

Module 1

Review of Visual C# Syntax

Module Overview

- Overview of Writing Application by Using Visual C#
- Data Types, Operators, and Expressions
- Visual C# Programming Language Constructs

Lesson 1: Overview of Writing Application by Using Visual C#

- What Is the .NET Framework?
- Key Features of Visual Studio 2017
- Templates in Visual Studio 2017
- Creating a .NET Framework Application
- Overview of XAML

What Is the .NET Framework?

- CLR
 - Robust and secure environment for your managed code
 - Memory management
 - Multithreading
- Class library
 - Foundation of common functionality
 - Extensible
- Development frameworks
 - WPF
 - Universal Windows Platform
 - ASP.NET

Key Features of Visual Studio 2017

- Intuitive IDE
- Rapid application development
- Server and data access
- IIS Express
- Debugging features
- Error handling
- Help and documentation

Templates in Visual Studio 2017

- Console Application
- WPF Application
- Universal Windows Platform (UWP)
- Class Library
- ASP.NET Web Application
- ASP.NET MVC 4 Application
- WCF Service Application

Creating a .NET Framework Application

1. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project dialog** box, choose a template, location, name, and then click **OK**.

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace ConsoleApplication1  
{  
    class Program  
    {  
        static void Main(string[] args) { }  
    }  
}
```

Overview of XAML

- XML-based language for declaring UIs
- Uses elements to define controls
- Uses attributes to define properties of controls

```
<Label Content="Name:" />
```

```
<TextBox Text="" Height="23" Width="120" />
```

```
<Button Content="Click Me!" Width="75" />
```


Lesson 2: Data Types, Operators, and Expressions

- What are Data Types?
- Expressions and Operators in Visual C#
- Declaring and Assigning Variables
- Accessing Type Members
- Casting Between Data Types
- Manipulating Strings

What are Data Types?

- int – whole numbers
- long – whole numbers (bigger range)
- float – floating-point numbers
- double - double precision
- decimal - monetary values
- char - single character
- bool - Boolean
- DateTime - moments in time
- string - sequence of characters

Expressions and Operators in Visual C#

Example expressions:

- + operator

```
a + 1
```

- / operator

```
5 / 2
```

- + and – operators

```
a + b - 2
```

- + operator (string concatenation)

```
"ApplicationName: " + appName.ToString()
```

Declaring and Assigning Variables

- Declaring variables:

```
int price;  
// OR  
int price, tax;
```

- Assigning variables:

```
price = 10;  
// OR  
int price = 10;
```

- Implicitly typed variables:

```
var price = 20;
```

- Instantiating object variables by using the **new** operator

```
ServiceConfiguration config = new ServiceConfiguration();
```

Accessing Type Members

- Invoke instance members

```
<instanceName>.<memberName>
```

- Example:

```
var config = new ServiceConfiguration();

// Invoke the LoadConfiguration method.
config.LoadConfiguration();

// Get the value from the ApplicationName property.
var applicationName = config.ApplicationName;

// Set the .DatabaseServerName property.
config.DatabaseServerName = "78.45.81.23";

// Invoke the SaveConfiguration method.
config.SaveConfiguration();
```

Casting Between Data Types

- Implicit conversion:

```
int a = 4;  
long b = 5;  
b = a;
```

- Explicit conversion:

```
int a = (int) b;
```

- **System.Convert** conversion:

```
string possibleInt = "1234";  
int count = Convert.ToInt32(possibleInt);
```

Manipulating Strings

- Concatenating strings

```
StringBuilder address = new StringBuilder();  
address.Append("23");  
address.Append(", Main Street");  
address.Append(", Buffalo");  
string concatenatedAddress = address.ToString();
```

- Validating strings

```
var textToTest = "hell0 w0rld";  
var regularExpression = "\\d";  
  
var result = Regex.IsMatch(textToTest, regularExpression,  
    RegexOptions.None);  
  
if (result)  
{  
    // Text matched expression.  
}
```

Lesson 3: Visual C# Programming Language Constructs

- Implementing Conditional Logic
- Implementing Iteration Logic
- Creating and Using Arrays
- Referencing Namespaces
- Using Breakpoints in Visual Studio 2017
- Demonstration: Developing the Class Enrollment Application Lab

Implementing Conditional Logic

- **if** statements

```
if (response == "connection_failed") { . . . }  
else if (response == "connection_error") { . . . }  
else { }
```

- **select** statements

```
switch (response)  
{  
  case "connection_failed":  
    . . .  
    break;  
  case "connection_success":  
    . . .  
    break;  
  default:  
    . . .  
    break;  
}
```

Implementing Iteration Logic

- **for** loop

```
for (int i = 0 ; i < 10; i++) { ... }
```

- **foreach** loop

```
string[] names = new string[10];  
foreach (string name in names) { ... }
```

- **while** loop

```
bool dataToEnter = CheckIfUserWantsToEnterData();  
while (dataToEnter)  
{  
    ...  
    dataToEnter = CheckIfUserHasMoreData();  
}
```

- **do** loop

```
do  
{  
    ...  
    moreDataToEnter = CheckIfUserHasMoreData();  
} while (moreDataToEnter);
```

Creating and Using Arrays

- C# supports:
 - Single-dimensional arrays
 - Multidimensional arrays
 - Jagged arrays
- Creating an array:

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

- Accessing data in an array:
 - By index

```
int result = arrayName[2];
```

- In a loop

```
for (int i = 0; i < arrayName.Length; i++)  
{  
    int result = arrayName[i];  
}
```

Referencing Namespaces

- Use namespaces to organize classes into a logically related hierarchy
- .NET class library includes:
 - **System.Windows**
 - **System.Data**
 - **System.Web**
- Define your own namespaces:

```
namespace FourthCoffee.Console
{
    class Program { . . . }
```

- Use namespaces:
 - Add reference to containing library
 - Add **using** directive to code file

Using Breakpoints in Visual Studio 2017

- Breakpoints enable you to view and modify the contents of variables:
 - Immediate Window
 - Autos, Locals, and Watch panes
- Debug menu and toolbar functions enable you to:
 - Start and stop debugging
 - Enter break mode
 - Restart the application
 - Step through code

Demonstration: Developing the Class Enrollment Application Lab

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

Lab: Developing the Class Enrollment Application

- Exercise 1: Implementing Edit Functionality for the Students List
- Exercise 2: Implementing Insert Functionality for the Students List
- Exercise 3: Implementing Delete Functionality for the Students List
- Exercise 4: Displaying a Student's Age

Estimated Time: 105 minutes

Lab Scenario

You are a Visual C# developer working for a software development company that is writing applications for The School of Fine Arts, an elementary school for gifted children.

The school administrators require an application that they can use to enroll students in a class. The application must enable an administrator to add and remove students from classes, as well as to update the details of students.

You have been asked to write the code that implements the business logic for the application.

During the labs for the first two modules in this course, you will write code for this class enrollment application.

When The School of Fine Arts ask you to extend the application functionality, you realize that you will need to test proof of concept and obtain client feedback before writing the final application, so in the lab for Module 3, you will begin developing a prototype application and continue with this until the end of Module 8.

In the lab for Module 9, after gaining signoff for the final application, you will develop the user interface for the production version of the application, which you will work on for the remainder of the course.

Module Review and Takeaways

- Review Questions

Module 2

Creating Methods, Handling
Exceptions, and Monitoring
Applications

Module Overview

- Creating and Invoking Methods
- Creating Overloaded Methods and Using Optional and Output Parameters
- Handling Exceptions
- Monitoring Applications

Lesson 1: Creating and Invoking Methods

- What Is a Method?
- Creating Methods
- Invoking Methods
- Debugging Methods
- Demonstration: Creating, Invoking, and Debugging Methods

What Is a Method?

- Methods encapsulate operations that protect data
- .NET Framework applications contain a **Main** entry point method
- The .NET Framework provides many methods in the base class library

Creating Methods

- Methods comprise two elements:
 - Method specification (return type, name, parameters)
 - Method body
- Use the **ref** keyword to pass parameter references

```
void StartService(int upTime, bool shutdownAutomatically)
{
    // Perform some processing here.
}
```

- Use the **return** keyword to return a value from the method

```
string GetServiceName()
{
    return "FourthCoffee.SalesService";
}
```

Invoking Methods

To call a method specify:

- Method name
- Any arguments to satisfy parameters

```
var upTime = 2000;  
var shutdownAutomatically = true;  
StartService(upTime, shutdownAutomatically);  
  
// StartService method.  
void StartService(int upTime, bool shutdownAutomatically)  
{  
    // Perform some processing here.  
}
```

Debugging Methods

- Visual Studio provides debug tools that enable you to step through code
- When debugging methods you can:
 - Step into the method
 - Step over the method
 - Step out of the method

Demonstration: Creating, Invoking, and Debugging Methods

In this demonstration, you will create a method, invoke the method, and then debug the method

Lesson 2: Creating Overloaded Methods and Using Optional and Output Parameters

- Creating Overloaded Methods
- Creating Methods that Use Optional Parameters
- Calling a Method by Using Named Arguments
- Creating Methods that Use Output Parameters

Creating Overloaded Methods

- Overloaded methods share the same method name
- Overloaded methods have a unique signature

```
void StopService()  
{  
    ...  
}  
  
void StopService(string serviceName)  
{  
    ...  
}  
  
void StopService(int serviceId)  
{  
    ...  
}
```

Creating Methods that Use Optional Parameters

- Define all mandatory parameters first

```
void StopService(  
    bool forceStop,  
    string serviceName = null,  
    int serviceId = 1)  
{  
    ...  
}
```

- Satisfy parameters in sequence

```
var forceStop = true;  
StopService(forceStop);
```

// OR

```
var forceStop = true;  
var serviceName = "FourthCoffee.SalesService";  
StopService(forceStop, serviceName);
```

Calling a Method by Using Named Arguments

- Specify parameters by name
- Supply arguments in a sequence that differs from the method's signature
- Supply the parameter name and corresponding value separated by a colon

```
StopService(true, serviceID: 1);
```

Creating Methods that Use Output Parameters

- Use the **out** keyword to define an output parameter

```
bool IsServiceOnline(string serviceName, out string statusMessage)
{
    ...
}
```

- Provide a variable for the corresponding argument when you call the method

```
var statusMessage = string.Empty;
var isServiceOnline = IsServiceOnline(
    "FourthCoffee.SalesService",
    out statusMessage);
```

Lesson 3: Handling Exceptions

- What Is an Exception?
- Handling Exception by Using a Try/Catch Block
- Using a Finally Block
- Throwing Exceptions

What Is an Exception?

- An exception is an indication of an error or exceptional condition
- The .NET Framework provides many exception classes:
 - **Exception**
 - **SystemException**
 - **ApplicationException**
 - **NullReferenceException**
 - **FileNotFoundException**
 - **SerializationException**

Handling Exception by Using a Try/Catch Block

- Use try/catch blocks to handle exceptions
- Use one or more catch blocks to catch different types of exceptions

```
try
{
}
catch (NullReferenceException ex)
{
    // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
    // Catch all other exceptions.
}
```

Using a Finally Block

Use a finally block to run code whether or not an exception has occurred

```
try
{
}
catch (NullReferenceException ex)
{
    // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
    // Catch all other exceptions.
}
finally
{
    // Code that always runs.
}
```

Throwing Exceptions

- Use the **throw** keyword to throw a new exception

```
var ex =  
    new NullReferenceException("The 'Name' parameter is null.");  
throw ex;
```

- Use the **throw** keyword to rethrow an existing exception

