Module Overview

Module Overview

- Implementing Structs and Enums
- Organizing Data into Collections
- Handling Events

Overview

To create effective applications by using Windows Presentation Foundation (WPF) or other .NET Framework platforms, you must first learn some basic Visual C# constructs. You need to know how to create simple structures to represent the data items you are working with. You need to know how to organize these structures into collections, so that you can add items, retrieve items, and iterate over your items. Finally, you need to know how to subscribe to events so that you can respond to the actions of your users.

In this module, you will learn how to create and use structs and enums, organize data into collections, and create and subscribe to events.

Objectives

After completing this module, you will be able to:

- Create and use structs and enums.
- Use collection classes to organize data.
- Create and subscribe to events.

Lesson 1: Implementing Structs and Enums

Lesson 1: Implementing Structs and Enums

- Creating and Using Enums
- Creating and Using Structs
- Initializing Structs
- Creating Properties
- Creating Indexers
- Demonstration: Creating and Using a Struct

Lesson Overview

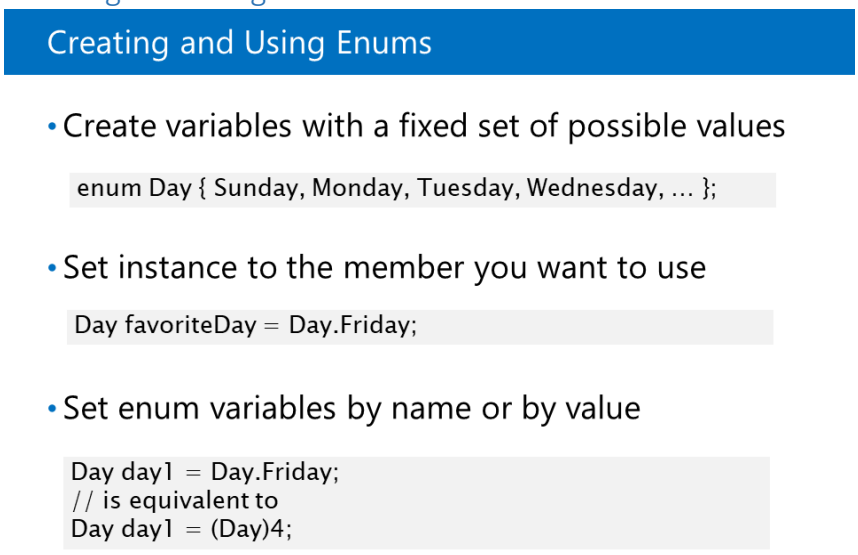
The .NET Framework includes various built-in data types, such as **Int32**, **Decimal**, **String**, and **Boolean**. However, suppose you want to create an object that represented a coffee. Which type would you use? You might use built-in types to represent the properties of a coffee, such as the country of origin (a string) or the strength of the coffee (an integer). However, you need a way to represent coffee as a discrete entity, so that you can perform actions such as add a coffee to a collection or compare one coffee to another.
In this lesson, you will learn how to use structs and enums to create your own simple types.

OBJECTIVES

After completing this lesson, you will be able to:

- Create and use enums.
- Create and use structs.
- Define constructors to instantiate structs.
- Create properties to get and set field values in a struct.
- Create indexers to expose struct members by using an integer index.

Creating and Using 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.
The following example shows how to create an enum:

Declaring an Enum

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

To use the enum, you create an instance of your enum variable and specify which enum member you want to use.
The following example shows how to use an enum:

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;
```

Reference Links: For more information about enums, see the Enumeration Types (C# Programming Guide) page at https://aka.ms/moc-20483c-m3-pg1.

Creating and Using Structs

Creating and Using Structs

- Use structs to create simple custom types:
  - Represent related data items as a single logical entity
  - Add fields, properties, methods, and events
- Use the **struct** keyword to create a struct

```
public struct Coffee { … }
```

- Use the **new** keyword to instantiate a struct

```
Coffee coffee1 = new Coffee();
```

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*— ***public*** in 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 fields, properties, methods, and events.

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

Initializing Structs

- Use constructors to initialize a struct

```
public struct Coffee
{
   public Coffee(int strength, string bean, string origin)
   { ... }
}
```

- Provide arguments when you instantiate the struct

```
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
```

- Add multiple constructors with different combinations of parameters

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", "Columbia");
```

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.

