C# Programming

1

Microsoft Power Platform

Tissana Tanaklang

Software and Solution Development Trainer
Iverson Training Center Co., Ltd.
tissana@iverson.co.th , tissana_t@hotmail.com

- Master of Science Program in Software Engineering King Mongkut's University of Technology Thonburi
- Bachelor of Science Program in Computer Science Naresuan University

- Microsoft Certified Trainer (MCT)

- Microsoft Certified Azure Data Engineer Associate
- Microsoft Certified Solutions Associate - Web Application Development

- Microsoft Certified Azure Fundamentals
- Microsoft Certified Azure Data Fundamentals
- Microsoft Certified Azure AI Fundamentals

2

Course Outline

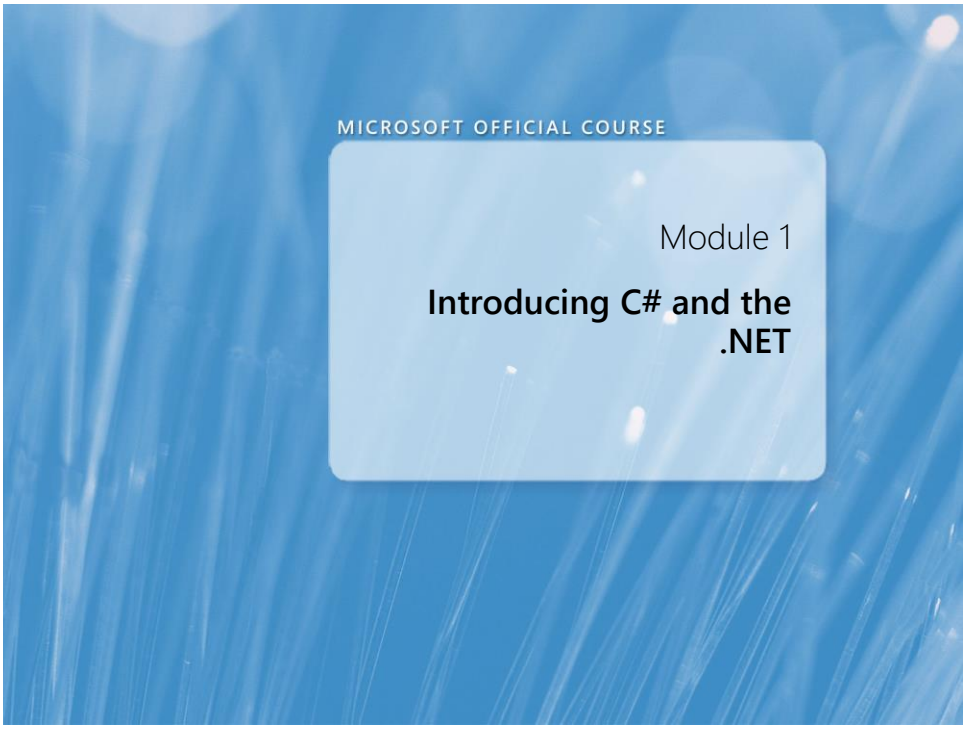
- Module 1: Introducing C# and the .NET
- Module 2: Using C# Programming Constructs
- Module 3: Declaring and Calling Methods
- Module 4: Handling Exceptions
- Module 5: Reading and Writing Files
- Module 6: Creating New Types
- Module 7: Encapsulating Data and Methods
- Module 8: Inheriting from Classes and Implementing Interfaces
- Module 9: Managing the Lifetime of Objects and Controlling Resources

3

Course Outline *(continued)*

- Module 10: Encapsulating Data and Defining Overloaded Operators
- Module 11: Decoupling Methods and Handling Events
- Module 12: Using Collections and Building Generic Types
- Module 13: Building and Enumerating Custom Collection Classes
- Module 14: Entity Framework and LINQ to Query Data
- Module 15: C# Test-Driven Development Concept