```
try  
{  
}  
catch (NullReferenceException ex)  
{  
}  
catch (Exception ex)  
{  
    ...  
    throw;  
}
```

Lesson 4: Monitoring Applications

- Using Logging and Tracing
- Using Application Profiling
- Using Performance Counters
- Demonstration: Extending the Class Enrollment Application Functionality Lab

Using Logging and Tracing

- *Logging* provides information to users and administrators
 - Windows event log
 - Text files
 - Custom logging destinations
- *Tracing* provides information to developers
 - Visual Studio Output window
 - Custom tracing destinations

Using Application Profiling

- Create and run a *performance session*
- Analyze the *profiling report*
- Revise your code and repeat

Using Performance Counters

- Create performance counters and categories in code or in Server Explorer
- Specify:
 - A name
 - Some help text
 - The base performance counter type
- Update custom performance counters in code
- View performance counters in Performance Monitor (perfmon.exe)

Demonstration: Extending the Class Enrollment Application Functionality Lab

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

Lab: Extending the Class Enrollment Application Functionality

- Exercise 1: Refactoring the Enrollment Code
- Exercise 2: Validating Student Information
- Exercise 3: Saving Changes to the Class List

Estimated Time: 90 minutes

Lab Scenario

You have been asked to refactor the code that you wrote in the lab exercises for module 1 into separate methods to avoid the duplication of code in the Class Enrollment Application.

Also, you have been asked to write code that validates the student information that the user enters and to enable the updated student information to be written back to the database, handling any errors that may occur.

Module Review and Takeaways

- Review Questions

Module 3

Basic types and constructs of Visual
C#

Module Overview

- 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;
```

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();
```


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

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

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];
```

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

Lesson 2: Organizing Data into Collections

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

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

Standard Collection Classes

Class	Description
ArrayList	<ul style="list-style-type: none">• General-purpose list collection• Linear collection of objects
BitArray	<ul style="list-style-type: none">• Collection of Boolean values• Useful for bitwise operations and Boolean arithmetic (for example, AND, NOT, and XOR)
Hashtable	<ul style="list-style-type: none">• General-purpose dictionary collection• Stores key/value object pairs
Queue	<ul style="list-style-type: none">• First in, first out collection
SortedList	<ul style="list-style-type: none">• Dictionary collection sorted by key• Retrieve items by index as well as by key
Stack	<ul style="list-style-type: none">• Last in, first out collection

Specialized Collection Classes

Class	Description
ListDictionary	<ul style="list-style-type: none">• Dictionary collection• Optimized for small collections (<10)
HybridDictionary	<ul style="list-style-type: none">• Dictionary collection• Implemented as ListDictionary when small, changes to Hashtable as collection grows larger
OrderedDictionary	<ul style="list-style-type: none">• Unsorted dictionary collection• Retrieve items by index as well as by key
NameValueCollection	<ul style="list-style-type: none">• Dictionary collection in which both keys and values are strings• Retrieve items by index as well as by key
StringCollection	<ul style="list-style-type: none">• List collection in which all items are strings
StringDictionary	<ul style="list-style-type: none">• Dictionary collection in which both keys and values are strings
BitVector32	<ul style="list-style-type: none">• Fixed size 32-bit structure• 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

```
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("Café Mocha", "Coffee, Milk, Chocolate");
```

- Retrieve items by key

```
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

- 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();
```

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

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;
```

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;
```

Demonstration: Writing Code for the Grades Prototype Application Lab

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

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

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

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.

Module Review and Takeaways

- Review Questions

Module 4

Creating Classes and Implementing
Type-Safe Collections

Module Overview

- Creating Classes
- Defining and Implementing Interfaces
- Implementing Type-Safe Collections

Lesson 1: Creating Classes

- Creating Classes and Members
- Instantiating Classes
- Using Constructors
- Reference Types and Value Types
- Demonstration: Comparing Reference Types and Value Types
- Creating Static Classes and Members
- Testing Classes

Creating Classes and Members

- Use the **class** keyword

```
public class DrinksMachine
{
    // Methods, fields, properties, and events.
}
```

- Specify an access modifier:
 - public
 - internal
 - private
- Add methods, fields, properties, and events

Instantiating Classes

- To instantiate a class, use the **new** keyword

```
DrinksMachine dm = new DrinksMachine();
```

- To infer the type of the new object, use the **var** keyword

```
var dm = new DrinksMachine();
```

- To call members on the instance, use the dot notation

```
dm.Model = "BeanCrusher 3000";  
dm.Age = 2;  
dm.MakeEspresso();
```

Using Constructors

- Constructors are a type of method:
 - Share the name of the class
 - Called when you instantiate a class
- A default constructor accepts no arguments

```
public class DrinksMachine
{
    public void DrinksMachine()
    {
        // This is a default constructor.
    }
}
```

- Classes can include multiple constructors
- Use constructors to initialize member variables

Reference Types and Value Types

- Value types

- Contain data directly

```
int First = 100;  
int Second = First;
```

- In this case, **First** and **Second** are two distinct items in memory

- Reference types

- Point to an object in memory

```
object First = new Object();  
object Second = First;
```

- In this case, **First** and **Second** point to the same item in memory

Demonstration: Comparing Reference Types and Value Types

In this demonstration, you will learn how to:

- Create a value type to store an integer value
- Create a reference type to store an integer value
- Observe the differences in behavior when you copy the value type and the reference type

Creating Static Classes and Members

- Use the static keyword to create a static class

```
public static class Conversions
{
    // Static members go here.
}
```

- Call members directly on the class name

```
double weightInKilos = 80;
double weightInPounds =
    Conversions.KilosToPounds(weightInKilos);
```

- Add static members to non-static classes

Testing Classes

Arrange

- Create the conditions for the test
- Configure any input values required

Act

- Invoke the action that you want to test

Assert

- Verify the results of the action
- Fail the test if the results were not as expected

Lesson 2: Defining and Implementing Interfaces

- Introducing Interfaces
- Defining Interfaces
- Implementing Interfaces
- Implementing Multiple Interfaces
- Implementing the IComparable Interface
- Implementing the IComparer Interface

Introducing Interfaces

- Interfaces define a set of characteristics and behaviors
 - Member signatures only
 - No implementation details
 - Cannot be instantiated
- Interfaces are implemented by classes or structs
 - Implementing class or struct must implement every member
 - Implementation details do not matter to consumers
 - Member signatures must match definitions in interface
- By implementing an interface, a class or struct guarantees that it will provide certain functionality

Defining Interfaces

- Use the **interface** keyword

```
public interface IBeverage
{
    // Methods, properties, events, and indexers.
}
```

- Specify an access modifier:
 - public
 - internal
- Add interface members:
 - Methods, properties, events, and indexers
 - Signatures only, no implementation details

Implementing Interfaces

- Add the name of the interface to the class declaration

```
public class Coffee : IBeverage
```

- Implement all interface members
- Use the interface type and the derived class type interchangeably

```
Coffee coffee1 = new Coffee();  
IBeverage coffee2 = new Coffee();
```

The **coffee2** variable will only expose members defined by the **IBeverage** interface

Implementing Multiple Interfaces

- Add the names of each interface to the class declaration

```
public class Coffee : IBeverage, IInventoryItem
```

- Implement every member of every interface
- Use explicit implementation if two interfaces have a member with the same name

```
// This is an implicit implementation.  
public bool IsFairTrade { get; set; }
```

```
// These are explicit implementations.  
public bool IInventoryItem.IsFairTrade { get; }  
public bool IBeverage.IsFairTrade { get; set; }
```

Implementing the Comparable Interface

- If you want instances of your class to be sortable in collections, implement the **Comparable** interface

```
public interface Comparable
{
    int CompareTo(Object obj);
}
```

- The **ArrayList.Sort** method calls the **Comparable.CompareTo** method on collection members to sort items in a collection

Implementing the IComparer Interface

- To sort collections by custom criteria, implement the **IComparer** interface

```
public interface IComparer
{
    int Compare(Object x, Object y);
}
```

- To use an **IComparer** implementation to sort an **ArrayList**, pass an **IComparer** instance to the **ArrayList.Sort** method

```
ArrayList coffeeList = new ArrayList();
// Add some items to the collection.
coffeeList.Sort(new CoffeeRatingComparer());
```

Lesson 3: Implementing Type-Safe Collections

- Introducing Generics
- Advantages of Generics
- Constraining Generics
- Using Generic List Collections
- Using Generic Dictionary Collections
- Using Collection Interfaces
- Creating Enumerable Collections
- Demonstration: Adding Data Validation and Type-Safety to the Application Lab

Introducing Generics

- Create classes and interfaces that include a type parameter

```
public class CustomList<T>
{
    public T this[int index] { get; set; }
    public void Add(T item) { ... }
    public void Remove(T item) { ... }
}
```

- Specify the type argument when you instantiate the class

```
CustomList<Coffee> coffees =
    new CustomList<Coffee>();
```

Advantages of Generics

Generic types offer three advantages over non-generic types:

- Type safety
- No casting
- No boxing and unboxing

Constraining Generics

You can constrain type parameters in six ways:

- where T : <name of interface>
- where T : <name of base class>
- where T : U
- where T : new()
- where T : struct
- where T : class

Using Generic List Collections

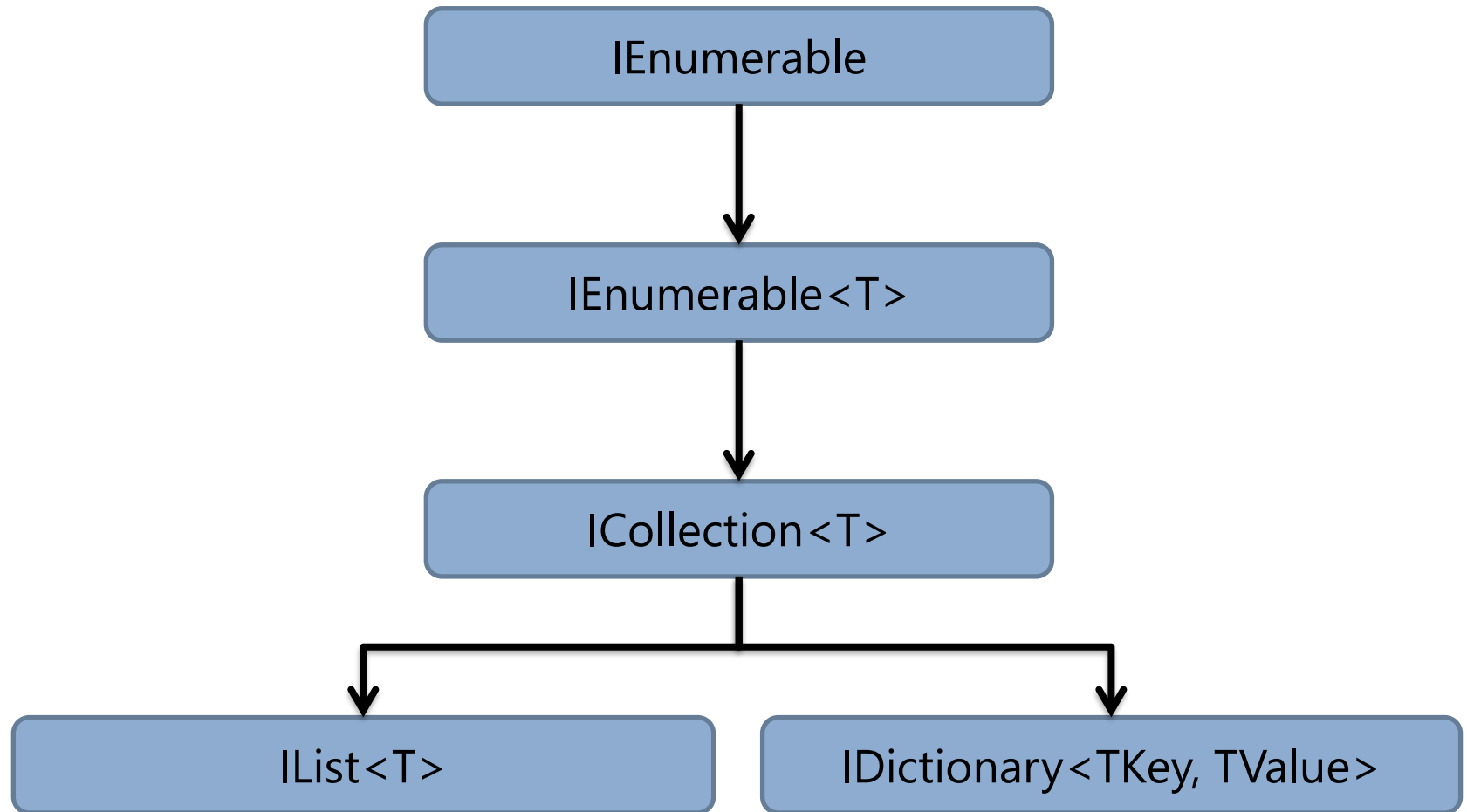
Generic list classes store collections of objects of type **T**:

