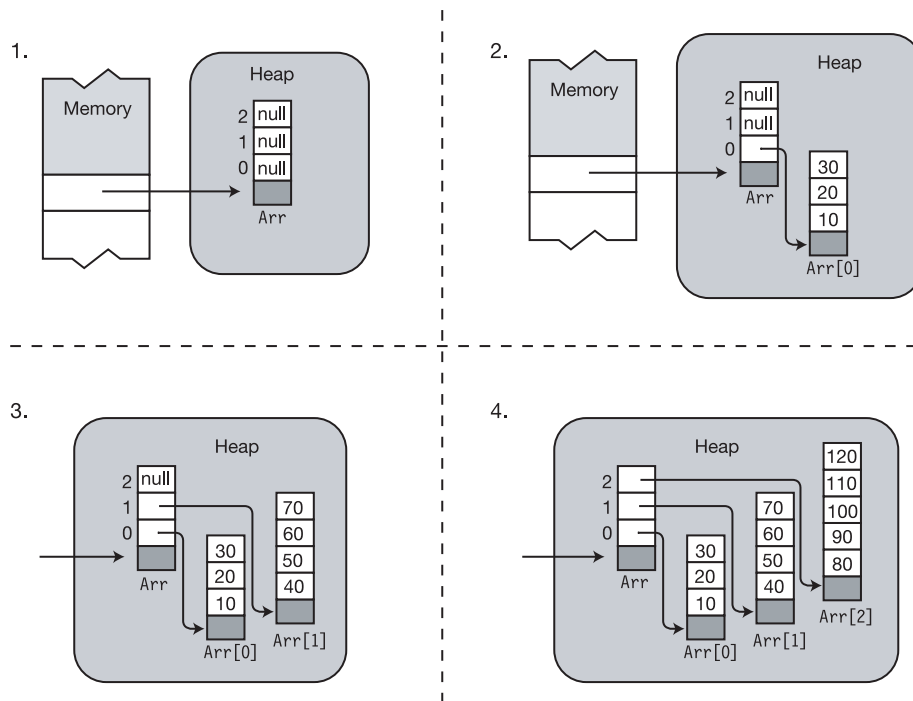


Figure 14-12. Creating a two-dimensional jagged array

Sub-Arrays in Jagged Arrays

Since the sub-arrays in a jagged array are themselves arrays, it is possible to have rectangular arrays inside jagged arrays. For example, the following code creates a jagged array of three two-dimensional rectangular arrays and initializes them with values. It then displays the values.

- The structure is illustrated in Figure 14-13.
- The code also uses the `GetLength(int n)` method of arrays, inherited from `System.Array`, to get the length of the specified dimension of the array.

```
int[,] Arr;           // An array of 2-D arrays
Arr = new int[3][,];  // Instantiate an array of three 2-D arrays.

Arr[0] = new int[,] { { 10, 20 },      { 100, 200 }      };
Arr[1] = new int[,] { { 30, 40, 50 },   { 300, 400, 500 }   };
Arr[2] = new int[,] { { 60, 70, 80, 90 }, { 600, 700, 800, 900 } };

    Get length of dimension 0 of Arr ↓
for (int i = 0; i < Arr.GetLength(0); i++)
{
    Get length of dimension 0 of Arr[i] ↓
    for (int j = 0; j < Arr[i].GetLength(0); j++)
    {
        Get length of dimension 1 of Arr[i] ↓
        for (int k = 0; k < Arr[i].GetLength(1); k++) {
            Console.WriteLine
                ("[{0}][{1},{2}] = {3}", i, j, k, Arr[i][j, k]);
        }
        Console.WriteLine("");
    }
    Console.WriteLine("");
}
```

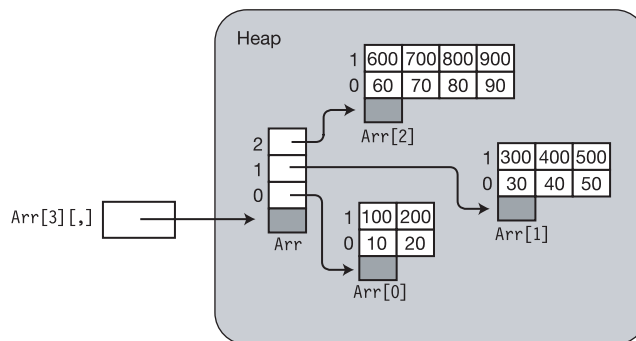


Figure 14-13. Jagged array of three two-dimensional arrays

Comparing Rectangular and Jagged Arrays

The structure of rectangular and jagged arrays is significantly different. For example, Figure 14-14 shows the structure of a rectangular three-by-three array, and a jagged array of three one-dimensional arrays of length 3.

