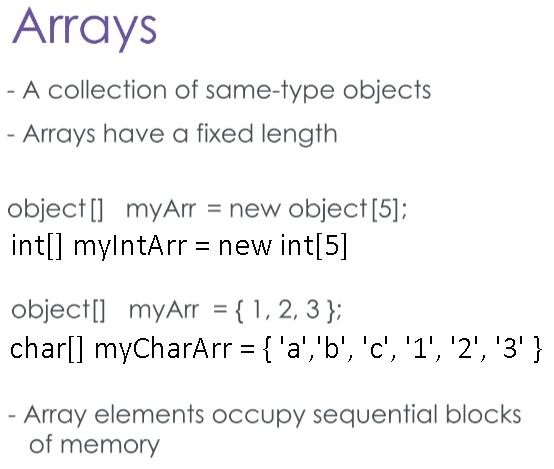
A Collection in C#, is simply an object that acts as a storage system for multiple other objects or values.

There are three broad types of collections under the .NET framework, though each has a range of more specialized subclasses.

The first type of collection is an **Array**, which you might have already seen before.

An array is a collection for data of the same type, with a fixed maximum size for the number of items it is allowed to contain.

The syntax for an array starts with a reference, which will be the data type for the elements, and this can either be a primitive type or a class reference type.

This is followed by the square brackets that indicate this variable is an array, and the usual reference name.

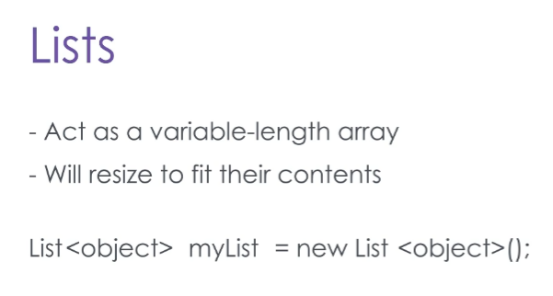
An array - no matter the type of the contents - is an object, and so the array is instantiated with the “new” keyword, restating the data type, and an integer value for the array’s size inside square brackets.

You can also pre-populate an array by using a different syntax.

Instead of instantiating an array with the new keyword, you include the contents of the array inside curly braces, like so.

Remember that an array can never change in size, so it’s size is fixed to the number of elements specified here.

Why are arrays fixed in size? This is because when you declare an array, .NET reserves spaces in memory for each element right away, and these spaces are next to one another in memory. This makes finding a value in an array incredibly fast, no matter how large that array gets. The CLR never has to look very hard for the element you need.

The next type of collection is the list, which is like a flexible array.

**Lists** can change in size, and will automatically do so when you add new elements.

To create a list, you start with the reference type, which will be List, followed by the type of the contents in angle brackets.

You then have the usual variable name, and you instantiate a list with the new keyword, which invokes the List constructor.

Remember to keep the content type between the angle brackets for the constructor call too.

Arrays and Lists both use an index to organize their contents, and this index is zero-based. This means that to get the first item in an array, you request the item at index 0. The second item is at index 1, and so on. Here we have created an array and a List, both of String objects.

The array we can prepopulate with string values, but for the List, we add them individually. Then we use a for-loop to iterate through the array, and print its contents.

Note that the count variable starts at 0, and the condition uses the Array’s size property to determine where to stop.

When we print, we print the values at the index specified by count. For the List, we could use a for-loop like the array, but instead we are using the foreach loop. This prevents us from being able to refer to the current index, but it is a slightly more concise way to write the loop. We can do this because both the List and the Array object implement the IEnumerable interface.

Lists have several other methods available for adding or removing elements. Like arrays, you can insert a new element at any index. But because lists are flexible in size, any elements with a higher index will be shifted up to make room. Similarly, if you remove an element, the contents of the rest of the list are shifted down to fill in the gap.