4



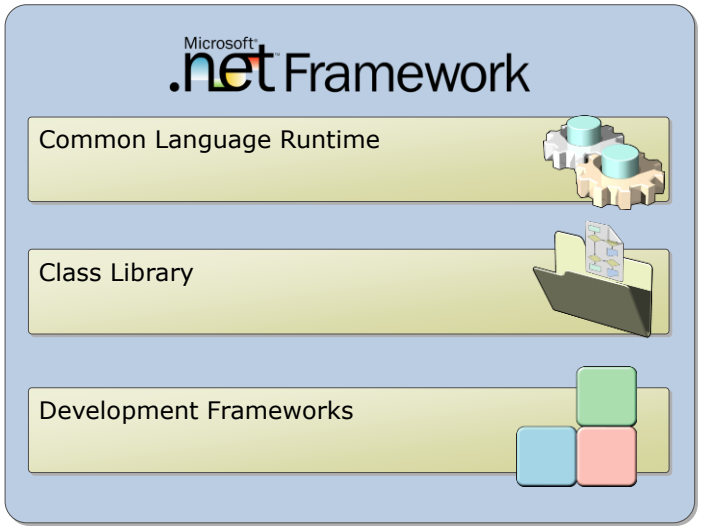
5

Module Overview

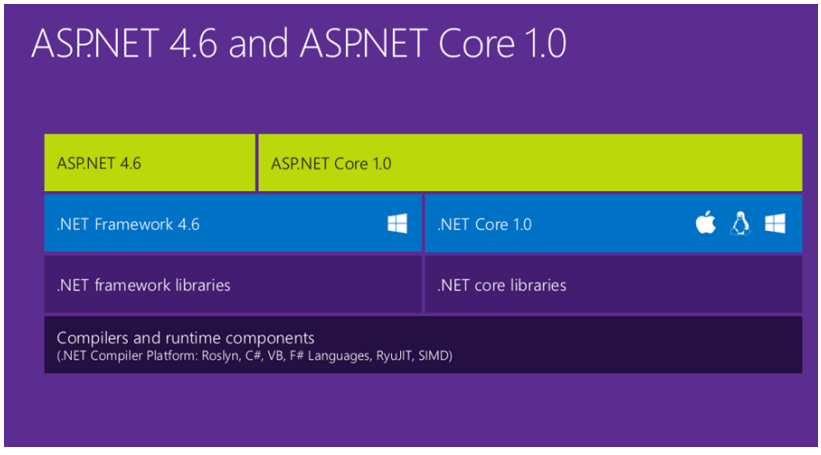
- Introduction to the .NET
- Creating Projects Within Visual Studio
- Writing a C# Application
- Debugging Applications By Using Visual Studio

6

What Is the .NET

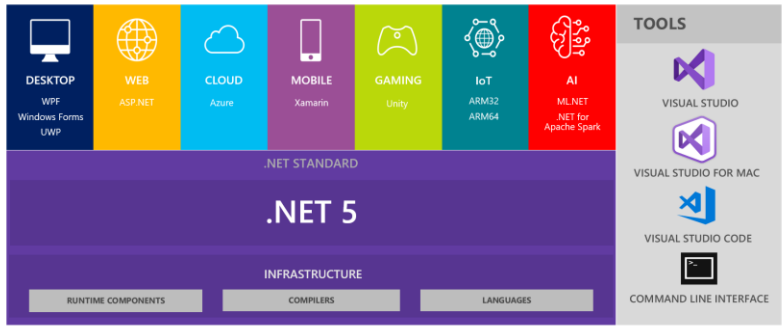


7



8

.NET – A unified platform



9

The Purpose of Visual C#

C#

- C# is the language of choice for many developers who build .NET Framework applications
- C# uses a very similar syntax to C, C++, and Java
- C# has been standardized and is described by the ECMA-334 C# Language Specification

10

Lesson 2: Creating Projects Within Visual Studio

- Key Features of Visual Studio
- Templates in Visual Studio
- The Structure of Visual Studio Projects and Solutions
- Creating a .NET Framework Application
- Building and Running a .NET Framework Application
- Demonstration: Disassembling a .NET Framework Assembly

11

Key Features of Visual Studio

Visual Studio :

