Microsoft® Official Course

Module 3

Developing the Code for a Graphical Application

Microsoft®

Module Overview

- To create effective graphical 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.
  - how to organize these structures into collections, so that you can add items, retrieve items, and iterate over your items.
  - how to subscribe to events so that you can respond to the actions of your users.

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

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

Creating and Using Enums

- Create variables with a fixed set of possible values

```
enum Day { Sunday, Monday, Tuesday, Wednesday, … };
```

- Set instance to the member you want to use

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

- Set enum variables by name or by value

```
Day day1 = Day.Friday;
// is equivalent to
Day day1 = (Day)4;
```

- Advantages over using text or numerical types:
  - Improved manageability.
    - By constraining a variable to a fixed set of valid values, you're less likely to experience invalid arguments & 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.

Creating and Using Structs

next module covers classes

- Use structs to create simple custom types (that hold multiple values):
  - Use to represent related data items as a single logical entity
  - To it one can add fields, properties, methods, and events
    - Example … Point(x,y), Circle(x,y,r),…

- Use the **struct** keyword to create a struct

```
public struct Coffee { … }
```

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

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

- The **struct** keyword is preceded by an access modifier (above public):
  - Structs can contain a variety of members, including fields, properties, methods, and events.

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.

Initializing Structs ← similar for classes!!!

- 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");
```

- A constructor is a method in the struct that has the **same name** as the struct.
  - Default constructors ... details
  - Default values ... details
  - One can add multiple constructors with different combinations of parameters (overloading ...)

- Use of this ...to enhance readability

Creating Properties

- Properties use get and set accessors to control access to private fields
  - A **get** accessor provides read access to a field.
    - uses the **return** keyword
    - a property that includes only a get accessor is **read-only**
  - A **set** accessor to provide write access to a field.
    - **value** variable contains the value provided by the client code
    - a property that includes only a set accessor is **write-only**

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

- Properties enable you to:
  - Control access to private fields (get/set, access modifiers)
  - Change accessor implementations without affecting clients
    - For example you can add validation logic …
  - Data-bind controls to property values
    - 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.

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

- auto-implemented properties:
  - the compiler will implicitly create a private field and map it to your property.

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

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

Creating Indexers

- In some scenarios, you might want to use a struct or a class as a container for an array of values.

- Use the **this** keyword to declare an indexer
  - this keyword indicates that the property will be accessed by using the name of the struct instance
- 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];
```

- Without the indexer you would have to use something like:

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

A few extra slides …

Next few slides provide more details about:

- Value type vs reference type
- Struct vs Class

Struct vs Class - EXTRA

- See also https://youtu.be/AGNW0jH1sn0
  - Part 29 - C# Tutorial - Difference between classes and structs in c#.avi

Classes Vs Structs

A struct is a value type where as a class is a reference type.

All the differences that are applicable to value types and reference types are also applicable to classes and structs.

Structs are stored on stack, where as classes are stored on the heap.

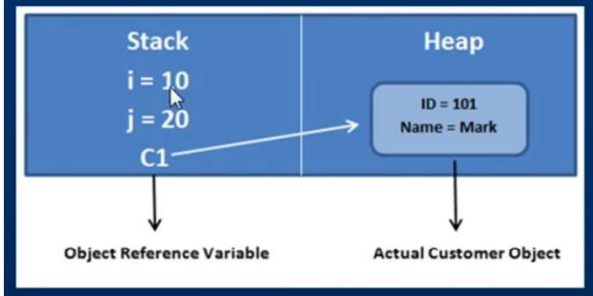
Value types hold their value in memory where they are declared, but reference types hold a reference to an object in memory.