Arrays can also be multidimensional. In other words, you can have an array of arrays. An array of arrays of arrays, even.

int[ , ] table = new int[10, 10];

//for each element in the first array dimension…

for (int I = 1; i<=table.GetLength(0); ++i)

{

//iterate over each element in the second array dimension

for (int j = 1; j <= table.GetLength(1); ++j)

{

//Remember that array indices start at 0!

table[i-1, j-1] = I \* j;

Console.Out.Write(table[i-1, j-1] + “ “);

}

}

Console.Out.WriteLine();

}

Here is an example of a 10-element array of more 10-element arrays. Using two nested for-loops and the size of each array, we print the multiplication tables for 1-10.

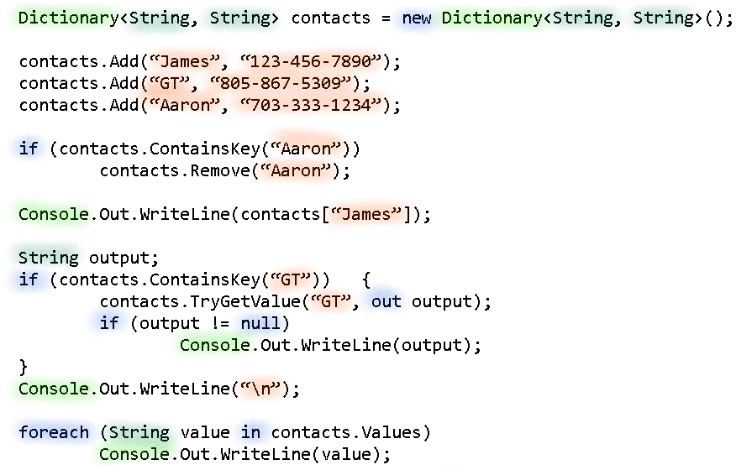
Note that when we want to refer to a specific element, we have to specify the index of both arrays it belongs to, like a coordinate on a graph.

The **dictionary** is the last type of collection we’ll discuss today. Unlike the Array and the List, a dictionary does not use an index to refer to its contents. Instead, it uses a key, which is mapped to a value.

The best example for this comes straight from the name – you use a dictionary to look up a definition for a given word. The word is the key, and the definition is the value.

Dictionaries are declared in a similar syntax to Lists, but you must have two types between the angle brackets: the first type is the type of the key, and the second type is the type of the value.

A dictionary’s keys must be unique, but the values do not have to be. Let’s look at another example to see how to use a dictionary.

Here in Visual Studio I have created a dictionary that uses strings as the keys and strings as the values.

I am using this as an address book, where the key-string will be a person’s name, and the value-string will be their phone number.

First I add a couple contacts to my dictionary.

Then, using the ContainsKey method, I search it for a particular contact. If that contact is found, I will delete it.

There are two ways to retrieve values from a dictionary. The first is similar to lists and arrays, but instead of referring to the index, you search for a value based on the key, as shown here.

Alternatively, and this is the safer way, you can use a combination of ContainsKey and the TryGetValue method to retrieve a certain value.

The TryGetValue method will return null if a value does not exist, so either use the ContainsKey method first, or guarantee that the rest of your logic could handle a potential null value.

Take it from me, nothing is more irritating than having an application crash because you got a null value where you weren’t expecting one.

You could also retrieve all the values from a dictionary, if you needed to iterate through them, by **using the Values Property**. This will return the values of a dictionary as a List, which of course implements the IEnumerable interface and so can be iterated through.

An example of this using the foreach loop is shown here.

These collections will allow you to manage large chunks of data all at once, by iterating across them.

It’s important to always use the right collection for a job.

An array might be the fastest collection for fetching, adding, or removing an element at a particular index, but their fixed sizes and lack of flexibility makes their usefulness much reduced compared to Lists.

Dictionaries should be used whenever you want to use something other than an integer to identify an element, or if the integer identification should be unique, but not necessarily sequential

Most professional applications involve performing operations to large collections of data all at once, so their efficiency at different tasks quickly becomes important to understand.