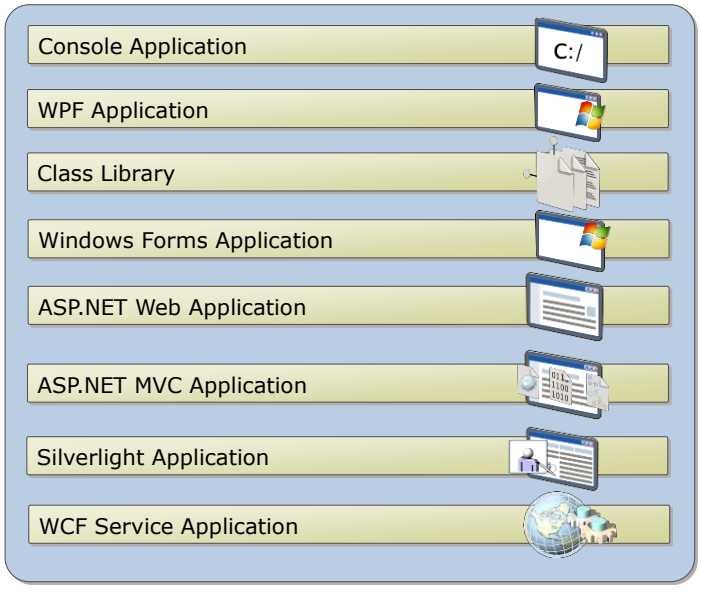
Intuitive IDE that enables developers to quickly build applications in their chosen programming language

Visual Studio features:

- Rapid application development
- Server and data access
- Debugging features
- Error handling
- Help and documentation

12

Templates in Visual Studio



13

The Structure of Visual Studio Projects and Solutions

Visual Studio Solution

Visual Studio solutions are wrappers for .NET projects

Visual Studio solutions can contain multiple .NET projects

Visual Studio solutions can contain different types of .NET projects

ASP.NET project

.aspx .csproj

.aspx.cs .config

WPF project

.xaml .csproj

.xaml.cs .config

Console project

.cs .csproj

.config

14

Creating a .NET Framework Application

1. Open Visual Studio
2. On the **File** menu, click **New**, and then click **Project**
3. In the **New Project** dialog box, specify the following, and then click **OK**:
   - Project template
   - Project name
   - Project save path

Programmer productivity features include:

- IntelliSense
- Code snippets

15

Lesson 3: Writing a C# Application

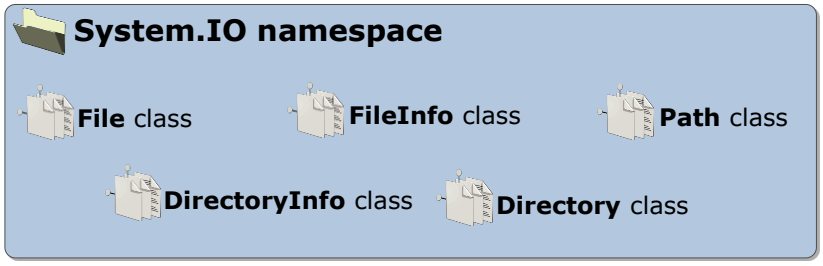
- What Are Classes and Namespaces?
- The Structure of a Console Application
- Performing Input and Output by Using a Console Application
- Best Practices for Commenting C# Applications

16

What Are Classes and Namespaces?

A class is essentially a blueprint that defines the characteristics of an entity

A namespace represents a logical collection of classes



17

The Structure of a Console Application

```
using System;

namespace MyFirstApplication
{
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Bring **System** namespace into scope

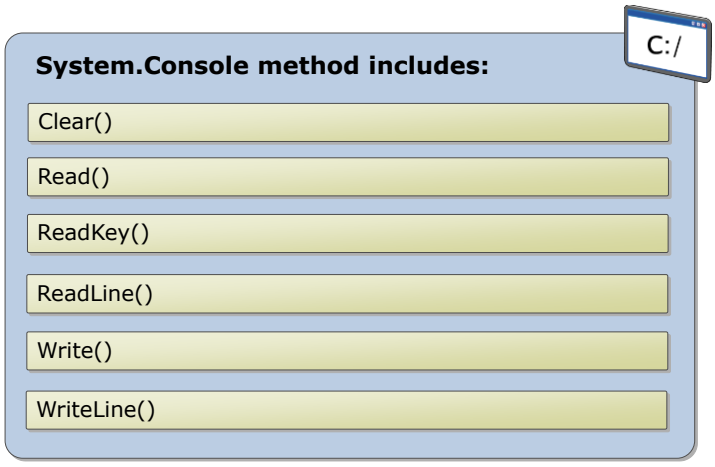
Namespace declaration

Program class declaration

Main method declaration

18

Performing Input and Output by Using a Console Application