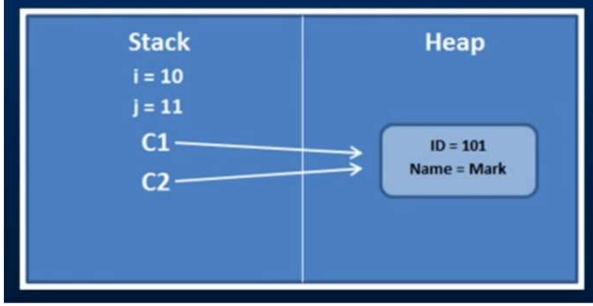
Value types are destroyed immediately after the scope is lost, where as for reference types only the reference variable is destroyed after the scope is lost. The object is later destroyed by garbage collector. (We will talk about this in the garbage collection session)

When you copy a struct into another struct, a new copy of that struct gets created and modifications on one struct will not affect the values contained by the other struct.

When you copy a class into another class, we only get a copy of the reference variable. Both the reference variables point to the same object on the heap. So, operations on one variable will affect the values contained by the other reference variable.

Stack and Heap





Struct is value type - EXTRA

- What will the following output? Why?

```
public class TestMe
{
    static void Main(string[] args)
    {
        TestClass t1 = new TestClass();
        t1.x = 10;
        AddOne(t1);

        Console.WriteLine(t1.x );
    }

    public static void AddOne(TestClass a)
    {
        a.x++;
    }

    public class TestClass
    {
        public int x;
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
11
Press any key to continue . . .
```

```
public class TestMe
{
    static void Main(string[] args)
    {
        TestClass t1 = new TestClass();
        t1.x = 10;
        AddOne(t1);

        Console.WriteLine(t1.x );
    }

    public static void AddOne(TestClass a)
    {
        a.x++;
    }

    public struct TestClass
    {
        public int x;
    }
}
```

```
Select C:\WINDOWS\system32\cmd.exe
10
Press any key to continue . . .
```

Struct is value type - EXTRA

- What will the following output? Why?

```
public class TestMe
{
    static void Main(string[] args)
    {
        TestClass t1 = new TestClass();
        t1.x = 10;
        AddOne(ref t1);

        Console.WriteLine(t1.x );
    }

    public static void AddOne(ref TestClass a)
    {
        a.x++;
    }

    public struct TestClass
    {
        public int x;
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
11
Press any key to continue . . .
```

```
public class TestMe
{
    static void Main(string[] args)
    {
        TestClass t1 = new TestClass();
        t1.x = 10;
        AddOne(t1);

        Console.WriteLine(t1.x );
    }

    public static void AddOne(TestClass a)
    {
        a.x++;
    }

    public struct TestClass
    {
        public int x;
    }
}
```

```
Select C:\WINDOWS\system32\cmd.exe
10
Press any key to continue . . .
```

Pass by reference/value types - EXTRA

- What will the following output? Why?

```
public class TestMe
{
    static void Main(string[] args)
    {
        TestClass t1 = new TestClass();
        t1.x = 10;
        TestClass t2 = new TestClass();
        t2.x = 20;

        Swap(t1, t2);
        Console.WriteLine(t1.x+" "+t2.x );
    }

    public static void Swap(TestClass a, TestClass b)
    {
        TestClass tmp = a;
        a = b;
        b = tmp;
    }

    public class TestClass
    {
        public int x;
    }
}
```

```
public class TestMe
{
    static void Main(string[] args)
    {
        TestClass t1 = new TestClass();
        t1.x = 10;
        TestClass t2 = new TestClass();
        t2.x = 20;

        Swap(ref t1, ref t2);
        Console.WriteLine(t1.x+" "+t2.x );
    }

    public static void Swap(ref TestClass a, ref TestClass b)
    {
        TestClass tmp = a;
        a = b;
        b = tmp;
    }

    public class TestClass
    {
        public int x;
    }
}
```

```
Select C:\WINDOWS\system32\cmd.exe
10 20
Press any key to continue . . .
```

```
Select C:\WINDOWS\system32\cmd.exe
20 10
Press any key to continue . . .
```