Creating Properties

Creating Properties

- Properties use get and set accessors to control access to private fields

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

- Properties enable you to:
  - Control access to private fields
  - Change accessor implementations without affecting clients
  - Data-bind controls to property values

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; }
}
}
```

- You can also create a **const** property by simply returning a literal value.
  `public string Hello { get { return "world"; }}`
- 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 can be publicly read, but set only by its containing class, you can use the following syntax:
  `public int Strength { get; private set; }`
- To create a **readonly** property where only the constructor can set the value to this property, 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 { private get; 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.

Additional Reading: In addition to controlling access to a property by omitting **get** or **set** accessors, you can also restrict access by applying access modifiers (such as **private** or **protected**) to your accessors. For example, you might create a property with a public **get** accessor and a protected **set** accessor. For more information, see the Restricting Accessor Accessibility (C# Programming Guide) page at https://aka.ms/moc-20483c-m3-pg2.

Creating Indexers

Creating Indexers

- Use the **this** keyword to declare an indexer
- Use **get** and **set** accessors to provide access to the collection

```
public int this[int index]
{
  get { return this.beverages[index]; }
  set { this.beverages[index] = value; }
}
```

- Use the instance name to interact with the indexer

```
Menu myMenu = new Menu();
string firstDrink = myMenu[0];
```

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.

Reference Links: For more information about indexers, see the Using Indexers (C# Programming Guide) page at https://aka.ms/moc-20483c-m3-pg3.

Demonstration: Creating and Using a Struct

Demonstration: Creating and Using a Struct

In this demonstration, you will learn how to:

- Create a custom struct
- Add properties to a custom struct
- Use a custom struct in the same way that you would use a standard .NET Framework type

In this demonstration, you will create a struct named **Coffee** and add several properties to the struct. You will then create an instance of the **Coffee** struct, set some property values, and display these property values in a console window.

Demonstration Steps

You will find the steps in the **Demonstration: Creating and Using a Struct** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD03_DEMO.md.

Lesson 2: Organizing Data into Collections

Lesson 2: Organizing Data into Collections

- Choosing Collections
- Standard Collection Classes
- Specialized Collection Classes
- Using List Collections
- Using Dictionary Collections
- Querying a Collection

Lesson Overview

When you create multiple items of the same type, regardless of whether they are integers, strings, or a custom type such as **Coffee**, you need a way of managing the items as a set. You need to be able to count the number of items in the set, add items to or remove items from the set, and iterate through the set one item at a time. To do this, you can use a *collection*.

Collections are an essential tool for managing multiple items. They are also central to developing graphical applications. Controls such as drop-down list boxes and menus are typically data-bound to collections.

In this lesson, you will learn how to use a variety of collection classes in Visual C#.

OBJECTIVES

After completing this lesson, you will be able to:

- Choose appropriate collections for different scenarios.
- Manage items in a collection.
- Use Language Integrated Query (LINQ) syntax to query a collection.

Choosing Collections

Choosing Collections

- *List* classes store linear collections of items
- *Dictionary* classes store collections of key/value pairs
- *Queue* classes store items in a first in, first out collection
- *Stack* classes store items in a last in, first out collection

All collection classes share various common characteristics. To manage a collection of items, you must be able to:

- Add items to the collection.
- Remove items from the collection.
- Retrieve specific items from the collection.
- Count the number of items in the collection.
- Iterate through the items in the collection, one item at a time.

Every collection class in Visual C# provides methods and properties that support these core operations. Beyond these operations, however, you will want to manage collections in different ways depending on the specific requirements of your application. Collection classes in Visual C# fall into the following broad categories:

- *List* classes store linear collections of items. You can think of a list class as a one-dimensional array that dynamically expands as you add items. For example, you might use a list class to maintain a list of available beverages at your coffee shop.
- *Dictionary* classes store a collection of key/value pairs. Each item in the collection consists of two objects—the *key* and the *value*. The value is the object you want to store and retrieve, and the key is the object that you use to index and look up the value. In most dictionary classes, the key must be unique, whereas duplicate values are perfectly acceptable. For example, you might use a dictionary class to maintain a list of coffee recipes. The key would contain the unique name of the coffee, and the value would contain the ingredients and the instructions for making the coffee.
- *Queue* classes represent a first in, first out collection of objects. Items are retrieved from the collection in the same order they were added. For example, you might use a queue class to process orders in a coffee shop to ensure that customers receive their drinks in turn.
- *Stack* classes represent a last in, first out collection of objects. The item that you added to the collection last is the first item you retrieve. For example, you might use a stack class to determine the 10 most recent visitors to your coffee shop.

When you choose a built-in collection class for a specific scenario, ask yourself the following questions:

- Do you need a list, a dictionary, a stack, or a queue?
- Will you need to sort the collection?
- How large do you expect the collection to get?
- If you are using a dictionary class, will you need to retrieve items by index as well as by key?
- Does your collection consist solely of strings?

If you can answer all of these questions, you will be able to select the Visual C# collection class that best meets your needs.

Standard Collection Classes

Standard Collection Classes

Class	Description
ArrayList	• General-purpose list collection • Linear collection of objects
BitArray	• Collection of Boolean values • Useful for bitwise operations and Boolean arithmetic (for example, AND, NOT, and XOR)
Hashtable	• General-purpose dictionary collection • Stores key/value object pairs
Queue	• First in, first out collection
SortedList	• Dictionary collection sorted by key • Retrieve items by index as well as by key
Stack	• Last in, first out collection

The **System.Collections** namespace provides a range of general-purpose collections that includes lists, dictionaries, queues, and stacks. The following table shows the most important collection classes in the **System.Collections** namespace:

Class	Description
ArrayList	The **ArrayList** is a general-purpose list that stores a linear collection of objects. The **ArrayList** includes methods and properties that enable you to add items, remove items, count the number of items in the collection, and sort the collection.

BitArray	The **BitArray** is a list class that represents a collection of bits as Boolean values. The **BitArray** is most commonly used for bitwise operations and Boolean arithmetic, and includes methods to perform common Boolean operations such as AND, NOT, and XOR.
Hashtable	The **Hashtable** class is a general-purpose dictionary class that stores a collection of key/value pairs. The **Hashtable** includes methods and properties that enable you to retrieve items by key, add items, remove items, and check for particular keys and values within the collection.
Queue	The **Queue** class is a first in, last out collection of objects. The **Queue** includes methods to add objects to the back of the queue (**Enqueue**) and retrieve objects from the front of the queue (**Dequeue**).
SortedList	The **SortedList** class stores a collection of key/value pairs that are sorted by key. In addition to the functionality provided by the **Hashtable** class, the **SortedList** enables you to retrieve items either by key or by index.
Stack	The **Stack** class is a first in, first out collection of objects. The **Stack** includes methods to view the top item in the collection without removing it (**Peek**), add an item to the top of the stack (**Push**), and remove and return the item at the top of the stack (**Pop**).

Reference Links: For more information about the classes listed in the previous table, see the System.Collections Namespace page at https://aka.ms/moc-20483c-m3-pg4.

Specialized Collection Classes

Specialized Collection Classes

Class	Description
ListDictionary	• Dictionary collection • Optimized for small collections (<10)
HybridDictionary	• Dictionary collection • Implemented as ListDictionary when small, changes to Hashtable as collection grows larger
OrderedDictionary	• Unsorted dictionary collection • Retrieve items by index as well as by key
NameValueCollection	• Dictionary collection in which both keys and values are strings • Retrieve items by index as well as by key
StringCollection	• List collection in which all items are strings
StringDictionary	• Dictionary collection in which both keys and values are strings
BitVector32	• Fixed size 32-bit structure • Represent values as Booleans or integers

The **System.Collections.Specialized** namespace provides collection classes that are suitable for more specialized requirements, such as specialized dictionary collections and strongly typed string collections. The following table shows the most important collection classes in the **System.Collections.Specialized** namespace:

Class	Description
ListDictionary	The **ListDictionary** is a dictionary class that is optimized for small collections. As a general rule, if your collection includes 10 items or fewer, use a **ListDictionary**. If your collection is larger, use a **Hashtable**.
HybridDictionary	The **HybridDictionary** is a dictionary class that you can use when you cannot estimate the size of the collection. The **HybridDictionary** uses a **ListDictionary** implementation when the collection size is small, and switches to a **Hashtable** implementation as the collection size grows larger.
OrderedDictionary	The **OrderedDictionary** is an indexed dictionary class that enables you to retrieve items by key or by index. Note that unlike the **SortedList** class, items in an **OrderedDictionary** are not sorted by key.
NameValueCollection	The **NameValueCollection** is an indexed dictionary class in which both the key and the value are strings. The **NameValueCollection** will throw an error if you attempt to set a key or a value to anything other than a string. You can retrieve items by key or by index.

StringCollection	The **StringCollection** is a list class in which every item in the collection is a string. Use this class when you want to store a simple, linear collection of strings.
StringDictionary	The **StringDictionary** is a dictionary class in which both the key and the value are strings. Unlike the **NameValueCollection** class, you cannot retrieve items from a **StringDictionary** by index.
BitVector32	The **BitVector32** is a struct that can represent a 32-bit value as both a bit array and an integer value. Unlike the **BitArray** class, which can expand indefinitely, the **BitVector32** struct is a fixed 32-bit size. As a result, the **BitVector32** is more efficient than the **BitArray** for small values. You can divide a **BitVector32** instance into sections to efficiently store multiple values.

Reference Links: For more information about the classes and structs listed in the previous table, see the System.Collections.Specialized Namespace page at https://aka.ms/moc-20483c-m3-pg5.

Using List Collections

Using List Collections

- Add objects of any type