```
using System;
...
Console.WriteLine("Hello there!“);
```

19

Best Practices for Commenting C# Applications

- Begin procedures by using a comment block
- In longer procedures, use comments to break up units of work
- When you declare variables, use a comment to indicate how the variable will be used
- When you write a decision structure, use a comment to indicate how the decision is made and what it implies

```
// This is a comment on a separate line.
string message = "Hello there!"; // This is an inline comment.
```

20

Lesson 4: Debugging Applications by Using Visual Studio

- Debugging in Visual Studio
- Using Breakpoints
- Stepping Through and Over Code
- Using the Debug Windows

21

Debugging in Visual Studio

Debugging is an essential part of application development

Visual Studio provides several tools to help you debug code

22

Using Breakpoints

When you run an application in Debug mode, you can pause execution and enter break mode

Visual Studio enables you to:

- ✓ Locate a specific line of code and set a breakpoint
- ✓ Locate a breakpoint and disable it
- ✓ Locate a breakpoint and remove it

23

MICROSOFT OFFICIAL COURSE

Module 2

Using C# Programming Constructs

24

Module Overview

- Declaring Variables and Assigning Values
- Using Expressions and Operators
- Creating and Using Arrays
- Using Decision Statements
- Using Iteration Statements

25

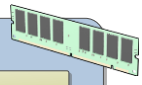
Lesson 1: Declaring Variables and Assigning Values

- What Are Variables?
- What Are Data Types?
- Declaring and Assigning Variables
- What Is Variable Scope?
- Converting a Value to a Different Data Type
- Read-Only Variables and Constants

26

What Are Variables?

Variables store values required by the application in temporary memory locations



Variables have the following facets:

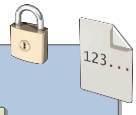
- Name
- Address
- Data type
- Value
- Scope
- Lifetime

27

What Are Data Types?

C# is type-safe language

The compiler guarantees that values stored in variables are always of the appropriate type



Data types include:

int	**char**
long	**bool**
float	**DateTime**
double	**string**
decimal	

28

Declaring and Assigning Variables

Before you can use a variable, you must declare it

```
DataType variableName;
…
DataType variableName1, variableName2;
…
DataType variableName = new DataType();
```

After you declare a variable, you can assign a value to it

```
variableName = Value;
…
DataType variableName = Value;
```

NOTE: Variable name is known as an identifier. Identifiers must:

- Only contain letters, digits, and underscore characters
- Start with a letter or an underscore
- Not be one of the keywords that C# reserves for its own use

29

What Is Variable Scope?

Block scope

```
if (length > 10)
{
    int area = length * length;
}
```

Procedure scope

```
void ShowName()
{
    string name = "Bob";
}
```

Class scope

```
private string message;

void SetString()
{
    message = "Hello World!";
}
```

Namespace scope

```
public class CreateMessage
{
  public string message
    = "Hello";
}

public class DisplayMessage
{
   public void ShowMessage()
   {
       CreateMessage newMessage
         = new CreateMessage();
       MessageBox.Show(
         newMessage.message);

   }
}
```

30

Converting a Value to a Different Data Type

Implicit conversion

Automatically performed by the common language runtime

```
int a = 4;
long b;
b = a;        // Implicit conversion of int to long
```

Explicit conversion

May require you to write code to perform the conversion

```
DataType variableName1 = (castDataType) variableName2;
…
int count = Convert.ToInt32("1234");
...
int number = 0;
if (int.TryParse("1234", out number)) {// Conversion succeeded }
```

31

Read-Only Variables and Constants

Read-only variables

Declared with the **readonly** keyword
Initialized at run time

```
readonly string currentDateTime = DateTime.Now.ToString();
```

Constants

Declared with the **const** keyword
Initialized at compile time

