

Contents

Arrays of Objects	1
Array of Objects From a Custom Class	1
Arrays of Arrays	2
Rectangular Multi-Dimensional Array	2
Jagged Array	3
Comparing Use Cases for Rectangular and Jagged Arrays .	4

Arrays of Objects

An array can contain more than simple datatypes: it can contain objects. It can be objects from a custom class, or even ... arrays, which are themselves objects!

Array of Objects From a Custom Class

In the following example, we will ask the user how many `Item` objects (the details of the implementation do not matter, but can be inspired by this example) they want to create, then fill an array with `Item` objects initialized from user input:

```
Console.WriteLine("How many items would you like to  
↪ stock?");  
Item[] items = new Item[int.Parse(Console.ReadLine())];  
int i = 0;  
while(i < items.Length)  
{  
    Console.WriteLine($"Enter description of item  
↪ {i+1}:");  
    string description = Console.ReadLine();  
    Console.WriteLine($"Enter price of item {i+1}:");  
    decimal price = decimal.Parse(Console.ReadLine());  
    items[i] = new Item(description, price);  
    i++;  
}
```

Observe that, since we do not perform any user-input validation, we can simply use the result of `int.Parse()` as the size declarator for the `items` array - no size variable is needed at all.

We can also use `while` loops to search through arrays for a particular value. For example, this code will find and display the lowest-priced item in the array `items`, which was initialized by user input:

```

Item lowestItem = items[0];
int i = 1;
while(i < items.Length)
{
    if(items[i].GetPrice() < lowestItem.GetPrice())
    {
        lowestItem = items[i];
    }
    i++;
}
Console.WriteLine($"The lowest-priced item is
↳ {lowestItem}");

```

Note that the `lowestItem` variable needs to be initialized to refer to an `Item` object before we can call the `GetPrice()` method on it; we cannot call `GetPrice()` if `lowestItem` is `null`. We could try to create an `Item` object with the “highest possible” price, but a simpler approach is to initialize `lowestItem` with `items[0]`. As long as the array has at least one element, `0` is a valid index, and the first item in the array can be our first “guess” at the lowest-priced item.

Arrays of Arrays

An array of arrays is called a multi-dimensional array. A multi-dimensional array can be rectangular (it then represents an n -dimensional block of memory) or jagged (in that case, it is an array of arrays).

Rectangular Multi-Dimensional Array

Also called 2-dimensional arrays, their syntax is very close to that of 1-dimensional arrays:

```
int[, ] matrix = new int[2, 3];
```

where `2` is the number of rows, and `3` is the number of columns. They can be accessed with `matrix.GetLength(0)` and `matrix.GetLength(1)` respectively.

Assignment is as for 1-dimensional arrays, starting at 0:

```

matrix[0, 0] = 1;
matrix[0, 1] = 2;
matrix[0, 2] = 3;
matrix[1, 0] = 4;
matrix[1, 1] = 5;
matrix[1, 2] = 6;

```

This will produce a matrix as follows:

	0th col.	1st col.	2nd col.
0th row	1	2	3
1st row	4	5	6

We could also have used a shortened notation to declare this 2-dimensional array, as follows:

```
int[,] matrix = new int[,]
{
    {1,2,3},
    {4,5,6}
};
```

or even simply

```
int[,] matrix = {{1,2,3},{4,5,6}};
```

To display such an array, nested loops are needed:

```
for (int row = 0; row < matrix.GetLength(0); row++)
{
    for (int col = 0; col < matrix.GetLength(1); col++)
        Console.Write(matrix[row, col] + " ");
    Console.WriteLine();
}
```

Jagged Array

A jagged array is an array of arrays. The difference from rectangular arrays is that the arrays stored can be of varying sizes. The syntax of a jagged array is as follows:

```
int [][] jaggedArray = new int[3][];
```

Here, 3 is the number of rows, but the number of columns is not included: this is because the stored arrays are different sizes, so the number of columns will vary depending on the row.

The syntax is straightforward once understood that jagged arrays are *exactly* arrays of arrays:

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

for (int row = 0; row < jaggedArray.Length; row++)
{
```

```

Console.WriteLine("The row #" + row + " contain: ");
for (
    int arrayCell = 0;
    arrayCell < jaggedArray[row].Length;
    arrayCell++)
{
    Console.WriteLine(jaggedArray[row][arrayCell] + " ");
}
Console.WriteLine("");
}

```

In this example, it should be clear that `jaggedArray[row]` is itself an array, and hence that we can use e.g., `jaggedArray[row].Length` or `jaggedArray[row][arrayCell]`.

Comparing Use Cases for Rectangular and Jagged Arrays

The main distinction between rectangular and jagged arrays is that the former is more consistent with its sizing—in other words, in a rectangular array, each array housed within it will be the same size in every row, by definition. In contrast, jagged arrays are not as consistent. This can be seen in the difference between how rectangular and jagged arrays are initialized. Recall that, when initializing a rectangular array, the number of rows and columns is required, but only the number of rows is required for a jagged array.

Therefore, when deciding which type of array to use in a given situation, you will want to base it off of whether you need an array that is predictable in its sizing. If you want to make a table where most or every cell will be filled, a rectangular array will make more sense, whereas a jagged array will be preferred when the number of entries per row could be highly variable or difficult to predict.

Rectangular arrays work well when creating a matrix (an array of numbers used in mathematics, especially in linear algebra) or a table/spreadsheet of data where the number of data points (and, therefore, the number of columns) is consistent in each row. Jagged arrays work well when the number of data points varies. A good example would be if you want to use an array to store the time it takes you to run a lap. If you run the same number of laps every single time you run, without fail, then a rectangular array may work here. However, the more likely scenario is that the number will vary. In this case, a jagged array is better: the array in each row will be whatever size it needs to be in order to accommodate all of the data.