```
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
ArrayList beverages = new ArrayList();
beverages.Add(coffee1);
```

- Retrieve items by index

```
Coffee firstCoffee = (Coffee)beverages[0];
```

- Use a foreach loop to iterate over the collection

```
foreach(Coffee c in beverages)
{
  // Console.WriteLine(c.CountryOfOrigin);
}
```

The **ArrayList** stores items as a linear collection of objects. You can add objects of any type to an **ArrayList** collection, but the **ArrayList** represents each item in the collection as a **System.Object** instance. When you add an item to an **ArrayList** collection, the **ArrayList** implicitly *casts*, or converts, your item to the **Object** type. When you retrieve items from the collection, you must explicitly cast the object back to its original type.

The following example shows how to add and retrieve items from an **ArrayList** collection:

Adding and Retrieving Items from an ArrayList

```
// Create a new ArrayList collection.
ArrayList beverages = new ArrayList();
// Create some items to add to the collection.
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
Coffee coffee2 = new Coffee(3, "Arabica", "Vietnam");
Coffee coffee3 = new Coffee(4, "Robusta", "Indonesia");
// Add the items to the collection.
// Items are implicitly cast to the Object type when you add them.
beverages.Add(coffee1);
beverages.Add(coffee2);
beverages.Add(coffee3);
// Retrieve items from the collection.
// Items must be explicitly cast back to their original type.
Coffee firstCoffee = (Coffee)beverages[0];
Coffee secondCoffee = (Coffee)beverages[1];
```

When you work with collections, one of your most common programming tasks will be to iterate over the collection. Essentially, this means that you retrieve each item from the collection in turn, usually to render a list of items, to evaluate each item against some criteria, or to extract specific member values from each item. To iterate over a collection, you use a **foreach** loop. The **foreach** loop exposes each item from the collection in turn, using the variable name you specify in the loop declaration.

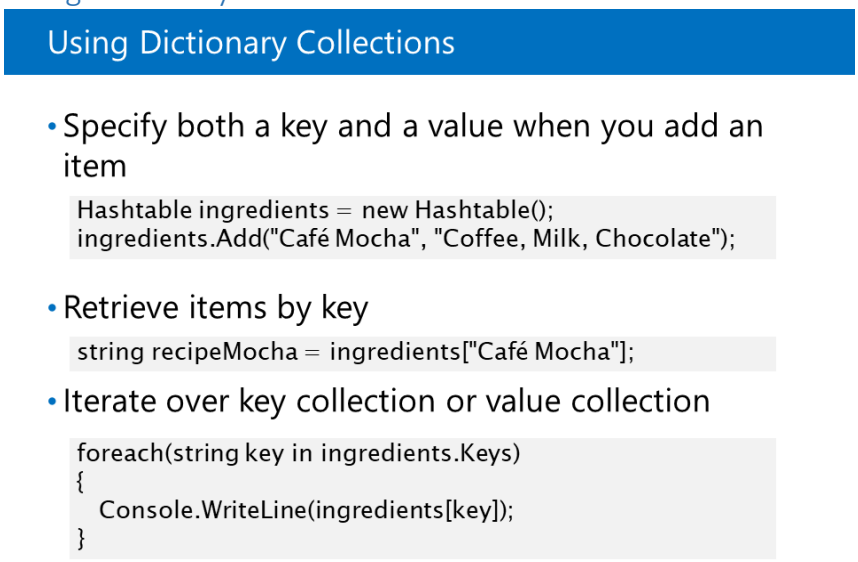
The following example shows how to iterate over an ArrayList collection:

Iterating Over a List Collection

```
foreach(Coffee coffee in beverages)
{
Console.WriteLine("Bean type: {0}", coffee.Bean);
Console.WriteLine("Country of origin: {0}", coffee.CountryOfOrigin);
Console.WriteLine("Strength (1-5): {0}", coffee.Strength);
}
```

Reference Links: For more information on the ArrayList class, see the ArrayList Class page at https://aka.ms/moc-20483c-m3-pg6.

Using Dictionary Collections



Dictionary classes store collections of key/value pairs. One useful dictionary class is the **Hashtable**. When you add an item to a **Hashtable** collection, you must specify a *key* and a *value*. Both the key and the value can be instances of any type, but the **Hashtable** implicitly casts both the key and the value to the **Object** type. When you retrieve values from the collection, you must explicitly cast the object back to its original type.

The following example shows how to add and retrieve items from a **Hashtable** collection. In this case both the key and the value are strings:

Adding and Retrieving Items from a Hashtable