```
const double PI = 3.14159;
int radius = 5;
double area = PI * radius * radius;
double circumference = 2 * PI * radius;
```

32

Lesson 2: Using Expressions and Operators

- What Is an Expression?
- What Are Operators?
- Specifying Operator Precedence
- Best Practices for Performing String Concatenation

33

What Is an Expression?

Expressions are the fundamental constructs that you use to evaluate and manipulate data

```
a + 1

(a + b) / 2

"Answer: " + c.ToString()

b * System.Math.Tan(theta)
```

34

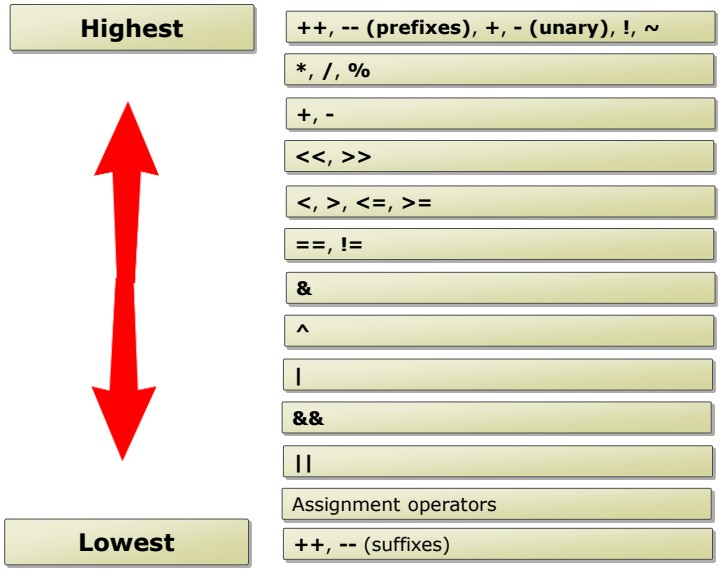
What Are Operators?

Operators are character sequences that you use to define operations that are to be performed on operands

Arithmetic **+, -, *, /, %**	Assignment **=, +=, -=, *=, /=**
Increment, decrement **++, --**	Bit shift **<<, >>**
Comparison **==, !=, <, >, <=, >=, is**	Type information **sizeof, typeof**
String concatenation **+**	Delegate concatenation **+, -**
Logical/bitwise **&, |, ^, !, ~, &&, ||**	Overflow **checked, unchecked**
Indexing **[ ]**	Indirection, Address ***, ->, [ ], &**
Casting **( ), as**	Conditional (ternary operator) **?:**

35

Specifying Operator Precedence



36

Best Practices for Performing String Concatenation

Concatenating multiple strings in C# is simple to achieve by using the **+** operator

```
string address = "23";
address = address + ", Oxford Street";
address = address + ", Thornbury";
```

This is considered bad practice because strings are immutable

A better approach is to use the **StringBuilder** class

```
StringBuilder address = new StringBuilder();

address.Append("23");
address.Append(", Oxford Street");
address.Append(", Thornbury");

string concatenatedAddress = address.ToString();
```

37

Lesson 3: Creating and Using Arrays

- What Is an Array?
- Creating and Initializing Arrays
- Common Properties and Methods Exposed by Arrays
- Accessing Data in an Array

38

What Is an Array?

An array is a sequence of elements that are grouped together

Array features include:

- Every element in the array contains a value
- Arrays are zero-indexed
- The length of an array is the total number of elements it can contain
- The lower bound of an array is the index of its first element
- Arrays can be single-dimensional, multidimensional, or jagged
- The rank of an array is the number of dimensions in the array

39

Creating and Initializing Arrays

An array can have more than one dimension

Single

```
Type[] arrayName = new Type[ Size ];
```

Multiple

```
Type[ , ] arrayName = new Type[ Size1, Size2];
```

Jagged

```
Type [][] JaggedArray = new Type[size][];
```

40

Common Properties and Methods Exposed by Arrays

Length

```
int[] oldNumbers = { 1, 2, 3, 4, 5 };
int numberCount = oldNumbers.Length;
```

Rank

```
int[] oldNumbers = { 1, 2, 3, 4, 5 };
int rank = oldNumbers.Rank;
```

CopyTo()

