Module 04(Complex Data Structures)

Introducing Arrays

An array is a set of objects that are grouped together and managed as a unit. You can think of an array as a sequence of elements, all of which are the same type. You can build simple arrays that have one dimension (a list), two dimensions (a table), three dimensions (a cube), and so on. Arrays in Visual C# have the following features:

- Every element in the array contains a value.
- Arrays are zero-indexed, that is, the first item in the array is element 0.
- The size of an array is the total number of elements that it can contain.
- Arrays can be single-dimensional, multidimensional, or jagged.
- The rank of an array is the number of dimensions in the array.

Arrays of a particular type can only hold elements of that type. If you need to manipulate a set of unlike objects or value types, consider using one of the collection types that are defined in the System.Collections namespace.

Creating and Using Single Dimension Arrays

When you declare an array, you specify the type of data that it contains and a name for the array. Declaring an array brings the array into scope, but does not actually allocate any memory for it. The CLR physically creates the array when you use the new keyword. At this point, you should specify the size of the array.

To declare a single-dimensional array, you specify the type of elements in the array and use brackets, [] to indicate that a variable is an array. Later, you specify the size of the array when you allocate memory for the array by using the new keyword. The size of an array can be any integer expression. The following code example shows how to create a single-dimensional array of integers with elements zero through nine.

```
int[] arrayName = new int[10];
```

You can also choose to create an array and initialize it with values at the same time as in the following example that declares and integer array and assigns values to it. The compiler know how large to make the array by the number of values in the curly braces:

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

Accessing Data in an Array

You can access data in an array in several ways, such as by specifying the index of a specific element that you require or by iterating through the entire array and returning each element in sequence.

The following code example uses an index to access the element at index two.

```
//Accessing Data by Index
int[] oldNumbers = { 1, 2, 3, 4, 5 };
//number will contain the value 3
int number = oldNumbers[2];
```

Note: Arrays are zero-indexed, so the first element in any dimension in an array is at index zero. The last element in a dimension is at index N-1, where N is the size of the dimension. If you attempt to access an element outside this range, the CLR throws an IndexOutOfRangeException exception.

You can iterate through an array by using a *for* loop. You can use the *Length* property of the array to determine when to stop the loop.

The following code example shows how to use a for loop to iterate through an array.

```
//Iterating Over an Array
int[] oldNumbers = { 1, 2, 3, 4, 5 };
for (int i = 0; i < oldNumbers.Length; i++)
{
   int number = oldNumbers[i];
   ...
}
```

Multidimensional arrays

An array can have more than one dimension. The number of dimensions corresponds to the number of indices that are used to identify an individual element in the array. You can specify up to 32 dimensions, but you will rarely need more than three. You declare a multidimensional array variable just as you declare a single-dimensional array, but you separate the dimensions by using commas. The following code example shows how to create an array of integers with two dimensions.

```
// Create an array that is 10 long(rows) by 10 wide(columns)
int[ , ] arrayName = new int[10,10];
```

In order to access elements in a multidimensional array, you must include all indices as in the example code here.

```
// Access the element in the first row and first column
int value = arrayName[0,0]

//Access the element in the first row and second column
int value2 = arrayName[0, 1];

//Access the element in the second row and first column
int value2 = arrayName[1, 0];
```

Jagged arrays

A jagged array is simply an array of arrays, and the size of each array can vary. Jagged arrays are useful for modeling sparse data structures where you might not always want to allocate memory for every item if it is not going to be used. The following code example shows how to declare and initialize a jagged array. Note that you must specify the size of the first array, but you must not specify the size of the arrays that are contained within this array. You allocate memory to each array within a jagged array separately, by using the new keyword.

```
int[][] jaggedArray = new int[10][];
jaggedArray[0] = new Type[5]; // Can specify different sizes.
jaggedArray[1] = new Type[7];
...
jaggedArray[9] = new Type[21];
```

Introducing enums

An enumeration type, or enum, is a structure that enables you to create a variable with a fixed set of possible values. The most common example is to use an enum to define the day of the week. There are only seven possible values for days of the week, and you can be reasonably certain that these values will never change.