- **List<T>** is a general purpose generic list
- **LinkedList<T>** is a generic list in which each item is linked to the previous item and the next item in the collection
- **Stack<T>** is a last in, first out collection
- **Queue<T>** is a first in, first out collection

Using Generic Dictionary Collections

- Generic dictionary classes store key-value pairs
- Both the key and the value are strongly typed
- **Dictionary<TKey, TValue>** is a general purpose, generic dictionary class
- **SortedList<TKey, TValue>** and **SortedDictionary<TKey, TValue>** collections are sorted by key

Using Collection Interfaces



Creating Enumerable Collections

- Implement **IEnumerable<T>** to support enumeration (**foreach**)
- Implement the **GetEnumerator** method by either:
 - Creating an **IEnumerator<T>** implementation
 - Using an iterator
- Use the **yield return** statement to implement an iterator

Demonstration: Adding Data Validation and Type-Safety to the Application Lab

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

Lab: Adding Data Validation and Type-Safety to the Application

- Exercise 1: Implementing the Teacher, Student, and Grade Structs as Classes
- Exercise 2: Adding Data Validation to the Grade Class
- Exercise 3: Displaying Students in Name Order
- Exercise 4: Enabling Teachers to Modify Class and Grade Data

Estimated Time: 75 minutes

Lab Scenario

Now that the user interface navigation features are working, you decide to replace the simple structs with classes to make your application more efficient and straightforward.

You have also been asked to include validation logic in the application to ensure that when a user adds grades to a student, that the data is valid before it is written to the database. You decide to create a unit test project that will perform tests against the required validation for different grade scenarios.

Teachers who have seen the application have expressed concern that the students in their classes are displayed in a random order. You decide to use the `Comparable` interface to enable them to be displayed in alphabetical order.

Finally, you have been asked to add functionality to the application to enable teachers to add students to and remove students from a class, and to add student grades to the database.

Module Review and Takeaways

- Review Questions

Module 5

Creating a Class Hierarchy by Using Inheritance

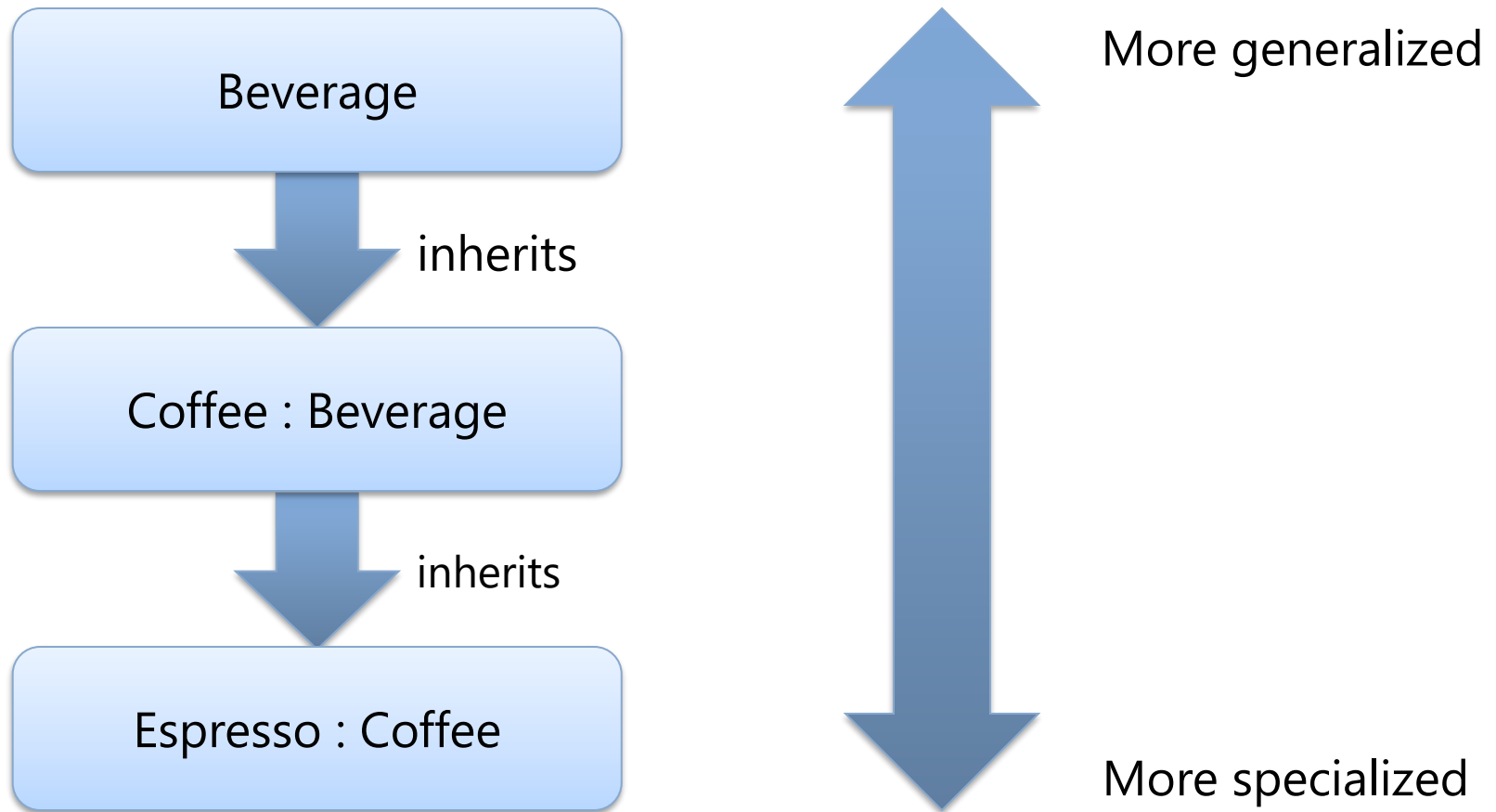
Module Overview

- Creating Class Hierarchies
- Extending .NET Framework Classes

Lesson 1: Creating Class Hierarchies

- What Is Inheritance?
- Creating Base Classes
- Creating Base Class Members
- Inheriting from a Base Class
- Calling Base Class Constructors and Members
- Demonstration: Calling Base Class Constructors

What Is Inheritance?



The diagram shows a class hierarchy where a class named **Espresso** inherits from a class named **Coffee**, which in turn inherits from a class named **Beverage**. The inherited classes are increasingly specialized instances of the base class.

Creating Base Classes

- Use the **abstract** keyword to create a base class that cannot be instantiated

```
public abstract class Beverage
```

- Create a class that derives from the abstract class
 - Implement any abstract members
- Use the **sealed** keyword to create a class that cannot be inherited

```
public sealed class Tea : Beverage
```

Creating Base Class Members

- Use the **virtual** keyword to create members that you can override in derived classes

```
public virtual int GetServingTemperature()
```

- Use the **protected** access modifier to make members available to derived types

```
protected int servingTemperature;
```

Inheriting from a Base Class

- To inherit from a base class, add the name of the base class to the class declaration

```
public class Coffee : Beverage
```

- To override virtual base class members, use the **override** keyword

```
public override int GetServingTemperature()
```

- To prevent classes further down the class hierarchy from overriding your override methods, use the **sealed** keyword

```
sealed public override int GetServingTemperature()
```


Calling Base Class Constructors and Members

- To call a base class constructor from a derived class, add the base constructor to your constructor declaration

```
public Coffee(string name, bool isFairTrade, int temp)  
    : base(name, isFairTrade, servingTemp)
```

- Pass parameter names to the base constructor as arguments
- Do not use the base keyword within the constructor body
- To call base class methods from a derived class, use the base keyword like an instance variable

```
base.GetServingTemperature();
```

Demonstration: Calling Base Class Constructors

In this demonstration, you will learn how to:

- Create a derived class constructor that calls a default base class constructor implicitly
- Create a derived class constructor that calls a specific base class constructor explicitly
- Observe the order of constructor execution as a derived class is instantiated

Lesson 2: Extending .NET Framework Classes

- Inheriting from .NET Framework Classes
- Creating Custom Exceptions
- Throwing and Catching Custom Exceptions
- Inheriting from Generic Types
- Creating Extension Methods
- Demonstration: Refactoring Common Functionality into the User Class Lab

Inheriting from .NET Framework Classes

- Inherit from .NET Framework classes to:
 - Reduce development time
 - Standardize functionality
- Inherit from any .NET Framework type that is not **sealed** or **static**
- Override any base class members that are marked as **virtual**
- Implement any base class members that are marked as **abstract**

Creating Custom Exceptions

To create a custom exception type:

1. Inherit from the **System.Exception** class
2. Implement three standard constructors:
 - base()
 - base(string message)
 - base(string message, Exception inner)
3. Add additional members if required

Throwing and Catching Custom Exceptions

- Use the **throw** keyword to throw a custom exception

```
throw new LoyaltyCardNotFoundException();
```

- Use a try/catch block to catch the exception

```
try
{
    // Perform the operation that could cause the exception.
}
catch(LoyaltyCardNotFoundException ex)
{
    // Use the exception variable, ex, to get more information.
}
```

Inheriting from Generic Types

For each base type parameter, you must either:

- Provide a type argument in your class declaration

```
public class CustomList : List<int>
```

- Include a matching type parameter in your class declaration

```
public class CustomList<T> : List<T>
```

Creating Extension Methods

- Create a static method in a static class
- Use the first parameter to indicate the type you want to extend
- Precede the first parameter with the **this** keyword

```
public static bool ContainsNumbers(this string s) {...}
```

- Call the method like a regular instance method

```
string text = "Text with numb3r5 ";  
if(text.ContainsNumbers)  
{  
    // Do something.  
}
```


Demonstration: Refactoring Common Functionality into the User Class Lab

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

Lab: Refactoring Common Functionality into the User Class

- Exercise 1: Creating and Inheriting from the User Base Class
- Exercise 2: Implementing Password Complexity by Using an Abstract Method
- Exercise 3: Creating the ClassFullException Custom Exception

Estimated Time: 60 minutes

Lab Scenario

You have noticed that the Student and Teacher classes in the Grades application contain some duplicated functionality. To make the application more maintainable, you decide to refactor this common functionality to remove the duplication.

You are also concerned about security. Teachers and students all require a password, but it is important to maintain confidentiality and at the same time ensure that students (who are children) do not have to remember long and complex passwords. You decide to implement different password policies for teachers and students; teachers' passwords must be stronger and more difficult to guess than student passwords.

Also, you have been asked to update the application to limit the number of students that can be added to a class. You decide to add code that throws a custom exception if a user tries to enroll a student in a class that is already at capacity.