```
int[] oldNumbers = { 1, 2, 3, 4, 5 };
int[] newNumbers = new
    int[oldNumbers.Length];
oldNumbers.CopyTo(newNumbers, 0);
```

Sort()

```
int[] oldNumbers = { 5, 2, 1, 3, 4 };
Array.Sort(oldNumbers);
```

41

Accessing Data in an Array

Elements are accessed from 0 to N-1

Accessing specific elements

```
int[] oldNumbers = { 1, 2, 3, 4, 5 };
int number = oldNumbers[2];
// OR
object number = oldNumbers.GetValue(2);
```

Iterating through all elements

```
int[] oldNumbers = { 1, 2, 3, 4, 5 };
...
for (int i = 0; i < oldNumbers.Length; i++)
{
    int number= oldNumbers[i];
}
// OR
foreach (int number in oldNumbers) { ... }
```

42

Lesson 4: Using Decision Statements

- Using One-Way If Statements
- Using Either-Or If Statements
- Using Multiple-Outcome If Statements
- Using the Switch Statement
- Guidelines for Choosing a Decision Construct

Using One-Way If Statements

Syntax

```
if ([condition]) [code to execute]

// OR

if ([condition])
{
    [code to execute if condition is true]
}
```

Conditional operators:
OR represented by **||** **AND** represented by **&&**

Example

```
if ((percent >= 0) && (percent <= 100))
{
    // Add code to execute if a is greater than 50 here.
}
```

Using Either-Or If Statements

Provide an additional code block to execute if *[condition]* evaluates to a Boolean false value

Example

```
if (a > 50)
{
    // Greater than 50 here
}
else
{
    // less than or equal to 50 here
}

// OR

string carColor = "green";

string response = (carColor == "red") ?
    "You have a red car" :
    "You do not have a red car";
```

45

Using Multiple-Outcome If Statements

You can combine several **if** statements to create a multiple-outcome statement

Example

```
if (a > 50)
{
    // Add code to execute if a is greater than 50 here.
}
else if (a > 10)
{
    // Add code to execute if a is greater than 10 and less than
    // or equal to 50 here.
}
else
{
    // Add code to execute if a is less than or equal to 50 here.
}
```

46

Using the Switch Statement

The **switch** statement enables you to execute one of several blocks of code depending on the value of a variable or expression

Example

```
switch (a)
{
    case 0:
        // Executed if a is 0.
        break;

    case 1:
    case 2:
    case 3:
        // Executed if a is 1, 2, or 3.
        break;

    default:
        // Executed if a is any other value.
        break;
}
```

47

Guidelines for Choosing a Decision Construct

Use an **if** structure when you have a single condition that controls the execution of a single block of code

Use an **if/else** structure when you have a single condition that controls the execution of one of two blocks of code

Use an **if/elseif/else** structure to run one of several blocks of code based on conditions that involve several variables

Use a nested **if** structure to perform more complicated analysis of conditions that involve several variables

Use a **switch** statement to perform an action based on the possible values of a single variable

48

Lesson 5: Using Iteration Statements

- Types of Iteration Statements
- Using the While Statement
- Using the Do Statement
- Using the For Statement
- Break and Continue Statements

49

Types of Iteration Statements

Iteration statements include:

while

A **while** loop enables you to execute a block of code zero or more times

do

A **do** loop enables you to execute a block of code one or more times

for

A **for** loop enables you to execute code repeatedly a set number of times

50

Using the While Statement

The syntax of a while loop contains:

- The **while** keyword to define the **while** loop
- A condition that is tested at the start of each iteration
- A block of code to execute for each iteration

Example

```
double balance = 100D;
double rate = 2.5D;
double targetBalance = 1000D;
int years = 0;
while (balance <= targetBalance)
{
    balance *= (rate / 100) + 1;
    years += 1;
}
```

51

Using the Do Statement

The syntax of a do loop contains:

- The **do** keyword to define the **do** loop
- A block of code to execute for each iteration
- A condition that is tested at the end of each iteration

Example

```
string userInput = "";
do
{
    userInput = GetUserInput();
    if (userInput.Length < 5)
    {
        // You must enter at least 5 characters.
    }
} while (userInput.Length < 5);
```

52

Using the For Statement

The syntax of a for loop contains:

- The **for** keyword to define the **for** loop
- The loop specification (counter, starting value, limit, modifier)
- A block of code to execute for each iteration

Example