```
// Create a new Hashtable collection.
Hashtable ingredients = new Hashtable();
// Add some key/value pairs to the collection.
ingredients.Add("Café au Lait", "Coffee, Milk");
ingredients.Add("Café Mocha", "Coffee, Milk, Chocolate");
ingredients.Add("Cappuccino", "Coffee, Milk, Foam");
ingredients.Add("Irish Coffee", "Coffee, Whiskey, Cream, Sugar");
ingredients.Add("Macchiato", "Coffee, Milk, Foam");
// Check whether a key exists.
if(ingredients.ContainsKey("Café Mocha"))
{
// Retrieve the value associated with a key.
Console.WriteLine("The ingredients of a Café Mocha are: {0}", ingredients["Café Mocha"]);
}
```

Dictionary classes, such as the **Hashtable**, actually contain two enumerable collections—the keys and the values. You can iterate over either of these collections. In most scenarios, however, you are likely to iterate through the key collection, for example to retrieve the value associated with each key in turn.

The following example shows how to iterate over the keys in a **Hashtable** collection and retrieve the value associated with each key:

Iterating Over a Dictionary Collection

```
foreach(string key in ingredients.Keys)
{
// For each key in turn, retrieve the value associated with the key.
Console.WriteLine("The ingredients of a {0} are {1}", key, ingredients[key]);
}
```

Reference Links: For more information on the Hashtable class, see the Hashtable Class page at https://aka.ms/moc-20483c-m3-pg7.

Querying a Collection

Querying a Collection

- Use LINQ expressions to query collections

```
var drinks =
  from string drink in prices.Keys
  orderby prices[drink] ascending
  select drink;
```

- Use extensions methods to retrieve specific items from results

```
decimal lowestPrice = drinks.FirstOrDefault();
decimal highestPrice = drinks.Last();
```

LINQ is a query technology that is built in to .NET languages such as Visual C#. LINQ enables you to use a standardized, declarative query syntax to query data from a wide range of data sources, such as .NET collections, SQL Server databases, ADO.NET datasets, and XML documents. A basic LINQ query uses the following syntax:

from <variable names> **in** <data source>
where <selection criteria>
orderby <result ordering criteria>
select <variable names>

For example, suppose you use a **Hashtable** to maintain a price list for beverages at Fourth Coffee. You might use a LINQ expression to retrieve all the drinks that meet certain price criteria, such as drinks that cost less than $2.00.
The following example shows how to use a LINQ expression to query a **Hashtable**:

Using LINQ to Query a Collection

```
// Create a new Hashtable and add some drinks with prices.
Hashtable prices = new Hashtable();
prices.Add("Café au Lait", 1.99M);
prices.Add("Caffe Americano", 1.89M);
prices.Add("Café Mocha", 2.99M);
prices.Add("Cappuccino", 2.49M);
prices.Add("Espresso", 1.49M);
prices.Add("Espresso Romano", 1.59M);
prices.Add("English Tea", 1.69M);
prices.Add("Juice", 2.89M);
// Select all the drinks that cost less than $2.00, and order them by cost.
var bargains =
from string drink in prices.Keys
where (Decimal)prices[drink] < 2.00M
orderby  prices[drink] ascending
select drink;
// Display the results.
foreach(string bargain in bargains)
{
Console.WriteLine(bargain);
}
Console.ReadLine();
```

Note: Appending the suffix m or M to a number indicates that the number should be treated as a decimal type.
In addition to this basic query syntax, you can call a variety of methods on your query results. For example:

- Call the **FirstOrDefault** method to get the first item from the results collection, or a default value if the collection contains no results. This method is useful if you have ordered the results of your query.
- Call the **Last** method to get the last item from the results collection. This method is useful if you have ordered the results of your query.
- Call the **Max** method to find the largest item in the results collection.
- Call the **Min** method to find the smallest item in the results collection.

Note: Most built-in types provide methods that enable you to compare one instance to another to determine which is considered larger or smaller. The **Max** and **Min** methods rely on these methods to find the largest or smallest

items. If your collection contains numerical types, these methods will return the highest and lowest values, respectively. If your collection contains strings, members are compared alphabetically—for example, "Z" is considered greater than "A". If your collection contains custom types, the **Max** and **Min** methods will use the comparison logic created by the type developer.

For example, if you have ordered your results by ascending cost, the first item in the results collection will be the cheapest and the last item in the results collection will be the most expensive. As such, you can use the **FirstOrDefault** and **Last** methods to find the cheapest and most expensive drinks, respectively.

The following example shows how to retrieve the smallest and largest items from a collection based on the sort criteria in the LINQ expression:

Using the FirstOrDefault and Last Methods