Module Review and Takeaways

- Review Questions

Module 6

Reading and Writing Local Data

Module Overview

- Reading and Writing Files
- Serializing and Deserializing Data
- Performing I/O by Using Streams

Lesson 1: Reading and Writing Files

- Reading and Writing Data by Using the File Class
- Manipulating Files
- Manipulating Directories
- Manipulating File and Directory Paths
- Demonstration: Manipulating Files, Directories, and Paths

Reading and Writing Data by Using the File Class

- The **System.IO namespace** contains classes for manipulating files and directories
- The **File** class contains atomic read methods, including:
 - **ReadAllText(...)**
 - **ReadAllLines(...)**
- The **File** class contains atomic write methods, including:
 - **WriteAllText(...)**
 - **AppendAllText(...)**

Manipulating Files

- The **File** class provides static members

```
File.Delete(...);  
bool exists = File.Exists(...);  
DateTime createdOn = File.GetCreationTime(...);
```

- The **FileInfo** class provides instance members

```
FileInfo file = new FileInfo(...);  
...  
string name = file.DirectoryName;  
bool exists = file.Exists;  
file.Delete();
```

Manipulating Directories

- The **Directory** class provides static members

```
Directory.Delete(...);  
bool exists = Directory.Exists(...);  
string[] files = Directory.GetFiles(...);
```

- The **DirectoryInfo** class provides instance members

```
DirectoryInfo directory = new DirectoryInfo(...);  
...  
string path = directory.FullName;  
bool exists = directory.Exists;  
FileInfo[] files = directory.GetFiles();
```

Manipulating File and Directory Paths

The **Path** class encapsulates file system utility functions

```
string settingsPath = "..could be anything here..";

// Check to see if path has an extension.
bool hasExtension = Path.HasExtension(settingsPath);

...

// Get the extension from the path.
string pathExt = Path.GetExtension(settingsPath);

...

// Get path to temp file.
string tempPath = Path.GetTempFileName();
// Returns C:\Users\LeonidsP\AppData\Local\Temp\ABC.tmp
```

Demonstration: Manipulating Files, Directories, and Paths

In this demonstration, you will use the **File**, **Directory**, and **Path** classes to build a utility that combines multiple files into a single file

Lesson 2: Serializing and Deserializing Data

- What Is Serialization?
- Creating a Serializable Type
- Serializing Objects as Binary
- Serializing Objects as XML
- Serializing Objects as JSON
- Serializing Objects as JSON by Using JSON.Net
- Demonstration: Serializing Objects as JSON using JSON.Net
- Creating a Custom Serializer

What Is Serialization?

- Binary

```
1010101010101111101011010101011010111111101010110110001
```

- XML

```
<SOAP-ENV:Envelope ...>  
  <SOAP-ENV:Body>  
    ...  
  </SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

- JSON

```
{  
  "ConfigName":"FourthCoffee_Default",  
  "DatabaseHostName":"database209.fourthcoffee.com"  
}
```

Creating a Serializable Type

Implement the **ISerializable** interface

```
[Serializable]
public class ServiceConfiguration : ISerializable
{
    ...
    public ServiceConfiguration(
        SerializationInfo info, StreamingContext ctxt)
    {
        ...
    }

    public void GetObjectData(
        SerializationInfo info, StreamingContext context)
    {
        ...
    }
}
```

Serializing Objects as Binary

- Serialize as binary

```
ServiceConfiguration config = ServiceConfiguration.Default;  
IFormatter formatter = new BinaryFormatter();  
FileStream buffer = File.Create("C:\\fourthcoffee\\config.txt");  
formatter.Serialize(buffer, config);  
buffer.Close();
```

- Deserialize from binary

```
IFormatter formatter = new BinaryFormatter();  
FileStream buffer = File.OpenRead("C:\\fourthcoffee\\config.txt");  
ServiceConfiguration config  
    = formatter.Deserialize(buffer) as ServiceConfiguration;  
buffer.Close();
```


Serializing Objects as XML

- Serialize as XML

```
ServiceConfiguration config = ServiceConfiguration.Default;  
IFormatter formatter = new SoapFormatter();  
FileStream buffer = File.Create(@"C:\fourthcoffee\config.xml");  
formatter.Serialize(buffer, config);  
buffer.Close();
```

- Deserialize from XML

```
IFormatter formatter = new SoapFormatter();  
FileStream buffer = File.OpenRead(@"C:\fourthcoffee\config.xml");  
ServiceConfiguration config  
    = formatter.Deserialize(buffer) as ServiceConfiguration;  
buffer.Close();
```

Serializing Objects as JSON

- Serialize as JSON

```
ServiceConfiguration config = ServiceConfiguration.Default;  
DataContractJsonSerializer jsonSerializer  
    = new DataContractJsonSerializer(config.GetType());  
FileStream buffer = File.Create(@"C:\fourthcoffee\config.txt");  
jsonSerializer.WriteObject(buffer, config);  
buffer.Close();
```

- Deserialize from JSON

```
DataContractJsonSerializer jsonSerializer = new  
    DataContractJsonSerializer(  
        typeof(ServiceConfiguration));  
FileStream buffer = File.OpenRead(@"C:\fourthcoffee\config.txt");  
ServiceConfiguration config = jsonSerializer.ReadObject(buffer)  
    as ServiceConfiguration;  
buffer.Close();
```

Serializing Objects as JSON by Using JSON.Net

- Serialize as JSON

```
// Create the object you want to serialize.  
ServiceConfiguration config = ServiceConfiguration.Default;  
  
// Serialize the object to a string  
var jsonString = JsonConvert.Serialize(config);
```

- Deserialize from JSON

```
// Deserialize to the desired type  
var deserializedConfig =  
    JsonConvert.DeserializeObject<ServiceConfiguration>(jsonString);
```

Demonstration: Serializing Objects as JSON using JSON.Net

In this demonstration you will see how to serialize and deserialize objects using JSON.NET

Creating a Custom Serializer

Implement the **IFormatter** interface

```
class IniFormatter : IFormatter
{
    public ISurrogateSelector SurrogateSelector { get; set; }
    public SerializationBinder Binder { get; set; }
    public StreamingContext Context { get; set; }

    public object Deserialize(Stream serializationStream)
    {
        ...
    }

    public void Serialize(Stream serializationStream, object graph)
    {
        ...
    }
}
```

Lesson 3: Performing I/O by Using Streams

- What are Streams?
- Types of Streams in the .NET Framework
- Reading and Writing Binary Data by Using Streams
- Reading and Writing Text Data by Using Streams
- Demonstration: Generating the Grades Report Lab

What are Streams?

- The **System.IO namespace** contains a number of stream classes, including:
 - The abstract **Stream** base class
 - The **FileStream** class
 - The **MemoryStream** class
- Typical stream operations include:
 - Reading chunks of data from a stream
 - Writing chunks of data to a stream
 - Querying the position of the stream

Types of Streams in the .NET Framework

- Classes that enable access to data sources include:

Class	Description
FileStream	Exposes a stream to a file on the file system.
MemoryStream	Exposes a stream to a memory location.
NetworkStream	Exposes a stream to a network location.

- Classes that enable reading and writing to and from data source streams include:

Class	Description
StreamReader	Read textual data from a source stream.
StreamWriter	Write textual data to a source stream.
BinaryReader	Read binary data from a source stream.
BinaryWriter	Write binary data to a source stream.

Reading and Writing Binary Data by Using Streams

You can use the **BinaryReader** and **BinaryWriter** classes to stream binary data

```
string filePath = "C:\\fourthcoffee\\applicationdata\\settings.txt";

// Underlying stream to file on the file system.
FileStream file = new FileStream(filePath);

// BinaryReader object exposes read operations on the underlying
// FileStream object.
BinaryReader reader = new BinaryReader(file);

// BinaryWriter object exposes write operations on the underlying
// FileStream object.
BinaryWriter writer = new BinaryWriter(file);
```

Reading and Writing Text Data by Using Streams

You can use the **StreamReader** and **StreamWriter** classes to stream plain text

```
string filePath = "C:\\fourthcoffee\\applicationdata\\settings.txt";

// Underlying stream to file on the file system.
FileStream file = new FileStream(filePath);

// StreamReader object exposes read operations on the underlying
// FileStream object.
StreamReader reader = new StreamReader(file);

// StreamWriter object exposes write operations on the underlying
// FileStream object.
StreamWriter writer = new StreamWriter(file);
```

Demonstration: Generating the Grades Report Lab

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

Lab: Generating the Grades Report

- Exercise 1: Serializing Data for the Grades Report as XML
- Exercise 2: Previewing the Grades Report
- Exercise 3: Persisting the Serialized Grade Data to a File

Estimated Time: 60 minutes

Lab Scenario

You have been asked to upgrade the Grades Prototype application to enable users to save a student's grades as an XML file on the local disk. The user should be able to click a new button on the StudentProfile view that asks the user where they would like to save the file, displays a preview of the data to the user, and asks the user to confirm that they wish to save the file to disk. If they do, the application should save the grade data in XML format in the location that the user specified.

Module Review and Takeaways

- Review Questions

Module 7

Accessing a Database

Module Overview

- Creating and Using Entity Data Models
- Querying Data by Using LINQ

Lesson 1: Creating and Using Entity Data Models

- Introduction to the ADO.NET Entity Framework
- Using the ADO.NET Entity Data Model Tools
- Demonstration: Creating an Entity Data Model
- Customizing Generated Classes
- Reading and Modifying Data by Using the Entity Framework
- Demonstration: Reading and Modifying Data in an EDM

Introduction to the ADO.NET Entity Framework

- The ADO.NET Entity Framework provides:
 - EDMs
 - Entity SQL
 - Object Services
- The ADO.NET Entity Framework supports:
 - Writing code against a conceptual model
 - Easy updating of applications to a different data source
 - Writing code that is independent from the storage system
 - Writing data access code that supports compile-time type-checking and syntax-checking

Using the ADO.NET Entity Data Model Tools

- Tools support:
 - Database-first design by using the Entity Data Model Wizard
 - Code-first design by using the Generate Database Wizard
- They also provide:
 - Designer pane for viewing, updating, and deleting entities and their relationships
 - Update Model Wizard for updating a model with changes that are made to the data source
 - Mapping Details pane for viewing, updating, and deleting mappings

Demonstration: Creating an Entity Data Model

In this demonstration, you will use the Entity Data Wizard to generate an EDM for an existing database

Customizing Generated Classes

- Do not modify the automatically generated classes in a model
- Use partial classes and partial methods to add business functionality to the generated classes

```
public partial class Employee
{
    partial void OnDateOfBirthChanging(DateTime? value)
    {
        if (GetAge() < 16)
        {
            throw new Exception("Employees must be 16 or over");
        }
    }
}
```

Reading and Modifying Data by Using the Entity Framework

- Reading data

```
FourthCoffeeEntities DBContext = new FourthCoffeeEntities();

// Print a list of employees.
foreach (FourthCoffee.Employees.Employee emp in
    DBContext.Employees)
{
    Console.WriteLine("{0} {1}", emp.FirstName, emp.LastName);
}
```

- Modifying data

```
var emp = DBContext.Employees.First(e => e.LastName ==
    "Prescott");
if (emp != null)
{
    emp.LastName = "Forsyth";
    DBContext.SaveChanges();
}
```

Demonstration: Reading and Modifying Data in an EDM

In this demonstration, you will use the **ObjectSet(TEntity)** class to read and modify data in an EDM

Lesson 2: Querying Data by Using LINQ

- Querying Data
- Demonstration: Querying Data
- Querying Data by Using Anonymous Types
- Demonstration: Querying Data by Using Anonymous Types
- Forcing Query Execution
- Demonstration: Retrieving and Modifying Grade Data Lab

Querying Data

- Use LINQ to query a range of data sources, including:
 - .NET Framework collections
 - SQL Server databases
 - ADO.NET data sets
 - XML documents
- Use LINQ to:
 - Select data
 - Filter data by row
 - Filter data by column

Demonstration: Querying Data

In this demonstration, you will use LINQ to Entities to query data

Querying Data by Using Anonymous Types

Use LINQ and anonymous types to:

- Filter data by column
- Group data
- Aggregate data
- Navigate data

Demonstration: Querying Data by Using Anonymous Types

In this demonstration, you will use LINQ to Entities to query data by using anonymous types

Forcing Query Execution

- Deferred query execution—default behavior for most queries
- Immediate query execution—default behavior for queries that return a singleton value
- Forced query execution—overrides deferred query execution:
 - **ToArray**
 - **ToDictionary**