```
for (int i = 0; i < 10; i++)
{
    // Code to loop, which can use i.
}
```

53

Break and Continue Statements

Break statement

```
while (oldNumbers.Length > count)
{
    if (oldNumbers[count] == 5)
    {
        break;
    }
    count++;
}
```

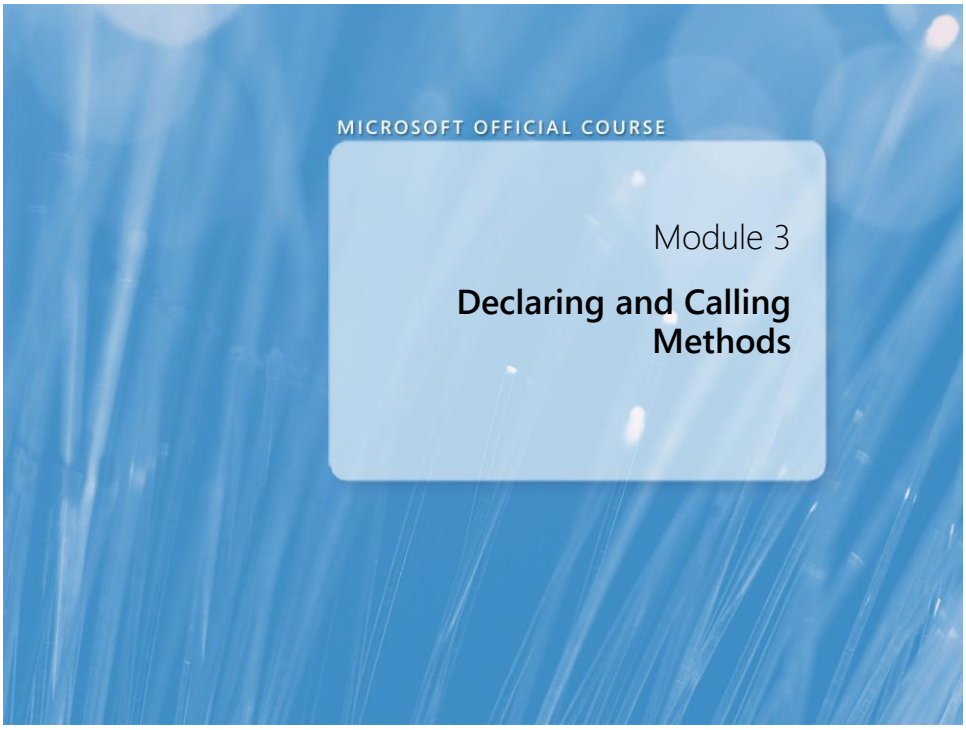
Enables you to exit the loop and skip to the next line of code

Continue statement

```
while (oldNumbers.Length > count)
{
    if (oldNumbers[count] == 5)
    {
        continue;
    }
    // Code that won't be hit
    count++;
}
```

Enables you to skip the remaining code in the current iteration, test the condition, and then start the next iteration of the loop

54



55

Module Overview

- Defining and Invoking Methods
- Specifying Optional Parameters and Output Parameters

56

Lesson 1: Defining and Invoking Methods

- What Is a Method?
- Creating a Method
- Calling a Method
- Creating and Calling Overloaded Methods
- Using Parameter Arrays
- Refactoring Code into a Method
- Testing a Method
- Demonstration: Refactoring and Testing a Method

57

What Is a Method?

A method is a class member that contains a code block that represents an action:

- All executable code belongs to a method
- A C# application must have at least one method
- Methods can be defined for private use by a type, or for public use by other types

The **Main** method defines the starting point for an application

C# supports instance and static methods

Instance method

```
int count = 99;
string strCount =
    count.ToString();
```

Static method

```
string strCount = "99";
count =
   Convert.ToInt32(strCount);
```

58

Creating a Method

A method contains:
- The method specification (access, return type, and signature)
- The method body (code)

Method signature:

1. **Name** (Pascal case)
2. **Parameters** (camel case)

Each method signature must be unique in a programming type

Example