```
// Query the Hashtable to order drinks by cost.
var drinks =
from string drink in prices.Keys
orderby prices[drink] ascending
select drink;
Console.WriteLine("The cheapest drink is {0}: ", drinks.FirstOrDefault());
// Output: "The cheapest drink is Espresso"
Console.WriteLine("The most expensive drink is {0}: ", drinks.Last());
// Output: "The most expensive drink is Café Mocha"
Console.WriteLine("The maximum is {0}: ", drinks.Max());
// Output: "The maximum is Juice"
// "Juice" is the largest value in the collection when ordered alphabetically.
Console.WriteLine("The minimum is {0}: ", drinks.Min());
// Output: "The minimum is Café au Lait"
// "Café au Lait" is the smallest value in the collection when ordered alphabetically.
```

Note: The return type of a LINQ expression is **IEnumerable<T>**, where T is the type of the items in the collection. **IEnumerable<T>** is an example of a generic type. The methods you use on a results set, such as **FirstOrDefault**, **Last, Max**, and **Min**, are extension methods. Generic types and extension methods are covered later in this course.

Reference Links: For more information about using LINQ to query collections, see the LINQ Query Expressions (C# Programming Guide) page at https://aka.ms/moc-20483c-m3-pg8.

Lesson 3: Handling Events

Lesson 3: Handling Events

- Creating Events and Delegates
- Raising Events
- Subscribing to Events
- Demonstration: Working with Events in XAML
- Demonstration: Writing Code for the Grades Prototype Application Lab

Lesson Overview

Events are mechanisms that enable objects to notify other objects when something happens. For example, controls on a web page or in a WPF user interface generate events when a user interacts with the control, such as by clicking a button. You can create code that *subscribes* to these events and takes some action in response to an event.

Without events, your code would need to constantly read control values to look for any changes in state that require action. This would be a very inefficient way of developing an application. In this lesson, you will learn how to create, or *raise*, events, and how to subscribe to events.

In this lesson, you will learn how to handle events.

OBJECTIVES

After completing this lesson, you will be able to:

- Create events and delegates.
- Raise events.
- Subscribe to events and unsubscribe from events.

Delegates and Events - Introduction

Delegates and Events - Introduction

- Delegates
  - Delegates are reference to methods.
  - They can be saved as field and passed as parameters.
  - They allow one to execute code that is provided from external sources.
  - Delegates can be executed exactly like methods.
- Events
  - An event wraps a delegate like a property wraps a field.
  - Events allow the class to notify other consumers of actions performed within it.
  - An event may only be raised by it's containing class.

There are be many cases when building an application, where you'll want to perform an action that's unknown in your current class or context, and that action will be supplied from an external source. For example, say there is a method that writes text to an output. You may want to let the user of that method configure the output beforehand: set the font, text size, etc. Perhaps the user may want to run some other complex logic before writing the text. Your method will need some way to receive and run that supplied code.

This is the problem that the delegates solve. A delegate can be thought of as reference to a method. You can save it as a field or variable and pass it along like any other reference type. The only difference is that where type fields save data, a Delegate saves logic. As such, a delegate can be invoked exactly like a method. It can accept parameters, and it can return a result.

Just as regular fields have properties to restrict access and protect the usage of their underlying field, delegates have events. Events encapsulate their delegate and prevent anyone from raising them or overriding their value. A consuming class can never assign a value to an event, like it can a delegate. It can only subscribe and unsubscribe to and from the event. Only the containing class may raise (invoked) the event.

Although it's easy to technically equate events to properties as they relate to their respective delegates and fields, it's not exactly true. While properties simply provide a protected access to their field, events are conceptually different from delegates, even though they rely on them. While delegates are often used to run the logic provided to the process from outside, events are used to notify other classes or consumers of the class that a certain action has just occurred (in other words – that an event has happened). Therefore, events are seldom used to influence the current process (though that's possible), but to start other processes in the subscribed targets.

Creating Events and Delegates

Creating Events and Delegates

- Create a delegate for the event

```
public delegate void OutOfBeansHandler(Coffee coffee,
EventArgs args);
```

- Create the event and specify the delegate

```
public event OutOfBeansHandler OutOfBeans;
```

When you create an event in a struct or a class, you need a way of enabling other code to subscribe to your event. In Visual C#, you accomplish this by creating a *delegate*. A delegate is a special type that defines a method signature; in other words, the return type and the parameters of a method. As the name suggests, a delegate behaves like a representative for methods with matching signatures.

When you define an event, you associate a delegate with your event. To subscribe to the event from client code, you need to:

- Create a method with a signature that matches the event delegate. This method is known as the *event handler*.
- Subscribe to the event by giving the name of your event handler method to the *event publisher*, in other words, the object that will raise the event.

When the event is raised, the delegate invokes all the event handler methods that have subscribed to the event.

Suppose you create a struct named **Coffee**. One of the responsibilities of this struct is to keep track of the stock level for each **Coffee** instance. When the stock level drops below a certain point, you might want to raise an event to warn an ordering system that you are running out of beans.

The first thing you need to do is to define a delegate. To define a delegate, you use the **delegate** keyword. A delegate includes two parameters:

- The first parameter is the object that raised the event—in this case, a **Coffee** instance.
- The second parameter is the event arguments—in other words, any other information that you want to provide to consumers. This must be an instance of the **EventArgs** class, or an instance of a class that derives from **EventArgs**.

Next, you need to define the event. To define an event, you use the **event** keyword. You precede the name of your event with the name of the delegate you want to associate with your event.

The following example shows how to define delegates and events:

Defining a Delegate and an Event