A best practice would be to define your enum directly within a namespace so that all classes in that namespace will have access to it, if needed. You can also nest your enums within classes or structs.

By default enum values start at 0 and each successive member is increased by a value of 1.

Creating and Using Enums

To create an enum, you declare it in your code file with the following syntax, which demonstrates creating an enum called Day, that contains the days of the week:

enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

By default enum values start at 0 and each successive member is increased by a value of 1. As a result, the previous enum 'Day' would contain the values:

- Sunday = 0
- Monday = 1
- Tuesday = 2
- etc.

You can change the default by specifying a starting value for your enum as in the following example.

enum Day { Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

In this example, Sunday is given the value 1 instead of the detaul 0. Now Monday is 2, Tuesday is 3, etc.

The keyword enum is used to specify the "type" that the variable Day will be. In this case, an enumeration type. Enums support intrinsic data types and can be any one of the following:

- byte
- sbyte
- short
- ushort
- int
- uint
- long
- ulong

In order to change the default data type of your enum, you precede the list with a data type from the list above, such as:

enum Day : short { Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

The underlying type specifies how much storage will be allocated for each enumerator in the enum. During compile time, your enum will be converted to numeric literals in your code. If

you are using Visual Studio, the Intellisense feature is fully capable of recognizing your enums and will display the string values automatically in the IDE as you type the enum name.

It's important to note that you will be required to use an explicit cast if you want to convert from an enum type to an integral type. Consider this example where the statement assigns the enumerator Sun to an int type, with a cast, to convert from enum to int.

```
int x = (int)Day.Sun;
```

Using an Enum

```
Day favoriteDay = Day.Friday;
```

Using enums has several advantages over using text or numerical types:

- Improved manageability. By constraining a variable to a fixed set of valid values, you are less likely to experience invalid arguments and spelling mistakes.
- Improved developer experience. In Visual Studio, the IntelliSense feature will prompt you with the available values when you use an enum.
- Improved code readability. The enum syntax makes your code easier to read and understand.

Each member of an enum has a name and a value. The name is the string you define in the braces, such as Sunday or Monday. By default, the value is an integer. If you do not specify a value for each member, the members are assigned incremental values starting with 0. For example, Day.Sunday is equal to 0 and Day.Monday is equal to 1.

The following example shows how you can use names and values interchangeably:

Using Enum Names and Values Interchangeably

```
// Set an enum variable by name.
Day favoriteDay = Day.Friday;
// Set an enum variable by value.
Day favoriteDay = (Day)4;
```

Introducing structs

In Visual C#, a struct is a programming construct that you can use to define custom types. Structs are essentially lightweight data structures that represent related pieces of information as a single item. For example:

- A struct named Point might consist of fields to represent an x-coordinate and a y-coordinate.
- A struct named Circle might consist of fields to represent an x-coordinate, a y-coordinate, and a radius.
- A struct named Color might consist of fields to represent a red component, a green component, and a blue component.

Most of the built-in types in Visual C#, such as int, bool, and char, are defined by structs. You can use structs to create your own types that behave like built-in types.

Creating a Struct

You use the struct keyword to declare a struct, as shown by the following example:

```
//Declaring a Struct
public struct Coffee
{
   public int Strength;
   public string Bean;
   public string CountryOfOrigin;
   // Other methods, fields, properties, and events.
}
```

The struct keyword is preceded by an access modifier— publicin the above example—that specifies where you can use the type. You can use the following access modifiers in your struct declarations:

Access Modifier Details

- public - The type is available to code running in any assembly.
- internal - The type is available to any code within the same assembly, but not available to code in another assembly. This is the default value if you do not specify an access modifier.

- private - The type is only available to code within the struct that contains it. You can only use the private access modifier with nested structs.