Pass by reference/value types - EXTRA

- What will the following output? Why?

```
public class TestMe
{
    static void Main(string[] args)
    {
        TestClass t1 = new TestClass();
        t1.x = 10;
        TestClass t2 = new TestClass();
        t2.x = 20;

        Swap(t1, t2);
        Console.WriteLine(t1.x+" "+t2.x );
    }

    public static void Swap(TestClass a, TestClass b)
    {
        TestClass tmp = a;
        a = b;
        b = tmp;
    }

    public class TestClass
    {
        public int x;
    }
}
```

```
Select C:\WINDOWS\system32\cmd.exe
10 20
Press any key to continue . . .
```

```
public class TestMe
{
    static void Main(string[] args)
    {
        TestClass t1 = new TestClass();
        t1.x = 10;
        TestClass t2 = new TestClass();
        t2.x = 20;

        Swap(t1, t2);
        Console.WriteLine(t1.x + " " + t2.x);
    }

    public static void Swap(TestClass a, TestClass b)
    {
        TestClass tmp = new TestClass();
        tmp.x = a.x;
        a.x = b.x;
        b.x = tmp.x;
    }

    public class TestClass
    {
        public int x;
    }
}
```

```
Select C:\WINDOWS\system32\cmd.exe
20 10
Press any key to continue . . .
```

Lesson 2: Organizing Data into Collections

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

- Note: in module 4 we'll see generic collections and standard collection interfaces (ICollection, IList, IDictionary, IEnumerable, etc).
  - In here we'll see non-generic collections

Choosing Collections

- When you create multiple items of the same type (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.
  - **You can reinvent the wheel** and write all the code from scratch or **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.

- *List* classes store linear collections of items.
  - think of a list class as a one-dimensional array that dynamically expands as you add items
- *Dictionary* classes store collections 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** (typically unique) is the object that you use to index and look up the value
  - E.g. key: recipe names, value: ingredients, and instructions
- *Queue* classes store items in a first in, first out (fifo) collection
  - E.g. use a queue class to process orders in a coffee shop
- *Stack* classes store items in a last in, first out (lifo) collection
  - For example, you might use a stack class to determine the 10 most recent visitors to your coffee shop.
  - Undo button …

Standard Collection Classes – Brief

*Don't spend too long on this topic. Rather than covering each class in detail, emphasize that the students should know when to use each class. Familiarize with each collection class in Visual Studio to prepare for the exam.

The **System.Collections** namespace provides a range of general-purpose collections that includes lists, dictionaries, queues, and stacks.

Class	Description
ArrayList	• General-purpose list collection • Linear collection of objects • methods and properties that enable you to add items, remove items, count the number of items in the collection, and sort the collection.
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 • retrieve items by key, add items, remove items, and check for particular keys and values
Queue	• First in, first out collection
SortedList	• Dictionary collection sorted by key • [Hashtable +] Retrieve items by index as well as by key
Stack	• Last in, first out collection

Specialized Collection Classes – Brief

*As with the previous topic, don't spend too long on this topic. Rather than covering each class in detail, emphasize that the students should know when to use each class.

The **System.Collections.Specialized** namespace provides collection classes that are suitable for more specialized requirements.

Class	Description
ListDictionary	• Dictionary collection • Optimized for small collections (≤10) (… >10 use Hashtable)
HybridDictionary	• Dictionary collection. Use when unsure about collection's size • Implemented as **ListDictionary** when small, changes to **Hashtable** as collection grows larger
OrderedDictionary	• Unsorted [by key] dictionary collection [vs SortedList …] • 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 [vs BitArray can expand …] • Represent values as Booleans or integers

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
  - 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.

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

- Use a foreach loop to iterate over the collection

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

Using Dictionary Collections

- Specify both a key and a value when you add an item

```
Hashtable ingredients = new Hashtable();
ingredients.Add("Cappuccino", "Coffee, Milk, Foam");
ingredients.Add("Café Mocha", "Coffee, Milk, Chocolate");
ingredients.Add("Macchiato", "Coffee, Milk, Foam");
```

- Retrieve items by key

```
if(ingredients.ContainsKey("Café Mocha"))
{
        string recipeMocha = ingredients["Café Mocha"];
}
```

- Iterate over key collection (or value collection)

```
foreach(string key in ingredients.Keys)
{
  Console.WriteLine(ingredients[key]);
}
```