- Both arrays hold nine integers, but as you can see, their structures are quite different.
- The rectangular array has a single array object, while the jagged array has four array objects.

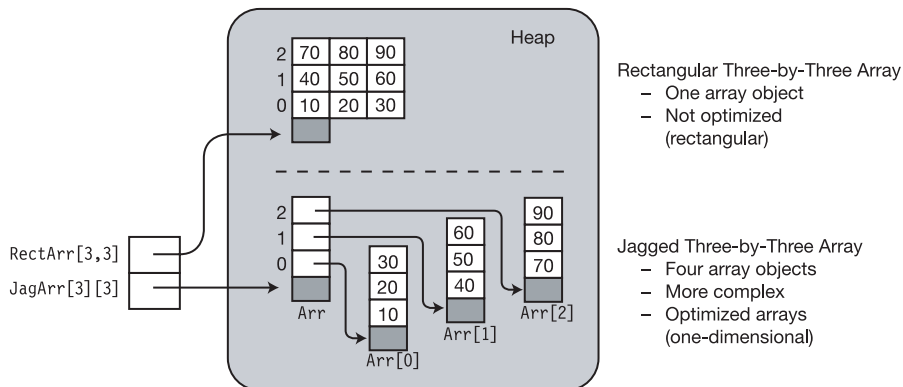


Figure 14-14. Comparing the structure of rectangular and jagged arrays

One-dimensional arrays have specific instructions in the CIL that allow them to be optimized for performance. Rectangular arrays do not have these instructions, and are not optimized to the same level. Because of this, it can sometimes be more efficient to use jagged arrays of one-dimensional arrays—which can be optimized—than rectangular arrays, which cannot.

On the other hand, the programming complexity can be less for a rectangular array because it can be treated as a single unit, rather than an array of arrays.

The foreach Statement

The foreach statement allows you to sequentially access each element in an array. It is actually a more general construct in that it also works with other collection types—but this section will only discuss its use with arrays. Chapter 20 will cover its use with other collection types.

The important points of the foreach statement are the following:

- The *iteration variable* is a temporary, read-only variable of the same type as the elements of the array. The foreach statement uses the iteration variable to sequentially represent each element in the array.
- The syntax of the foreach statement is where
 - *Type* is the type of the elements of the array.
 - *Identifier* is the name of the *iteration variable*.
 - *ArrayName* is the name of the array to be processed.
 - *Statement* is a simple statement or a block that is executed once for each element in the array.

```

      Iteration variable declaration
          ↓
foreach( Type Identifier in ArrayName )
      Statement
  
```

The way the foreach statement works is the following:

- It starts with the first element of the array and assigns that value to the *iteration variable*.
- It then executes the body of the statement. Inside the body, you can use the iteration variable as a read-only alias for the array element.
- After the body is executed, the foreach statement selects the next element in the array and repeats the process.

In this way, it cycles through the array, allowing you to access each element one by one. For example, the following code shows the use of a foreach statement with a one-dimensional array of four integers:

- The `WriteLine` statement, which is the body of the foreach statement, is executed once for each of the elements of the array.
- The first time through the loop, iteration variable `item` has the value of the first element of the array. Each successive time, it will have the value of the next element in the array.

```

int[] arr1 = {10, 11, 12, 13};
      Iteration variable declaration
          ↓
foreach( int item in arr1 )           Iteration variable use
      Console.WriteLine("Item Value: {0}", item);
  
```


The Iteration Variable Is Read-Only

Since the value of the iteration variable is read-only, clearly, it cannot be changed. But this has different effects on value type arrays and reference type arrays.

For value type arrays, this means that you cannot change the data of the array. For example, in the following code, the attempt to change the data in the iteration variable produces a compile-time error message:

```
int[] arr1 = {10, 11, 12, 13};

foreach( int item in arr1 )
    item++;           // Wrong. Changing variable value is not allowed
```

For reference type arrays, you still cannot change the iteration variable, but the iteration variable only holds the reference to the data, not the data itself. You *can*, therefore, change the data through the iteration variable.