Structs can contain a variety of members including constructors, fields, constants, properties, indexers, methods, operators, events, and even nested types. Keep in mind that structs are intended to be lightweight therefore if you find yourself adding multiple methods, constructors, and events, you should consider using a class instead.

Using a Struct

To create an instance of a struct, you use the new keyword, as shown by the following example:

Instantiating a Struct

```
Coffee coffee1 = new Coffee();
coffee1.Strength = 3;
coffee1.Bean = "Arabica";
coffee1.CountryOfOrigin = "Kenya";
```

Initializing Structs

You might have noticed that the syntax for instantiating a struct, for example, new Coffee(), is similar to the syntax for calling a method. This is because when you instantiate a struct, you are actually calling a special type of method called a constructor. A constructor is a method in the struct that has the same name as the struct.

When you instantiate a struct with no arguments, such as new Coffee(), you are calling the default constructor which is created by the Visual C# compiler. If you want to be able to specify default field values when you instantiate a struct, you can add constructors that accept parameters to your struct.

The following example shows how to create a constructor in a struct:

Adding a Constructor

```
public struct Coffee
{
  // This is the custom constructor.
  public Coffee(int strength, string bean, string countryOfOrigin)
  {
    this.Strength = strength;
    this.Bean = bean;
    this.CountryOfOrigin = countryOfOrigin;
```

```
  }
 // These statements declare the struct fields and set the default values.
  public int Strength;
  public string Bean;
  public string CountryOfOrigin;
  // Other methods, fields, properties, and events.
}
```

The following example shows how to use this constructor to instantiate a Coffee item:

Calling a Constructor

```
// Call the custom constructor by providing arguments for the three required parameters.
Coffee coffee1 = new Coffee(4, "Arabica", "Colombia");
```

You can add multiple constructors to your struct, with each constructor accepting a different combination of parameters. However, you cannot add a default constructor to a struct because it is created by the compiler.

Extending structs

In order to go beyond a simple struct, you can extend it by adding properties and indexers. This section discusses using properties and indexers in your struct. Again, if you find yourself going to this extent, evaluate your use of structs against class files.

Creating Properties

In Visual C#, a *property* is a programming construct that enables client code to get or set the value of private fields within a struct or a class. To consumers of your struct or class, the property behaves like a public field. Within your struct or class, the property is implemented by using accessors, which are a special type of method. A property can include one or both of the following:

- A get accessor to provide read access to a field.
- A set accessor to provide write access to a field.

The following example shows how to implement a property in a struct:

```
//Implementing a Property
public struct Coffee
{
   private int strength;
```

```
    public int Strength
    {
        get { return strength; }
        set { strength = value; }
    }
}
```

Within the property, the *get* and *set* accessors use the following syntax:

- The get accessor uses the return keyword to return the value of the private field to the caller.
- The set accessor uses a special local variable named value to set the value of the private field. The value variable contains the value provided by the client code when it accessed the property.

The following example shows how to use a property:

```
//Using a Property
Coffee coffee1 = new Coffee();
// The following code invokes the set accessor. coffee1.Strength = 3;
// The following code invokes the get accessor. int coffeeStrength = coffee1.Strength;
```

The client code uses the property as if as it was a public field. However, using public properties to expose private fields offers the following advantages over using public fields directly:

- You can use properties to control external access to your fields. A property that includes only a get accessor is read-only, while a property that includes only a set accessor is write-only.

```
// This is a read-only property.
public int Strength
{
    get { return strength; }
}
// This is a write-only property.
public string Bean
{
    set { bean = value; }
}
```

- You can change the implementation of properties without affecting client code. For example, you can add validation logic, or call a method instead of reading a field value.

```
public int Strength
{
   get { return strength; }
   set
   {
      if(value < 1)
      { strength = 1; }
      else if(value > 5)
      { strength = 5; }
      else { strength = value; }
    }
}
```

- Properties are required for data binding in WPF. For example, you can bind controls to property values, but you cannot bind controls to field values.