```
public struct Coffee
{
public EventArgs e;
public delegate void OutOfBeansHandler(Coffee coffee, EventArgs args);
public event OutOfBeansHandler OutOfBeans;
}
```

In this example, you define an event named **OutOfBeans**. You associate a delegate named **OutOfBeansHandler** with your event. The **OutOfBeansHandler** delegate takes two parameters, an instance of **Coffee** that will represent the object that raised the event and an instance of **EventArgs** that could be used to provide more information about the event.

Raising Events

Raising Events

- Check whether the event is null
- Raise the event by using method syntax

```
if (OutOfBeans != null)
{
  OutOfBeans(this, e);
}
```

After you have defined an event and a delegate, you can write code that raises the event when certain conditions are met. When you raise the event, the delegate associated with your event will invoke any event handler methods that have subscribed to your event.

To raise an event, you need to do two things:

1. Check whether the event is null. The event will be null if no code is currently subscribing to it.
2. Invoke the event and provide arguments to the delegate.

For example, suppose a **Coffee** struct includes a method named **MakeCoffee**. Every time you call the **MakeCoffee** method, the method reduces the stock level of the **Coffee** instance. If the stock level drops below a certain point, the **MakeCoffee** method will raise an **OutOfBeans** event.

The following example shows how to raise an event:

Raising an Event

```
public struct Coffee
{
// Declare the event and the delegate.
public EventArgs e = null;
public delegate void OutOfBeansHandler(Coffee coffee, EventArgs args);
public event OutOfBeansHandler OutOfBeans;
int currentStockLevel;
int minimumStockLevel;
public void MakeCoffee()
{
// Decrement the stock level.
currentStockLevel--;
// If the stock level drops below the minimum, raise the event.
if (currentStockLevel < minimumStockLevel)
{
// Check whether the event is null.
if (OutOfBeans != null)
{
// Raise the event.
OutOfBeans(this, e);
}
}
}
}
```

To raise the event, you use a similar syntax to calling a method. You provide arguments to match the parameters required by the delegate. The first argument is the object that raised the event. Note how the **this** keyword is used to indicate the current **Coffee** instance. The second parameter is the **EventArgs** instance, which can be null if you do not need to provide any other information to subscribers.

Subscribing to Events

Subscribing to Events

- Create a method that matches the delegate signature

```
public void HandleOutOfBeans(Coffee c, EventArgs e)
{
  // Do something useful here.
}
```

- Subscribe to the event

```
coffee1.OutOfBeans += HandleOutOfBeans;
```

- Unsubscribe from the event

```
coffee1.OutOfBeans += HandleOutOfBeans;
```

If you want to handle an event in client code, you need to do two things:

- Create a method with a signature that matches the delegate for the event.
- Use the addition assignment operator (+=) to attach your event handler method to the event.

For example, suppose you have created an instance of the **Coffee** struct named **coffee1**. In your **Inventory** class, you want to subscribe to the **OutOfBeans** that may be raised by **coffee1**.

Note: The previous topic shows how the Coffee struct, the OutOfBeans event, and the OutOfBeansHandler delegate are defined.

The following example shows how to subscribe to an event:

Subscribing to an Event

```
public class Inventory
{
public void HandleOutOfBeans(Coffee sender, EventArgs args)
{
string coffeeBean = sender.Bean;
// Reorder the coffee bean.
}
public void SubscribeToEvent()
{
coffee1.OutOfBeans += HandleOutOfBeans;
}
}
```

In this example, the signature of the **HandleOutOfBeans** method matches the delegate for the **OutOfBeans** event. When you call the **SubscribeToEvent** method, the **HandleOutOfBeans** method is added to the list of subscribers for the **OutOfBeans** event on the **coffee1** object.

To unsubscribe from an event, you use the subtraction assignment operator (-=) to remove your event handler method from the event. It's safe to unsubscribe from an event to which you're not subscribed. In that case nothing will happen, no exception will be thrown.

The following example shows how to unsubscribe from an event:

Unsubscribing from an Event