The following code creates an array of four `MyClass` objects and initializes them. In the first `foreach` statement, the data in each of the objects is changed. In the second `foreach` statement, the changed data is read from the objects.

```
class MyClass
{
    public int MyField = 0;
}

class Program {
    static void Main() {
        MyClass[] mcArray = new MyClass[4];           // Create array
        for (int i = 0; i < 4; i++)
        {
            mcArray[i] = new MyClass();               // Create class objects
            mcArray[i].MyField = i;                   // Set field
        }
        foreach (MyClass item in mcArray)
            item.MyField += 10;                        // Change the data.

        foreach (MyClass item in mcArray)
            Console.WriteLine("{0}", item.MyField);   // Read the changed data.
    }
}
```

This code produces the following output:

```
10
11
12
13
```

The foreach Statement with Multidimensional Arrays

In a multidimensional array, the elements are processed in the order in which the rightmost index is incremented fastest. When the index has gone from 0 to *length* - 1, the next index to the left is incremented, and the indexes to the right are reset to 0.

Example with a Rectangular Array

The following example shows the foreach statement used with a rectangular array:

```
class Sample
{
    static void Main()
    {
        int nTotal = 0;
        int[,] arr1 = { {10, 11}, {12, 13} };

        foreach( int element in arr1 )
        {
            nTotal += element;
            Console.WriteLine
                ("Element: {0}, Current Total: {1}", element, nTotal);
        }
    }
}
```

The output is the following:

```
Element: 10, Current Total: 10
Element: 11, Current Total: 21
Element: 12, Current Total: 33
Element: 13, Current Total: 46
```

Example with a Jagged Array

Since jagged arrays are arrays of arrays, separate `foreach` statements must be used for each dimension in the jagged array. The `foreach` statements must be nested properly to make sure that each nested array is processed properly.

For example, in the following code, the first `foreach` statement cycles through the top-level array—`arr1`—selecting the next sub-array to process. The inner `foreach` statement processes the elements of that sub-array.

```
static void Main()
{
    int nTotal = 0;
    int[][] arr1 = new int[2][];
    arr1[0] = new int[] { 10, 11 };
    arr1[1] = new int[] { 12, 13, 14 };

    foreach (int[] array in arr1)           // Process the top level.
    {
        Console.WriteLine("Starting new array");
        foreach (int item in array)         // Process the second level.
        {
            nTotal += item;
            Console.WriteLine("  Item: {0}, Current Total: {1}", item, nTotal);
        }
    }
}
```

The output is the following:

```
Starting new array
  Item: 10, Current Total: 10
  Item: 11, Current Total: 21
Starting new array
  Item: 12, Current Total: 33
  Item: 13, Current Total: 46
  Item: 14, Current Total: 60
```

Array Covariance

Under certain conditions, you can assign an object to an array element even if the object is not of the array's base type. This property is called *covariance*. You can use covariance if

- The array is a reference type array.
- There is an implicit or explicit conversion between the type of the object you are assigning and the array's base type.

Since there is always an implicit conversion between a derived class and its base class, you can always assign an object of a derived class to an array declared for the base class.

For example, the following code declares two classes, A and B, where class B derives from class A. The last line shows covariance by assigning objects of type B to array elements of type A. The memory layout for the code is shown in Figure 14-15.

```
class A { ... } // Base class
class B : A { ... } // Derived class

class Program {
    static void Main() {
        // Two arrays of type A[]
        A[] AArray1 = new A[3];
        A[] AArray2 = new A[3];

        // Normal--assigning objects of type A to an array of type A
        AArray1[0] = new A(); AArray1[1] = new A(); AArray1[2] = new A();

        // Covariant--assigning objects of type B to an array of type A
        AArray2[0] = new B(); AArray2[1] = new B(); AArray2[2] = new B();
    }
}
```

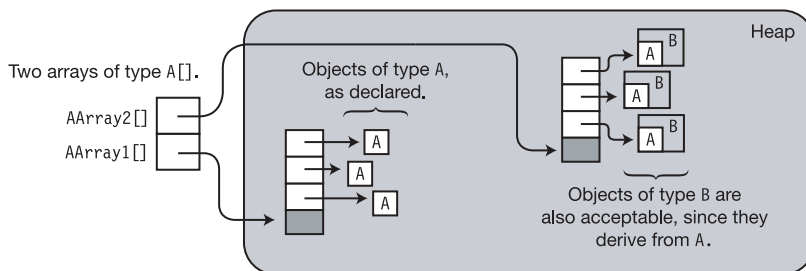


Figure 14-15. Arrays showing covariance

Note There is no covariance for value type arrays.