```
IList<Employee> emp = (from e in FCEntities.Employees  
                      orderby e.LastName  
                      select e).ToList();
```

Demonstration: Retrieving and Modifying Grade Data Lab

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

Lab: Retrieving and Modifying Grade Data

- Exercise 1: Creating an Entity Data Model from The School of Fine Arts Database
- Exercise 2: Updating Student and Grade Data by Using the Entity Framework
- Exercise 3: Extending the Entity Data Model to Validate Data

Estimated Time: 75 minutes

Lab Scenario

You have been asked to upgrade the prototype application to use an existing SQL Server database. You begin by working with a database that is stored on your local machine and decide to use the Entity Data Model Wizard to generate an EDM to access the data. You will need to update the data access code for the Grades section of the application, to display grades that are assigned to a student and to enable users to assign new grades. You also decide to incorporate validation logic into the EDM to ensure that students cannot be assigned to a full class and that the data that users enter when they assign new grades conforms to the required values.

Module Review and Takeaways

- Review Questions

Module 8

Accessing Remote Data

Module Overview

- Accessing Data Across the Web
- Accessing Data by Using OData Connected Services

Lesson 1: Accessing Data Across the Web

- Overview of Web Connectivity in the .NET Framework
- Defining a Data Contract
- Creating a Request and Processing a Response
- Authenticating a Web Request
- Sending and Receiving Data
- Demonstration: Consuming a Web Service

Overview of Web Connectivity in the .NET Framework

- Use the request and response pattern
- Use the classes in the **System.Net** namespace:
 - **WebRequest** (abstract base class)
 - **WebResponse** (abstract base class)
 - **HttpRequest**
 - **HttpResponse**
 - **FtpWebRequest**
 - **FtpWebResponse**
 - **FileWebRequest**
 - **FileWebResponse**

Defining a Data Contract

Use the **DataContract** and **DataMember** attributes to expose types from a web service

```
[DataContract()]
public class SalesPerson
{
    [DataMember()]
    public string FirstName { get; set; }

    [DataMember()]
    public string LastName { get; set; }

    [DataMember()]
    public string Area { get; set; }

    [DataMember()]
    public string EmailAddress { get; set; }
}
```

Creating a Request and Processing a Response

- Get a URI

```
var uri =  
    "http://sales.fourthcoffee.com/SalesService.svc/GetSalesPerson";
```

- Create a request object

```
var request = WebRequest.Create(uri) as HttpWebRequest;
```

- Get a response object from the request object

```
var response = request.GetResponse() as HttpWebResponse;
```

- Read the properties in the response object

```
var status = response.StatusCode;  
// Returns OK if a response is received.
```

Authenticating a Web Request

- Create the request object

```
var uri =  
    "http://sales.fourthcoffee.com/SalesService.svc/GetSalesPerson";  
var request = WebRequest.Create(uri) as HttpWebRequest;
```

- Use the **NetworkCredential** class

```
var username = "jespera";  
var password = "Pa$$w0rd";  
request.Credentials = new NetworkCredential(username, password);
```

- Use the **CredentialCache** class

```
request.Credentials = CredentialCache.DefaultCredentials;
```

- Use the **X509Certificate2** class

```
var certificate = FourthCoffeeCertificateServices.GetCertificate();  
request.ClientCertificates.Add(certificate);
```


Sending and Receiving Data

- Send data

```
var uri =  
    "http://sales.fourthcoffee.com/SalesService.svc/GetSalesPerson";  
var rawData = Encoding.Default.GetBytes(  
    "{\"emailAddress\":\"jespera@fourthcoffee.com\"}");  
var request = WebRequest.Create(uri) as HttpWebRequest;  
request.Method = "POST";  
request.ContentType = "application/json";  
request.ContentLength = rawData.Length;  
var dataStream = request.GetRequestStream();  
dataStream.Write(rawData, 0, rawData.Length);  
dataStream.Close();
```

- Process the response

```
var response = request.GetResponse() as HttpWebResponse;  
var stream = new StreamReader(response.GetResponseStream());  
// Code to process the stream.  
stream.Close();
```

Demonstration: Consuming a Web Service

In this demonstration, you will use the **HttpWebRequest** and **HttpWebResponse** classes to consume a web service over HTTP

Lesson 2: Accessing Data by Using OData Connected Services

- What Is WCF Data Services?
- Defining a WCF Data Service
- Exposing a Data Model by Using WCF Data Services
- Exposing Web Methods by Using WCF Data Services
- Referencing a WCF Data Source
- Retrieving and Updating Data in a WCF Data Service
- Demonstration: Retrieving and Modifying Grade Data Remotely

What Is WCF Data Services?

WCF Data Services:

- Enables you to create highly flexible data services
- Uses the REST model for data access

`http://FourthCoffee.com/SalesService.svc/SalesPersons`

`http://FourthCoffee.com/SalesService.svc/SalesPersons/99`

`http://FourthCoffee.com/SalesService.svc/SalesPersons?top=10`

- Enables you to query and modify data by using URIs with standard HTTP verbs: GET, PUT, POST, and DELETE

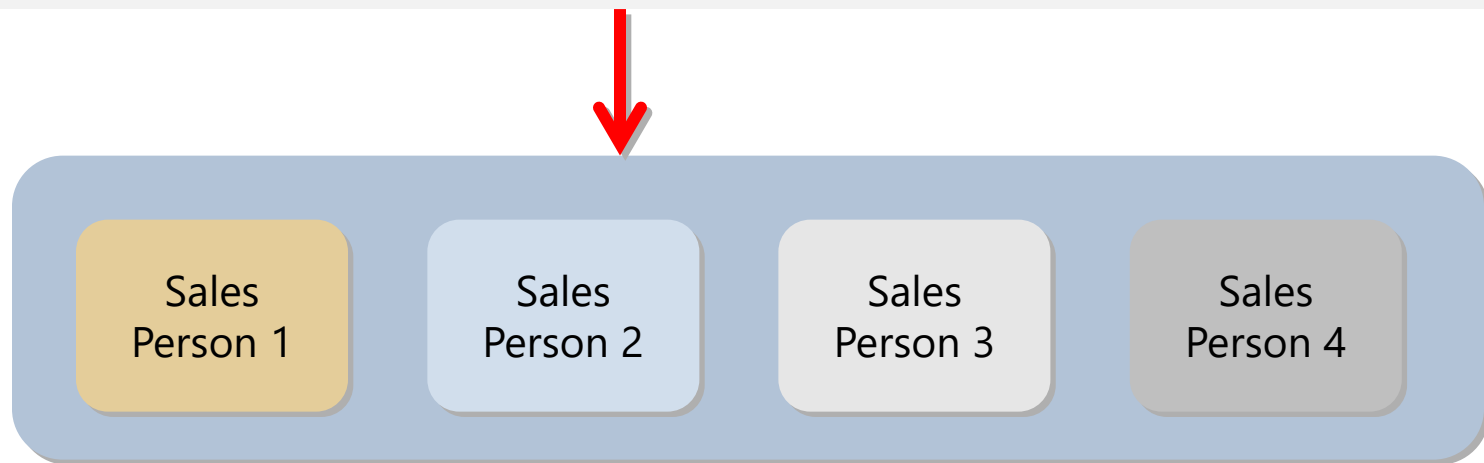
Defining a WCF Data Service

- WCF Data Services is based on the **System.Data.Services.DataService** generic class

```
public class FourthCoffeeDataService : DataService<FourthCoffee>
{
    ...
}
```

- URIs are mapped to entity sets by a data service:

<http://FourthCoffee.com/SalesService.svc/SalesPersons>



Exposing a Data Model by Using WCF Data Services

Configure the access rules on the WCF Data Service by using the **SetEntitySetAccessRule** method

```
public class FourthCoffeeDataService : DataService<FourthCoffee>
{
    public static void InitializeService(
        DataServiceConfiguration config)
    {
        config.SetEntitySetAccessRule("*", EntitySetRights.All);
    }
}
```

Exposing Web Methods by Using WCF Data Services

Expose operations by using the **WebGet** and **WebInvoke** attributes

```
public class FourthCoffeeDataService : DataService<FourthCoffee>
{
    public static void InitializeService(DataServiceConfiguration config)
    {
        config.SetServiceOperationAccessRule("SalesPersonByEmail",
            ServiceOperationRights.ReadMultiple);
    }