  When you want to create a property that simply gets and sets the value of a private field without performing any additional logic, you can use an abbreviated syntax.

  To create a property that reads and writes to a private field, you can use the following syntax:

  ```
  public int Strength { get; set; }
  ```

- To create a property that reads from a private field, you can use the following syntax:

  ```
  public int Strength { get; }
  ```

- To create a property that writes to a private field, you can use the following syntax:

  ```
  public int Strength { set; }
  ```

  In each case, the compiler will implicitly create a private field and map it to your property. These are known as auto-implemented properties. You can change the implementation of your property at any time.

  Creating Indexers

  In some scenarios, you might want to use a struct or a class as a container for an array of values. For example, you might create a struct to represent the beverages available at a coffee shop. The struct might use an array of strings to store the list of beverages.

  The following example shows a struct that includes an array:

```
//Creating a Struct that Includes an Array
public struct Menu
{
   public string[] beverages;
   public Menu(string bev1, string bev2)
   {
      beverages = new string[] { "Americano", "Café au Lait", "Café Macchiato", "Cappuccino",
"Espresso" };
   }
}
```

When you expose the array as a public field, you would use the following syntax to retrieve beverages from the list:

```
//Accessing Array Items Directly
Menu myMenu = new Menu();
string firstDrink = myMenu.beverages[0];
```

A more intuitive approach would be if you could access the first item from the menu by using the syntax myMenu[0]. You can do this by creating an indexer. An indexer is similar to a property, in that it uses get and set accessors to control access to a field. More importantly, an indexer enables you to access collection members directly from the name of the containing struct or class by providing an integer index value. To declare an indexer, you use the this keyword, which indicates that the property will be accessed by using the name of the struct instance.

The following example shows how to define an indexer for a struct:

```
//Creating an Indexer
public struct Menu
{
   private string[] beverages;
   // This is the indexer.
   public string this[int index]
   {
      get { return this.beverages[index]; }
      set { this.beverages[index] = value; }
   }
   // Enable client code to determine the size of the collection.
   public int Length
   {
      get { return beverages.Length; }
```

```
    }
}
```

When you use an indexer to expose the array, you use the following syntax to retrieve the beverages from the list:

```
//Accessing Array Items by Using an Indexer
Menu myMenu = new Menu();
string firstDrink = myMenu[0];
int numberOfChoices = myMenu.Length;
```

Just like a property, you can customize the get and set accessors in an indexer without affecting client code. You can create a read-only indexer by including only a get accessor, and you can create a write-only indexer by including only a set accessor.

Module Four Assignment

Prior to object oriented considerations and class files, programmers created structs in languages such as C. Some programmers still use structs for storing related information, although the trend is more towards class files. Because there may still be occasions where a struct makes sense in your code, you're going to create some structs in this assignment. Because a struct has a similar layout to a class file, this will provide you with a layout for the variables in your student, teacher, program, and course aspects.

For this assignment, complete the following tasks. For the structs, just include member variables and a constructor. Do not create properties at this time. Include all the variables that you have created up to this point in time.

- Create a struct to represent a student
- Create a struct to represent a teacher
- Create a struct to represent a program
- Create a struct to represent a course
- Create an array to hold 5 student structs.
- Assign values to the fields in at least one of the student structs in the array
- Using a series of Console.WriteLine() statements, output the values for the student struct that you assigned in the previous step

When complete, submit your code in the peer review section.

Challenge

For this challenge, expand on the struct array steps to complete the following:

- Use an appropriate looping structure to add values to all student structs in the array by prompting a user of the application to enter values for fields.

  For example, if you created fristName, lastName, address, city fields as an example, for each of the 5 students in the array, you need to prompt the user for each field in the struct, for each student struct in the array.

- Once completed, create another loop to iterate over the array and write the values to the console window