Useful Inherited Array Members

I mentioned earlier that C# arrays are derived from class `System.Array`. From that base class, they inherit a number of useful properties and methods. Some of the most useful ones are listed in Table 14-1.

Table 14-1. *Some Useful Members Inherited by Arrays*

Member	Type	Lifetime	Meaning
Rank	Property	Instance	Gets the number of dimensions of the array
Length	Property	Instance	Gets the total number of elements in all the dimensions of the array
GetLength	Method	Instance	Returns the length of a particular dimension of the array
Clear	Method	Static	Sets a range of elements to 0 or null
Sort	Method	Static	Sorts the elements in a one-dimensional array
BinarySearch	Method	Static	Searches a one-dimensional array for a value, using binary search
Clone	Method	Instance	Performs a shallow copy of the array—copying only the elements, both for arrays of value types and reference types
IndexOf	Method	Static	Returns the index of the first occurrence of a value in a one-dimensional array
Reverse	Method	Static	Reverses the order of the elements of a range of a one-dimensional array
GetUpperBound	Method	Instance	Gets the upper bound at the specified dimension

For example, the following code uses some of these properties and methods:

```
public static void PrintArray(int[] a)
{
    foreach (int x in a)
        Console.Write("{0} ", x);
    Console.WriteLine("");
}

static void Main()
{
    int[] arr = new int[] { 15, 20, 5, 25, 10 }; PrintArray(arr);
    Array.Sort(arr);                           PrintArray(arr);
    Array.Reverse(arr);                        PrintArray(arr);

    Console.WriteLine("Rank = {0}, Length = {1}",arr.Rank, arr.Length);
    Console.WriteLine("GetLength(0) = {0}",arr.GetLength(0));
    Console.WriteLine("GetType()    = {0}",arr.GetType());
}
```

The Clone Method

The Clone method performs a shallow copy of an array. This means that it only creates a clone of the array itself. If it is a reference type array, it does *not* copy the objects referenced by the elements. This has different results for value type arrays and reference type arrays.

- Cloning a value type array results in two independent arrays.
- Cloning a reference type array results in two arrays pointing at the same objects.

The Clone method returns a reference of type object, which must be cast to the array type.

```
int[] intArr1 = { 1, 2, 3 };
               Array type   Returns object
               ↓           ↓
int[] intArr2 = ( int[] ) intArr1.Clone();
```

For example, the following code shows an example of cloning a value type array, producing two independent arrays. Figure 14-16 illustrates the steps shown in the code.

```
static void Main()
{
    int[] intArr1 = { 1, 2, 3 };           // Step 1
    int[] intArr2 = (int[]) intArr1.Clone(); // Step 2

    intArr2[0] = 100; intArr2[1] = 200; intArr2[2] = 300; // Step 3
}
```