    [WebGet]
    [SingleResult]
    public SalesPerson SalesPersonByEmail(string emailAddress)
    {
        ...
    }
}
```

Referencing a WCF Data Source

- Client libraries:
 - Are derived from the **DataServiceContext** class
 - Expose entities that the **DataServiceQuery** collection contains
- Create a client library by using:
 - The **Add Service Reference** function in Visual Studio
 - The **DataSvcUtil** command line utility

Retrieving and Updating Data in a WCF Data Service

- Retrieve entities:
 - Use the properties that are exposed by the context
 - Invoke custom service operations
 - Use eager or explicit loading to get related entities
- Modify entities:
 - Use the **AddToXXXX** method to add a new entity
 - Use the **DeleteObject** method to remove an existing entity
 - Use the **UpdateObject** method to modify an existing entity

Demonstration: Retrieving and Modifying Grade Data Remotely

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

Lab: Retrieving and Modifying Grade Data Remotely

- Exercise 1: Creating a WCF Data Service for the SchoolGrades Database
- Exercise 2: Integrating the Data Service into the Application
- Exercise 3: Retrieving Student Photographs Over the Web (If Time Permits)

Estimated Time: 60 minutes

Lab Scenario

Currently, the application retrieves data from a local database. However, you have decided to store the data in the cloud and must configure the application so that it can retrieve data across the web.

You must create a WCF Data Service for the **SchoolGrades** database that will be integrated into the application to enable access to the data.

Finally, you have been asked to write code that displays student images by retrieving them from across the web.

Module Review and Takeaways

- Review Questions

Module 9

Designing the User Interface for a
Graphical Application

Module Overview

- Using XAML to Design a User Interface
- Binding Controls to Data
- Styling a UI

Lesson 1: Using XAML to Design a User Interface

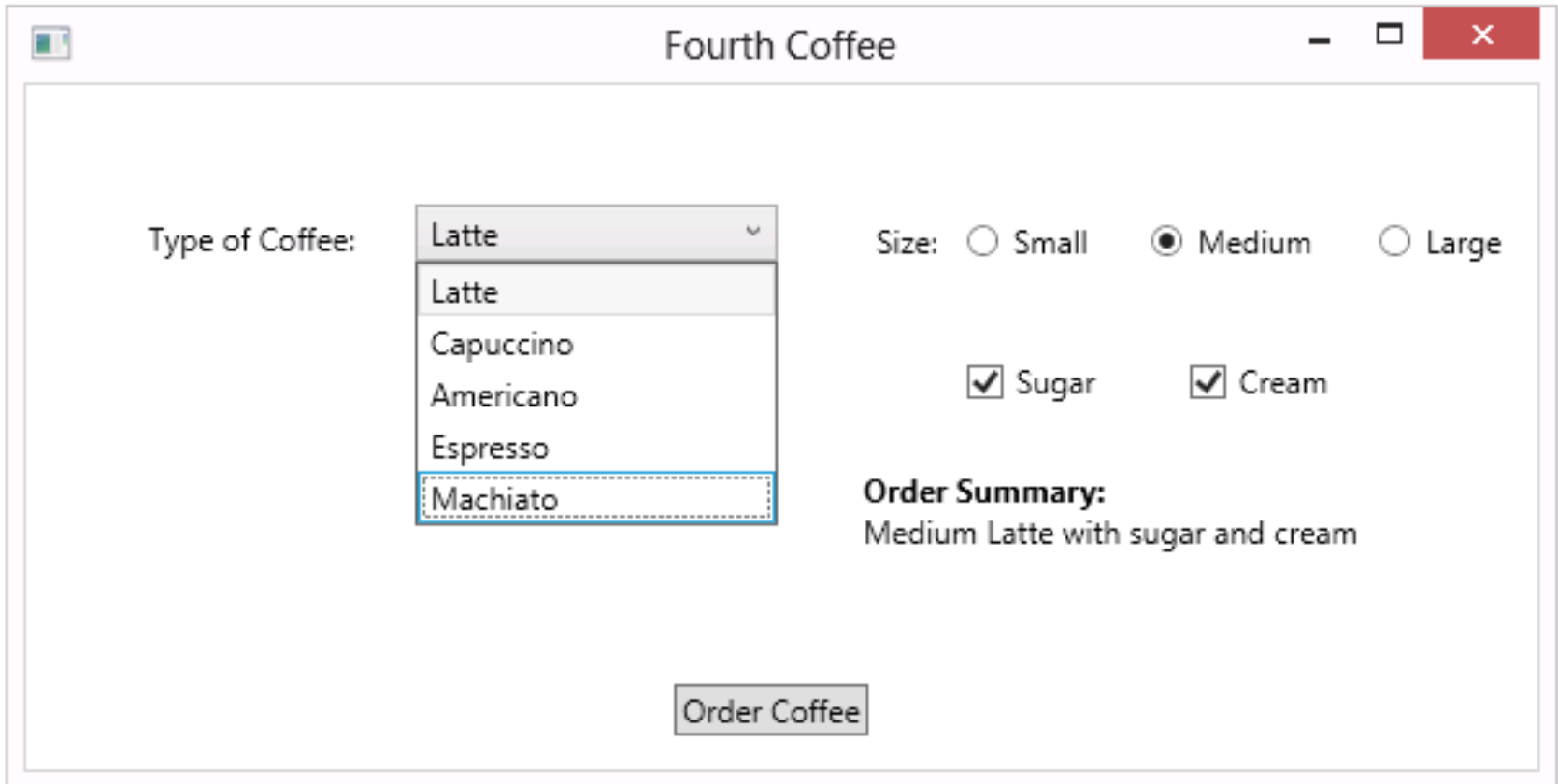
- Introducing XAML
- Common Controls
- Setting Control Properties
- Handling Events
- Using Layout Controls
- Demonstration: Using Design View to Create a XAML UI
- Creating User Controls

Introducing XAML

- Use XML elements to create controls
- Use attributes to set control properties
- Create hierarchies to represent parent controls and child controls

Common Controls

- Button
- Checkbox
- ComboBox
- Label
- ListBox
- RadioButton
- TabControl
- TextBlock
- TextBox



The screenshot shows a Windows application window titled "Fourth Coffee". The window contains a form for ordering coffee. On the left, the label "Type of Coffee:" is followed by a ComboBox. The ComboBox is open, showing a list of coffee types: "Latte", "Capuccino", "Americano", "Espresso", and "Machiato". The "Machiato" item is currently selected and highlighted with a dashed border. To the right of the ComboBox, the label "Size:" is followed by three radio buttons: "Small", "Medium", and "Large". The "Medium" radio button is selected. Below the radio buttons, there are two checkboxes: "Sugar" and "Cream", both of which are checked. At the bottom right, there is a section titled "Order Summary:" followed by the text "Medium Latte with sugar and cream". At the bottom center of the window, there is a button labeled "Order Coffee".

Fourth Coffee

Type of Coffee: Latte

Latte
Capuccino
Americano
Espresso
Machiato

Size: ☐ Small ☒ Medium ☐ Large

☒ Sugar ☒ Cream

Order Summary:
Medium Latte with sugar and cream

Order Coffee

Setting Control Properties

- Use attribute syntax to define simple property values

```
<Button Content="Click Me" Background="Yellow" />
```

- Use property element syntax to define complex property values

```
<Button Content="Click Me">  
  <Button.Background>  
    <LinearGradientBrush StartPoint="0.5, 0.5"  
                          EndPoint="1.5, 1.5">  
      <GradientStop Color="AliceBlue" Offset="0" />  
      <GradientStop Color="Aqua" Offset="0.5" />  
    </LinearGradientBrush>  
  </Button.Background>  
</Button>
```

Handling Events

- Specify the event handler method in XAML

```
<Button x:Name="btnMakeCoffee"  
        Content="Make Me a Coffee!"  
        Click="btnMakeCoffee_Click" />
```

- Handle the event in the code-behind class

```
private void btnMakeCoffee_Click(object sender,  
    RoutedEventArgs e)  
{  
    lblResult.Content = "Your coffee is on its way.";  
}
```

- Events are bubbled to parent controls

Using Layout Controls

- Canvas
- DockPanel
- Grid
- StackPanel
- VirtualizingStackPanel
- WrapPanel

Demonstration: Using Design View to Create a XAML UI

In this demonstration, you will learn how to:

- Add controls to the design surface in Visual Studio
- Edit controls by using designer tools
- Edit controls by editing XAML directly
- Use Visual Studio tools to create event handlers

Creating User Controls

- To create a user control:
 - Define the control in XAML
 - Expose properties and events in the code-behind class
- To use a user control:
 - Add an XML namespace prefix for the assembly and namespace
 - Use the control like a standard XAML control

Lesson 2: Binding Controls to Data

- Introduction to Data Binding
- Binding Controls to Data in XAML
- Binding Controls to Data in Code
- Binding Collections to Control
- Creating Data Templates

Introduction to Data Binding

- Data binding has three components:
 - Binding source
 - Binding target
 - Binding object
- A data binding can be bidirectional or unidirectional:
 - TwoWay
 - OneWay
 - OneTime
 - OneWayToSource
 - Default

Binding Controls to Data in XAML

- Use a binding expression to identify the source object and the source property

```
<TextBlock  
    Text="{Binding Source={StaticResource coffee1},  
        Path=Bean}" />
```

- Specify the data context on a parent control

```
<StackPanel>  
    <StackPanel.DataContext>  
        <Binding Source="{StaticResource coffee1}" />  
    </StackPanel.DataContext>  
    <TextBlock Text="{Binding Path=Name}" />  
    ...  
</StackPanel>
```

Binding Controls to Data in Code

- Create data binding entirely in code
- Create **Path** bindings in XAML and set the **DataContext** in code

```
<StackPanel x:Name="stackCoffee">  
    <TextBlock Text="{Binding Path=Name}" />  
    <TextBlock Text="{Binding Path=Bean}" />  
    <TextBlock Text="{Binding Path=CountryOfOrigin}" />  
    <TextBlock Text="{Binding Path=Strength}" />  
</StackPanel>
```

```
stackCoffee.DataContext = coffee1;
```

Binding Collections to Control

- Set the **ItemsSource** property to bind to an **IEnumerable** collection

```
IstCoffees.ItemsSource = coffees;
```

- Use the **DisplayMemberPath** property to specify the source field to display

```
<ListBox x:Name="IstCoffees"  
        DisplayMemberPath="Name" />
```

Creating Data Templates

Specify how each item in a collection should be displayed

```
<DataTemplate>
  <Grid>
    ...
    <TextBlock Text="{Binding Path=Name}" Grid.Row="0"
      FontSize="22" Background="Black"
      Foreground="White" />
    <TextBlock Text="{Binding Path=Bean}"
      Grid.Row="1" />
    <TextBlock Text="{Binding Path=CountryOfOrigin}"
      Grid.Row="2" />
    <TextBlock Text="{Binding Path=Strength}"
      Grid.Row="3" />
  </Grid>
</DataTemplate>
```

Lesson 3: Styling a UI

- Creating Reusable Resources in XAML
- Defining Styles as Resources
- Using Property Triggers
- Creating Dynamic Transformations
- Demonstration: Customizing Student Photographs and Styling the Application Lab

Creating Reusable Resources in XAML

- Define resources in a **Resources** collection
- Add an **x:Key** to uniquely identify the resource

```
<Window.Resources>  
    <SolidColorBrush x:Key="MyBrush" Color="Coral" />  
    ...  
</Window.Resources>
```

- Reference the resource in property values

```
<TextBlock Text="Foreground"  
    Foreground="{StaticResource MyBrush}" />
```

- Use a resource dictionary to manage large collections of resources

Defining Styles as Resources

- Identify the target control type
- Provide an **x:Key** value if required
- Use **Setter** elements to specify property values

```
<Style TargetType="TextBlock" x:Key="BlockStyle1">  
  <Setter Property="FontSize" Value="20" />  
  <Setter Property="Background" Value="Black" />  
  ...  
</Style>
```

- Reference the style as a static resource

```
<TextBlock Text="Drink More Coffee"  
  Style="{StaticResource BlockStyle1}" />
```


Using Property Triggers

Use triggers to apply style properties based on conditions:

- Use the **Trigger** element to identify the condition
- Use **Setter** elements apply the conditional changes

```
<Style TargetType="Button">
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="FontWeight" Value="Bold" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Creating Dynamic Transformations

- Use an **EventTrigger** to identify the event that starts the animation
- Use a **Storyboard** to identify the properties that should change
- Use a **DoubleAnimation** to define the changes

```
<EventTrigger RoutedEvent="Image.MouseDown">  
  <BeginStoryboard>  
    <Storyboard>  
      <DoubleAnimation  
        Storyboard.TargetProperty="Height"  
        From="200" To="300" Duration="0:0:2" />  
    </Storyboard>  
  </BeginStoryboard>  
</EventTrigger>
```

Demonstration: Customizing Student Photographs and Styling the Application Lab

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

Lab: Customizing Student Photographs and Styling the Application

- Exercise 1: Customizing the Appearance of Student Photographs
- Exercise 2: Styling the Logon View
- Exercise 3: Animating the StudentPhoto Control (If Time Permits)

Estimated Time: 90 minutes

Lab Scenario

Now that you and The School of Fine Arts are happy with the basic functionality of the application, you need to improve the appearance of the interface to give the user a nicer experience through the use of animations and a consistent look and feel.

You decide to create a `StudentPhoto` control that will enable you to display photographs of students in the student list and other views. You also decide to create a fluid method for a teacher to remove a student from their class. Finally, you want to update the look of the various views, keeping their look consistent across the application.