Querying a Collection

- 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.
    - Standardized means that the syntax is the same regardless of the data source.
    - Declarative is a specific programming concept; it means a syntax that describes what you want to do, without explicitly describing how you want to do it. This contrasts with imperative programming, such as Visual C# code, in which you must provide specific algorithm implementation

- Use LINQ expressions to query collections
  - 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. Generic types and extension methods are covered later in this course
  - 1.69M (or 1.69m) ←m indicates that the number be treated as **decimal** type

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

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

```
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);
```

- Use extensions methods to retrieve specific items from results

```
decimal lowestPrice = drinks.FirstOrDefault(); //finds cheapest drink (since sorted)
decimal highestPrice = drinks.Last();          //finds most expensive drink
```

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

Events

- 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.

Handling Events ←by example

- The first thing you need to do is to define a delegate. It includes 2 parameters:
  - The first parameter is the object that raised the event – e.g. a Coffee instance.
  - The second parameter is any other information – must be an instance of **EventArgs** (or derived)
- Next, you need to define the event.
  - use the event keyword and precede the name of the event with the name of the delegate you want to associate with the event.
- 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:
    - Check whether the event is null. The event will be null if no code is currently subscribing to it.
    - Invoke the event and provide arguments to the delegate.
      - You provide arguments to match the parameters required by the delegate.
- To subscribe to the event from client code, you will 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.

Handling Events ←by example

- The
  - Th
  - Th
- Nex
  - us
    wa
- Afte
  raise
  - W
    m
  - To
    - You provide arguments to match the param
- To subscribe to the event from
  - Create a method with a signature that ma
  - Subscribe (+=) to the event by giving the
    words, the object that will raise the event
  - When the event is raised, the delegate in
    event.

It includes 2 parameters:

se, a Coffee instance.

tance of **EventArgs** (or derived)

he name of the delegate you

an write code that

nt will invoke any event handler

er.

er

```
public struct Coffee
{
    public EventArgs e;
    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);
            }
        }
    }
```

```
public class Inventory
{
    Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");

    public void HandleOutOfBeans(Coffee sender, EventArgs args)
    {
        string coffeeBean = sender.Bean;
        // Reorder the coffee bean.
        Console.WriteLine("Reorder Coffee Beans for: " + coffeeBean);
    }
    public void SubscribeToEvent()
    {
        coffee1.OutOfBeans += HandleOutOfBeans;
    }
```

Creating Events and Delegates

- 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

- Create a delegate for the event
  - A **delegate** is a special type that defines a method signature;
  - A **delegate** behaves like a representative for methods with matching signatures

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

- Create the event and specify the delegate
  - When you define an **event**, you associate a delegate with your event.

```
public event OutOfBeansHandler OutOfBeans;
```

Raising Events

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

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

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;
```

Module Review and Takeaways

- **Question:** 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?
  - ( )Option 1: public string CountryOfOrigin { get; set; }
  - ( )Option 2: public string CountryOfOrigin { get; }
  - ( )Option 3: public string CountryOfOrigin { set; }
  - ( )Option 4: public string CountryOfOrigin { get; private set; }
  - ( )Option 5: private string CountryOfOrigin { get; set; }
- **Question:** 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?
  - ( )Option 1: ArrayList
  - ( )Option 2: Hashtable
  - ( )Option 3: SortedList
  - ( )Option 4: NameValueCollection
  - ( )Option 5: StringDictionary

Module Review and Takeaways

- **Question:** 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?

  - ( )Option 1:
    ```
    public void HandleOutOfBeans(delegate OutOfBeansHandler)
    {
    }
    ```
  - ( )Option 2:
    ```
    public void HandleOutOfBeans(Coffee c, EventArgs e)
    {
    }
    ```
  - ( )Option 3:
    ```
    public Coffee HandleOutOfBeans(EventArgs e)
    ```
  - ( )Option 4:
    ```
    public Coffee HandleOutOfBeans(Coffee coffee, EventArgs args)
    ```
  - ( )Option 5:
    ```
    public void HandleOutOfBeans(Coffee c, EventArgs e)
    ```