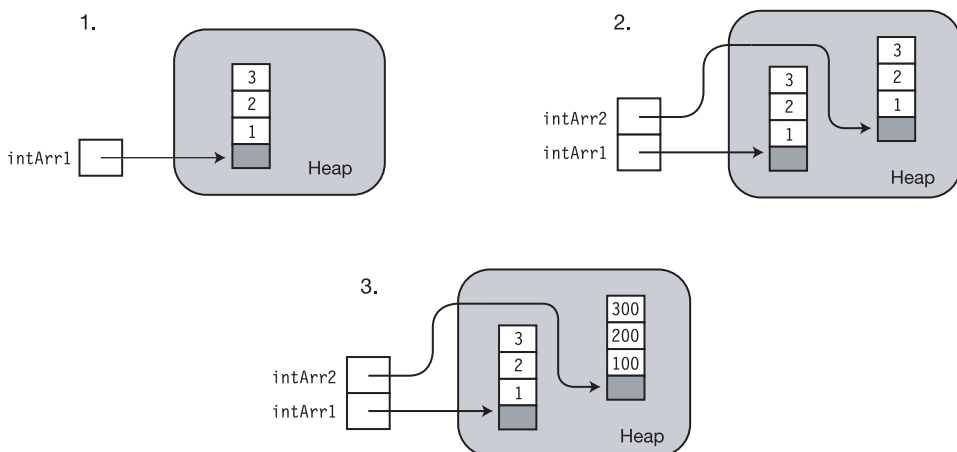


Figure 14-16. Cloning a value type array produces two independent arrays.

Cloning a reference type array results in two arrays *pointing at the same objects*. The following code shows an example. Figure 14-17 illustrates the steps shown in the code.

```
class A
{
    public int Value = 5;
}

class Program
{
    static void Main()
    {
        A[] AArray1 = new A[3] { new A(), new A(), new A() };    // Step 1
        A[] AArray2 = (A[]) AArray1.Clone();                    // Step 2

        AArray2[0].Value = 100;
        AArray2[1].Value = 200;
        AArray2[2].Value = 300;                                // Step 3
    }
}
```

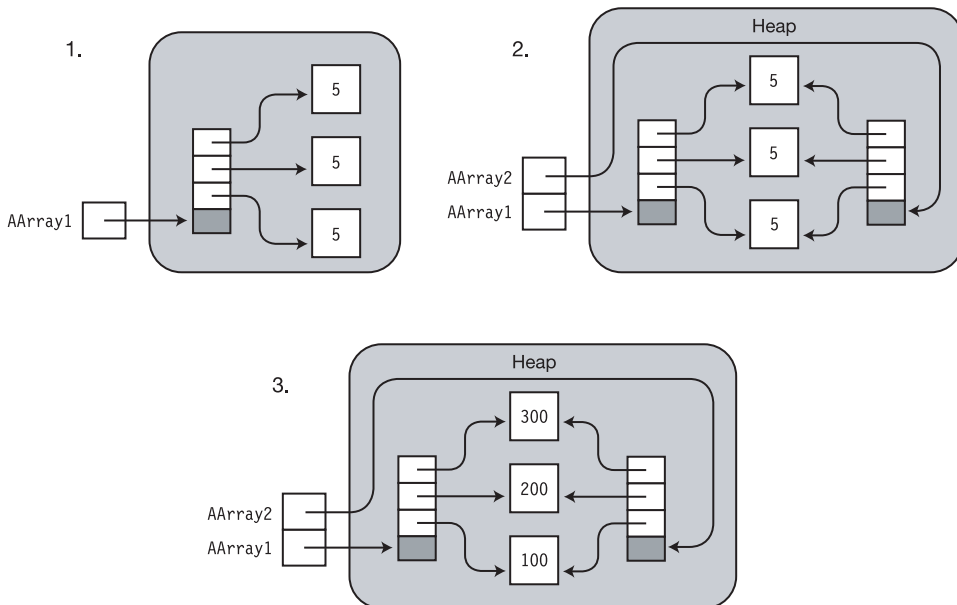

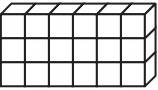
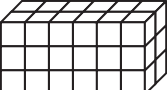
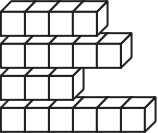


Figure 14-17. Cloning a reference type array produces two arrays referencing the same objects.

Comparing Array Types

Table 14-2 summarizes some of the important similarities and differences between the three types of arrays.

Table 14-2. *Summary Comparing Array Types*

Array Type	Array Objects	Syntax			Shape
		Brackets	Commas		
One-dimensional	1	Single set	No		One-Dimensional <code>int[3]</code>
<ul style="list-style-type: none"> Has optimizing instructions in CIL. 					
Rectangular	1	Single set	Yes		Two-Dimensional <code>int[3,6]</code>
<ul style="list-style-type: none"> Multidimensional All sub-arrays in a multidimensional array must be the same length. 					Three-Dimensional <code>int[3,6,2]</code>
Jagged	Multiple	Multiple sets	No		Jagged <code>int[4][]</code>
<ul style="list-style-type: none"> Multidimensional Sub-arrays can be of different lengths. 					