Module Review and Takeaways

- Review Questions

Module 10

Improving Application Performance
and Responsiveness

Module Overview

- Implementing Multitasking
- Performing Operations Asynchronously
- Synchronizing Concurrent Access to Data

Lesson 1: Implementing Multitasking

- Creating Tasks
- Controlling Task Execution
- Returning a Value from a Task
- Cancelling Long-Running Tasks
- Running Tasks in Parallel
- Linking Tasks
- Handling Task Exceptions

Creating Tasks

- Use an **Action** delegate

```
Task task1 = new Task(new Action(MyMethod));
```

- Use an anonymous delegate/anonymous method

```
Task task2 = new Task(delegate  
{  
    Console.WriteLine("Task 2 reporting");  
});
```

- Use lambda expressions (recommended)

```
Task task2 = new Task(() =>  
{  
    Console.WriteLine(" Task 2 reporting");  
});
```

Controlling Task Execution

- To start a task:
 - **Task.Start** instance method
 - **Task.Factory.StartNew** static method
 - **Task.Run** static method
- To wait for tasks to complete:
 - **Task.Wait** instance method
 - **Task.WaitAll** static method
 - **Task.WaitAny** static method

Returning a Value from a Task

- Use the **Task<TResult>** class
- Specify the return type in the type argument

```
Task<string> task1 = Task.Run<string>( () =>  
    DateTime.Now.DayOfWeek.ToString() );
```

- Get the result from the **Result** property

```
Console.WriteLine("Today is {0}", task1.Result);
```

Cancelling Long-Running Tasks

- Pass a cancellation token as an argument to the delegate method
- Request cancellation from the joining thread
- In the delegate method, check whether the cancellation token is cancelled
- Return or throw an **OperationCanceledException** exception

Running Tasks in Parallel

- Use **Parallel.Invoke** to run multiple tasks simultaneously

```
Parallel.Invoke( () => MethodForFirstTask(),  
                () => MethodForSecondTask(),  
                () => MethodForThirdTask() );
```

- Use **Parallel.For** to run **for** loop iterations in parallel
- Use **Parallel.ForEach** to run **foreach** loop iterations in parallel
- Use PLINQ to run LINQ expressions in parallel

Linking Tasks

- Use task continuations to chain tasks together:
 - **Task.ContinueWith** method links continuation task to antecedent task
 - Continuation task starts when antecedent task completes
 - Antecedent task can pass result to continuation task
- Use nested tasks if you want to start an *independent* task from a task delegate
- Use child tasks if you want to start a *dependent* task from a task delegate

Handling Task Exceptions

- Call **Task.Wait** to catch propagated exceptions
- Catch **AggregateException** in the **catch** block
- Iterate the **InnerExceptions** property and handle individual exceptions

```
try
{
    task1.Wait();
}
catch(AggregateException ae)
{
    foreach(var inner in ae.InnerExceptions)
    {
        // Deal with each exception in turn.
    }
}
```


Lesson 2: Performing Operations Asynchronously

- Using the Dispatcher
- Using `async` and `await`
- Creating Awaitable Methods
- Creating and Invoking Callback Methods
- Working with APM Operations
- Demonstration: Using the Task Parallel Library to Invoke APM Operations
- Handling Exceptions from Awaitable Methods

Using the Dispatcher

- To update a UI element from a background thread:
 - Get the **Dispatcher** object for the thread that owns the UI element
 - Call the **BeginInvoke** method
 - Provide an **Action** delegate as an argument

```
IblTime.Dispatcher.BeginInvoke(new Action(() =>  
    SetTime(currentTime)));
```

Using async and await

- Add the **async** modifier to method declarations
- Use the **await** operator within **async** methods to wait for a task to complete without blocking the thread

```
private async void btnLongOperation_Click(object sender,
RoutedEventArgs e)
{
    ...
    Task<string> task1 = Task.Run<string>(() =>
    {
        ...
    })
    lblResult.Content = await task1;
}
```

Creating Awaitable Methods

- The **await** operator is always used to wait for a task to complete
- If your synchronous method returns **void**, the asynchronous equivalent should return **Task**
- If your synchronous method has a return type of **T**, the asynchronous equivalent should return **Task<T>**

Creating and Invoking Callback Methods

- Use the **Action<T>** delegate to represent your callback method
- Add the delegate to your asynchronous method parameters

```
public async Task  
GetCoffees(Action<IEnumerable<string>> callback)
```

- Invoke the delegate asynchronously within your method

```
await Task.Run(() => callback(coffees));
```

Working with APM Operations

- Use the **TaskFactory.FromAsync** method to call methods that implement the APM pattern

```
HttpRequest request =  
    (HttpRequest)WebRequest.Create(url);  
  
HttpResponse response =  
    await Task<WebResponse>.Factory.FromAsync(  
        request.BeginGetResponse,  
        request.EndGetResponse,  
        request) as HttpResponse;
```

Demonstration: Using the Task Parallel Library to Invoke APM Operations

- In this demonstration, you will learn how to:
 - Use a conventional approach to invoke APM operations
 - Use the Task Parallel Library to invoke APM operations
 - Compare the two approaches

Handling Exceptions from Awaitable Methods

- Use a conventional **try/catch** block to catch exceptions in asynchronous methods
- Subscribe to the **TaskScheduler.UnobservedTaskException** event to create an event handler of last resort

```
TaskScheduler.UnobservedTaskException +=  
    (object sender, UnobservedTaskExceptionEventArgs e) =>  
    {  
        // Respond to the unobserved task exception.  
    }
```


Lesson 3: Synchronizing Concurrent Access to Data

- Using Locks
- Demonstration: Using Lock Statements
- Using Synchronization Primitives with the Task Parallel Library
- Using Concurrent Collections
- Demonstration: Improving the Responsiveness and Performance of the Application Lab

Using Locks

- Create a private object to apply the lock to
- Use the **lock** statement and specify the locking object
- Enclose your critical section of code in the **lock** block

```
private object lockingObject = new object();  
lock (lockingObject)  
{  
    // Only one thread can enter this block at any one time.  
}
```

Demonstration: Using Lock Statements

In this demonstration, you will see how to:

- Use **lock** statements to apply mutual-exclusion locks to critical sections of code
- Observe the consequences if you omit the **lock** statement

Using Synchronization Primitives with the Task Parallel Library

- Use the **ManualResetEventSlim** class to limit resource access to one thread at a time
- Use the **SemaphoreSlim** class to limit resource access to a fixed number of threads
- Use the **CountdownEvent** class to block a thread until a fixed number of tasks signal completion
- Use the **ReaderWriterLockSlim** class to allow multiple threads to read a resource or a single thread to write to a resource at any one time
- Use the **Barrier** class to block multiple threads until they all satisfy a condition

Using Concurrent Collections

The **System.Collections.Concurrent** namespace includes generic classes and interfaces for thread-safe collections:

- **ConcurrentBag<T>**
- **ConcurrentDictionary<TKey, TValue>**
- **ConcurrentQueue<T>**
- **ConcurrentStack<T>**
- **IProducerConsumerCollection<T>**
- **BlockingCollection<T>**

Demonstration: Improving the Responsiveness and Performance of the Application Lab

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

Lab: Improving the Responsiveness and Performance of the Application

- Exercise 1: Ensuring That the UI Remains Responsive When Retrieving Teacher Data
- Exercise 2: Providing Visual Feedback During Long-Running Operations

Estimated Time: 75 minutes

Lab Scenario

You have been asked to update the Grades application to ensure that the UI remains responsive while the user is waiting for operations to complete. To achieve this improvement in responsiveness, you decide to convert the logic that retrieves the list of students for a teacher to use asynchronous methods. You also decide to provide visual feedback to the user to indicate when an operation is taking place.

Module Review and Takeaways

- Review Questions

Module 11

Integrating with Unmanaged Code

Module Overview

- Creating and Using Dynamic Objects
- Managing the Lifetime of Objects and Controlling Unmanaged Resources

Lesson 1: Creating and Using Dynamic Objects

- What Are Dynamic Objects?
- What Is the Dynamic Language Runtime?
- Creating a Dynamic Object
- Invoking Methods on a Dynamic Object
- Demonstration: Interoperating with Microsoft Word

What Are Dynamic Objects?

- Objects that do not conform to the strongly typed object model
- Objects that enable you to take advantage of dynamic languages, such as IronPython
- Objects that simplify the process of interoperating with unmanaged code

What Is the Dynamic Language Runtime?

The DLR provides:

- Support for dynamic languages, such as IronPython
- Run-time type checking for dynamic objects
- Language binders to handle the intricate details of interoperating with another language

Creating a Dynamic Object

- Dynamic objects are declared by using the **dynamic** keyword

```
using Microsoft.Office.Interop.Word;  
...  
dynamic word = new Application();
```

- Dynamic objects are variables of type **object**
- Dynamic objects do not support:
 - Type checking at compile time
 - Visual Studio IntelliSense

Invoking Methods on a Dynamic Object

- You can access members by using the dot notation

```
string filePath = "C:\\FourthCoffee\\Documents\\Schedule.docx";  
...  
dynamic word = new Application();  
dynamic doc = word.Documents.Open(filePath);  
doc.SaveAs(filePath);
```

- You do not need to:
 - Pass **Type.Missing** to satisfy optional parameters
 - Use the **ref** keyword
 - Pass all parameters as type **object**

Demonstration: Interoperating with Microsoft Word

In this demonstration, you will use dynamic objects to consume the Microsoft.Office.Word.Interop COM assembly in an existing .NET Framework application

Lesson 2: Managing the Lifetime of Objects and Controlling Unmanaged Resources

- The Object Life Cycle
- Implementing the Dispose Pattern
- Managing the Lifetime of an Object
- Demonstration: Upgrading the Grades Report Lab

The Object Life Cycle

- When an object is created:
 1. Memory is allocated
 2. Memory is initialized to the new object
- When an object is destroyed:
 1. Resources are released
 2. Memory is reclaimed

Implementing the Dispose Pattern

Implement the **IDisposable** interface

```
public class ManagedWord : IDisposable
{
    bool _isDisposed;

    ~ManagedWord
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool isDisposing) { ... }
}
```

Managing the Lifetime of an Object

- Explicitly invoke the **Dispose** method

```
var word = default(ManagedWord);  
try  
{  
    word = new ManagedWord();  
    // Code to use the ManagedWord object.  
}  
finally  
{  
    if(word!=null) word.Dispose();  
}
```

- Implicitly invoke the **Dispose** method

```
using (var word = default(ManagedWord))  
{  
    // Code to use the ManagedWord object.  
}
```

Demonstration: Upgrading the Grades Report Lab

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

Lab: Upgrading the Grades Report

- Exercise 1: Generating the Grades Report by Using Word
- Exercise 2: Controlling the Lifetime of Word Objects by Implementing the Dispose Pattern

Estimated Time: 60 minutes

Lab Scenario

You have been asked to upgrade the grades report functionality to generate reports in Word format. In Module 6, you wrote code that generates reports as an XML file; now you will update the code to generate the report as a Word document.

Module Review and Takeaways

- Review Questions

Module 12

Creating Reusable Types and
Assemblies

Module Overview

- Examining Object Metadata
- Creating and Using Custom Attributes
- Generating Managed Code
- Versioning, Signing, and Deploying Assemblies

Lesson 1: Examining Object Metadata

- What Is Reflection?
- Loading Assemblies by Using Reflection
- Examining Types by Using Reflection
- Invoking Members by Using Reflection
- Demonstration: Inspecting Assemblies

What Is Reflection?

- Reflection enables you to inspect and manipulate assemblies at run time
- The **System.Reflection** namespace contains:
 - **Assembly**
 - **TypeInfo**
 - **ParameterInfo**
 - **ConstructorInfo**
 - **FieldInfo**
 - **MemberInfo**
 - **PropertyInfo**
 - **MethodInfo**

Loading Assemblies by Using Reflection

- The **Assembly.LoadFrom** method

```
var assemblyPath = "...";  
var assembly = Assembly.LoadFrom(assemblyPath);
```

- The **Assembly.ReflectionOnlyLoad** method

```
var assemblyPath = "...";  
var rawBytes = File.ReadAllBytes(assemblyPath);  
var assembly = Assembly.ReflectionOnlyLoad(rawBytes);
```

- The **Assembly.ReflectionOnlyLoadFrom** method

```
var assemblyPath = "...";  
var assembly = Assembly.ReflectionOnlyLoadFrom(assemblyPath);
```

Examining Types by Using Reflection

- Get a type by name

```
var assembly = FourthCoffeeServices.GetAssembly();  
var type = assembly.GetType("...");
```

- Get all of the constructors

```
var constructors = type.GetConstructors();
```

- Get all of the fields

```
var fields = type.GetFields();
```

- Get all of the properties

```
var properties = type.GetProperties();
```

- Get all of the methods

```
var methods = type.GetMethods();
```

Invoking Members by Using Reflection

- Instantiate a type

```
var type = FourthCoffeeServices.GetHandleErrorType();  
...  
var constructor = type.GetConstructor(new Type[0]);  
...  
var initializedObject = constructor.Invoke(new object[0]);
```

- Invoke methods on the instance

```
var methodToExecute = type.GetMethod("LogError");  
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();  
...  
var response = methodToExecute.Invoke(initializedObject,  
    new object[] { "Error message" }) as string;
```

- Get or set property values on the instance

```
var property = type.GetProperty("LastErrorMessages");  
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();  
...  
var lastErrorMessage = property.GetValue(initializedObject) as string;
```


Demonstration: Inspecting Assemblies

In this demonstration, you will create a tool that you can use to inspect the contents of an assembly

Lesson 2: Creating and Using Custom Attributes

- What Are Attributes?
- Creating and Using Custom Attributes
- Processing Attributes by Using Reflection
- Demonstration: Consuming Custom Attributes by Using Reflection

What Are Attributes?