```
public void UnsubscribeFromEvent()
{
coffee1.OutOfBeans -= HandleOutOfBeans;
}
```

It's important to unsubscribe to events when you are done with them or the subscribed object. A subscription to an event saves a reference to the subscribed object instance. Failing to unsubscribe may result in a memory leak.

Reference Links: For more information see How to: Subscribe to and Unsubscribe from Events (C# Programming Guide) page at https://aka.ms/moc-20483c-m3-pg9.

Demonstration: Working with Events in XAML

Demonstration: Working with Events in XAML

In this demonstration, you will learn how to:

- Create an event handler for a button click event in XAML
- Use the event handler to set the contents of a label

Visual Studio provides tools that make it easy to work with events in WPF applications. In this demonstration, you will learn how to subscribe to events raised by WPF controls.

Demonstration Steps

You will find the steps in the **Demonstration: Working with Events in XAML** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD03_DEMO.md.

Demonstration: Writing Code for the Grades Prototype Application Lab

Demonstration: Writing Code for the Grades Prototype Application Lab

In this demonstration, you will learn about the tasks that you will perform in the lab for this module

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Demonstration Steps

You will find the steps in the **Demonstration: Writing Code for the Grades Prototype Application Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD03_DEMO.md.

Lab Scenario

Lab Scenario

The School of Fine Arts has decided that they want to extend their basic class enrollment application to enable teachers to record the grades that students in their class have achieved for each subject, and to allow students to view their own grades. This functionality necessitates implementing application log on functionality to authenticate the user and to determine whether the user is a teacher or a student.

You decide to start by developing parts of a prototype application to test proof of concept and to obtain client feedback before embarking on the final application. The prototype application will use basic WPF views rather than separate forms for the user interface. These views have already been designed and you must add the code to navigate among them.

You also decide to begin by storing the user and grade information in basic structs, and to use a dummy data source in the application to test your log on functionality.

Lab: Writing the Code for the Grades Prototype Application

Lab: Writing the Code for the Grades Prototype Application

- Exercise 1: Adding Navigation Logic to the Grades Prototype Application
- Exercise 2: Creating Data Types to Store User and Grade Information
- Exercise 3: Displaying User and Grade Information

Estimated Time: 90 minutes

Scenario

The School of Fine Arts has decided that they want to extend their basic class enrollment application to enable teachers to record the grades that students in their class have achieved for each subject, and to allow students to view their own grades. This functionality necessitates implementing application log on functionality to authenticate the user and to determine whether the user is a teacher or a student.

You decide to start by developing parts of a prototype application to test proof of concept and to obtain client feedback before embarking on the final application. The prototype application will use basic WPF views rather than separate forms for the user interface. These views have already been designed and you must add the code to navigate among them.

You also decide to begin by storing the user and grade information in basic structs, and to use a dummy data source in the application to test your log on functionality.

Objectives

After completing this lab, you will be able to:

- Navigate among views.
- Create and use collections of structs.
- Handle events.

Lab Setup

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD03_LAB_MANUAL.md.
You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD03_LAK.md.

Module Review and Takeaways

Module Review and Takeaways

- Review Questions

In this module, you learned about some of the core features provided by the .NET Framework and Microsoft Visual Studio®. You also learned about some of the core Visual C#® constructs that enable you to start developing .NET Framework applications.

Review Questions

1. You want to create a string property named **CountryOfOrigin**. You want to be able to read the property value from any code, but you should only be able to write to the property from within the containing struct. How should you declare the property?

( ) public string CountryOfOrigin { get; set; }
( ) public string CountryOfOrigin { get; }
( ) public string CountryOfOrigin { set; }
( ) public string CountryOfOrigin { get; private set; }

2. You want to create a collection to store coffee recipes. You must be able to retrieve each coffee recipe by providing the name of the coffee. Both the name of the coffee and the coffee recipe will be stored as strings. You also need to be able to retrieve coffee recipes by providing an integer index. Which collection class should you use?

( ) ArrayList
( ) Hashtable
( ) SortedList
( ) NameValueCollection
( ) StringDictionary

3.You are creating a method to handle an event named **OutOfBeans**. The delegate for the event is as follows:

```
public delegate void OutOfBeansHandler(Coffee coffee, EventArgs args);
```

Which of the following methods should you use to subscribe to the event?

( ) public void HandleOutOfBeans(delegate OutOfBeansHandler)
{
}
( ) public void HandleOutOfBeans(Coffee c, EventArgs e)
{
}
( ) public Coffee HandleOutOfBeans(EventArgs e)
{
}

( ) public Coffee HandleOutOfBeans(Coffee coffee, EventArgs args)
{
}
( ) public void HandleOutOfBeans(Coffee coffee)
{
}