- Use attributes to provide additional metadata about an element
- Use attributes to alter run-time behavior

```
[DataContract(Name = "SalesPersonContract", IsReference=false)]
public class SalesPerson
{
    [Obsolete("This property will be removed in the next release.")]
    [DataMember]
    public string Name { get; set; }

    ...
}
```

Creating and Using Custom Attributes

Derive from the **Attribute** class or another attribute

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperInfoAttribute : Attribute
{
    private string _emailAddress;
    private int _revision;

    public DeveloperInfo(string emailAddress, int revision)
    {
        this._emailAddress = emailAddress;
        this._revision = revision;
    }
}
```

```
[DeveloperInfo("holly@fourthcoffee.com", 3)]
public class SalePerson
{
    ...
}
```

Processing Attributes by Using Reflection

Use reflection to access the metadata that is encapsulated in custom attributes

```
var type = FourthCoffee.GetSalesPersonType();  
  
var attributes = type.GetCustomAttributes(typeof(DeveloperInfo),  
false);  
  
foreach (var attribute in attributes)  
{  
    var developerEmailAddress = attribute.EmailAddress;  
    var codeRevision = attribute.Revision;  
}
```

Demonstration: Consuming Custom Attributes by Using Reflection

In this demonstration, you will use reflection to read the **DeveloperInfo** attributes that have been used to provide additional metadata on types and type members

Lesson 3: Generating Managed Code

- What Is CodeDOM?
- Defining a Type and Type Members
- Compiling a CodeDOM Model
- Compiling Source Code into an Assembly

What Is CodeDOM?

- Define a model that represents your code by using:
 - The **CodeCompileUnit** class
 - The **CodeNamespace** class
 - The **CodeTypeDeclaration** class
 - The **CodeMemberMethod** class
- Generate source code from the model:
 - Visual C# by using the **CSharpCodeProvider** class
 - JScript by using the **JScriptCodeProvider** class
 - Visual Basic by using the **VBCodeProvider** class
- Generate a .dll or a .exe that contains your code

Defining a Type and Type Members

Defining a type with a **Main** method

```
var unit = new CodeCompileUnit();

var dynamicNamespace = new CodeNamespace("FourthCoffee.Dynamic");
unit.Namespaces.Add(dynamicNamespace);

dynamicNamespace.Imports.Add(new CodeNamespaceImport("System"));

var programType = new CodeTypeDeclaration("Program");
dynamicNamespace.Types.Add(programType);

var mainMethod = new CodeEntryPointMethod();
programType.Members.Add(mainMethod);

var expression = new CodeMethodInvokeExpression(
    new CodeTypeReferenceExpression("Console"), "WriteLine",
    new CodePrimitiveExpression("Hello Development Team..!!"));
```

Compiling a CodeDOM Model

Generate source code files from your CodeDOM model

```
var provider = new CSharpCodeProvider();

var fileName = "program.cs";
var stream = new StreamWriter(fileName);
var textWriter = new IndentedTextWriter(stream);

var options = new CodeGeneratorOptions();
options.BlankLinesBetweenMembers = true;

var compileUnit = FourthCoffee.GetModel();
provider.GenerateCodeFromCompileUnit(
    compileunit,
    textWriter,
    options);

textWriter.Close();
stream.Close();
```

Compiling Source Code into an Assembly

Generate an assembly from your source code files

```
var provider = new CSharpCodeProvider();

var compilerSettings = new CompilerParameters();
compilerSettings.ReferencedAssemblies.Add("System.dll");
compilerSettings.GenerateExecutable = true;
compilerSettings.OutputAssembly = "FourthCoffee.exe";

var sourceCodeFileName = "program.cs";
var compilationResults = provider.CompileAssemblyFromFile(
    compilerSettings,
    sourceCodeFileName);

var buildFailed = false;
foreach (var error in compilationResults.Errors)
{
    var errorMessage = error.ToString();
    buildFailed = true;
}
```

Lesson 4: Versioning, Signing, and Deploying Assemblies

- What Is an Assembly?
- What Is the GAC?
- Signing Assemblies
- Versioning Assemblies
- Installing an Assembly into the GAC
- Demonstration: Signing and Installing an Assembly into the GAC
- Demonstration: Specifying the Data to Include in the Grades Report Lab

What Is an Assembly?

- An assembly is a collection of types and resources
- An assembly is a versioned deployable unit
- An assembly can contain:
 - IL code
 - Resources
 - Type metadata
 - Manifest

What Is the GAC?

- The GAC provide a robust solution to share assemblies between multiple application on the same machine
- Find the contents of the GAC at `C:\Windows\assembly`
- Benefits:
 - Side-by-side deployment
 - Improved loading time
 - Reduced memory consumption
 - Improved search time
 - Improved maintainability

Signing Assemblies

Sign an assembly:

- Create a key file
- Associate the key file with an assembly

```
sn -k FourthCoffeeKeyFile.snk
```

Delay the signing of an assembly:

1. Open the properties for the project

```
[assembly: AssemblyKeyFileAttribute("FourthCoffeeKeyFile.snk")]
```

2. Click the **Signing** tab
3. Select the **Sign the assembly** check box
4. Specify a key file
5. Select the **Delay sign only** check box

Versioning Assemblies

A version number of an assembly is a four-part string:

```
<major version>.<minor version>.<build number>.<revision>
```

Applications reference particular versions of assemblies

```
<configuration>  
  <runtime>  
    <assemblyBinding xmlns="...">  
      <dependentAssembly>  
        <assemblyIdentity name="FourthCoffee.Core"  
          publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />  
        <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>  
      </dependentAssembly>  
    </assemblyBinding>  
  </runtime>  
</configuration>
```


Installing an Assembly into the GAC

Install an assembly in the GAC by using:

- Global Assembly Cache tool
- Microsoft Windows Installer

Examples:

- Install an assembly by using Gacutil.exe:

```
gacutil -i "<pathToAssembly>"
```

- View an assembly by using Gacutil.exe:

```
gacutil -l "<assemblyName>"
```

Demonstration: Signing and Installing an Assembly into the GAC

In this demonstration, you will use the Sn.exe and Gacutil.exe command-line tools to sign and install an existing assembly into the GAC

Demonstration: Specifying the Data to Include in the Grades Report Lab

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

Lab: Specifying the Data to Include in the Grades Report

- Exercise 1: Creating and Applying the IncludeInReport attribute
- Exercise 2: Updating the Report
- Exercise 3: Storing the Grades.Utilities Assembly Centrally (If Time Permits)

Estimated Time: 75 minutes

Lab Scenario

You decide to update the Grades application to use custom attributes to define the fields and properties that should be included in a grade report and to format them appropriately. This will enable further reusability of the Microsoft Word reporting functionality.

You will host this code in the GAC to ensure that it is available to other applications that require its services.

Module Review and Takeaways

- Review Questions

Module 13

Encrypting and Decrypting Data

Module Overview

- Implementing Symmetric Encryption
- Implementing Asymmetric Encryption

Lesson 1: Implementing Symmetric Encryption

- What Is Symmetric Encryption?
- Encrypting Data by Using Symmetric Encryption
- Hashing Data
- Demonstration: Encrypting and Decrypting Data

What Is Symmetric Encryption?

- Symmetric encryption is the cryptographic transformation of data by using a mathematical algorithm
- The same key is used to encrypt and decrypt the data
- The **System.Security.Cryptography** namespace includes:
 - **DESCryptoServiceProvider** class
 - **AesManaged** class
 - **RC2CryptoServiceProvider** class
 - **RijndaelManaged** class
 - **TripleDESCryptoServiceProvider** class

Encrypting Data by Using Symmetric Encryption

To encrypt and decrypt data symmetrically, perform the following steps:

1. Create an **Rfc2898DeriveBytes** object
2. Create an **AesManaged** object
3. Generate a secret key and an IV
4. Create a stream to buffer the transformed data
5. Create a symmetric encryptor or decryptor object
6. Create a **CryptoStream** object
7. Write the transformed data to the buffer stream
8. Close the streams

Hashing Data

- A hash is a numerical representation of a piece of data
- A hash can be computed by using the following code

```
public byte[] ComputeHash(byte[] dataToHash, byte[] secretKey)
{
    using (var hashAlgorithm = new HMACSHA1(secretKey))
    {
        using (var bufferStream = new MemoryStream(dataToHash))
        {
            return hashAlgorithm.ComputeHash(bufferStream);
        }
    }
}
```

Demonstration: Encrypting and Decrypting Data

In this demonstration, you will use symmetric encryption to encrypt and decrypt a message

Lesson 2: Implementing Asymmetric Encryption

- What Is Asymmetric Encryption?
- Encrypting Data by Using Asymmetric Encryption
- Creating and Managing X509 Certificates
- Managing Encryption Keys
- Demonstration: Encrypting and Decrypting Grade Reports Lab

What Is Asymmetric Encryption?

- Asymmetric encryption uses:
 - A public key to encrypt data
 - A private key to decrypt data
- The **System.Security.Cryptography** namespace includes:
 - The **RSACryptoServiceProvider** class
 - The **DSACryptoServiceProvider** class

Encrypting Data by Using Asymmetric Encryption

To encrypt and decrypt data asymmetrically

```
var rawBytes = Encoding.Default.GetBytes("hello world..");
var decryptedText = string.Empty;

using (var rsaProvider = new RSACryptoServiceProvider())
{
    var useOaepPadding = true;

    var encryptedBytes =
        rsaProvider.Encrypt(rawBytes, useOaepPadding);

    var decryptedBytes =
        rsaProvider.Decrypt(encryptedBytes, useOaepPadding);

    decryptedText = Encoding.Default.GetString(decryptedBytes);
}
// decryptedText == hello world..
```


Creating and Managing X509 Certificates

- Use MakeCert to create certificates

```
makecert -n "CN=FourthCoffee" -a sha1 -pe -r -sr LocalMachine -  
ss my -sky exchange
```

- Use the MMC Certificates snap-in to manage your certificate stores

Managing Encryption Keys

The

System.Security.Cryptography.X509Certificates namespace contains classes that enable access to the certificate store and certificate metadata

```
var store = new X509Store(  
    StoreName.My,  
    StoreLocation.LocalMachine);  
  
store.Open(OpenFlags.ReadOnly);  
  
foreach (var storeCertificate in store.Certificates)  
{  
    // Code to process each certificate.  
}  
  
store.Close();
```

Demonstration: Encrypting and Decrypting Grade Reports Lab

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

Lab: Encrypting and Decrypting the Grades Report

- Exercise 1: Encrypting the Grades Report
- Exercise 2: Decrypting the Grades Report

Estimated Time: 60 minutes

Lab Scenario

You have been asked to update the Grades application to ensure that reports are secure when they are stored on a user's computer. You decide to use asymmetric encryption to protect the report as it is generated, before it is written to disk. Administrative staff will need to merge reports for each class into one document, so you decide to develop a separate application that generates a combined report and prints it.

Module Review and Takeaways

- Review Questions

Course Evaluation

- Your evaluation of this course will help Microsoft understand the quality of your learning experience.
- Please work with your training provider to access the course evaluation form.
- Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.