```
bool LockReport(string userName)
{
    bool success = false;
    // Perform some processing here.
    return success;
}
```

59

Calling a Method

To call a method:
- Specify the method name
- Provide an argument for each parameter
- Handle the return value

Example method

```
bool LockReport(string reportName, string userName)
{
    ...;
}
```

Method call

```
bool isReportLocked =
    LockReport("Medical Report", "Don Hall");
```

Note: Arguments are evaluated in left-to-right order

60

Creating and Calling Overloaded Methods

An overloaded method:
- Has the same name as an existing method
- *Should* perform the same operation as the existing method
- Uses different parameters to perform the operation

The compiler invokes the version that is appropriate to the arguments that are specified when the method is called

Example

```
void Deposit(decimal amount)
{
    _balance += amount;
}

void Deposit(int dollars, int cents)
{
    _balance += dollars + (cents / 100.0m);
}
```

The signature of the **Deposit** method is different for each implementation

61

Using Parameter Arrays

Overloading may not be possible or appropriate if a method can take a variable number of arguments

Example

```
int Add(params int[] data)
{
    int sum = 0;
    for (int i = 0; i < data.Length; i++)
    {
        sum += data[i];
    }
    return sum;
}
```

Method call

```
int sum = Add(99, 2, 55, -26);
```

62

Refactoring Code into a Method

1. Select the code that you want to extract to a method, right-click, point to **Refactor**, and then click **Extract Method**
2. In the **Extract Method** dialog box, in the **New method name** box, type a name for the method, and then click **OK**

Example of refactored output

```
LogMessage(messageContents, filePath);
...
private void LogMessage(string messageContents, string filePath)
{
    ...
}
```

63

Testing a Method

```
int Calculate(int operandOne, int operandTwo)
{
    int result = 0;

    // Perform some calculation.

    return result;
}
```

Create Unit Test Wizard

```
[TestMethod()]
public void CalculateTest()
{
    Program target = new Program();
    int operandOne = 0;
    int operandTwo = 0;
    int expected = 0;
    int actual;
    actual = target.Calculate(operandOne, operandTwo);
    Assert.AreEqual(expected, actual);
    ...
}
```

64

Lesson 2: Specifying Optional Parameters and Output Parameters

- What Are Optional Parameters?
- Calling a Method by Using Named Arguments
- What Are Output Parameters?

65

What Are Optional Parameters?

Optional parameters enable you to define a method and provide default values for the parameters in the parameter list:

- They are useful for interoperating with other technologies that support optional parameters
- They can provide an alternative implementation where overloading is not appropriate

Example

```
void MyMethod(
    int intData, float floatData, int moreIntData = 99)
{
    ...
}
```

```
// Arguments provided for all three parameters
MyMethod(1o, 123.45, 99);

// Arguments provided for 1st two parameters only
MyMethod(100, 54.321);
```

66

Calling a Method by Using Named Arguments

Specify parameters by name:
- Arguments can occur in any order
- Omitted arguments are assigned default values if the corresponding parameter is optional

Example

```
void MyMethod(
  int intData, float floatData = 101.1F, int moreIntData = 99)
{
    ...
}
```

```
MyMethod(moreIntData: 100, intData: 10);
// floatData is assigned default value
```

67

What Are Output Parameters?

Output parameters enable you to pass a value back from a method to an argument:
- Use the **out** keyword in the parameter list

```
void MyMethod(int first, double second, out int data)
{
    ...
    data = 99;
}
```

Call the method with a variable specified as an **out** argument

```
int value;
MyMethod(10, 101.1F, out value);
// value = 99
```

68

69

Module Overview

- Handling Exceptions
- Raising Exceptions

70

Lesson 1: Handling Exceptions

- What Is an Exception?
- Using a Try/Catch Block
- Using Exception Properties
- Using a Finally Block
- Using the Checked and Unchecked Keywords
- Demonstration: Raising Exceptions in Visual Studio

What Is an Exception?

An exception is an indication of an error or exceptional condition, such as trying to open a file that does not exist

When a method throws an exception, the calling code must be prepared to detect and handle the exception

If the calling code cannot handle the exception, the exception is automatically propagated to the code that invoked the calling code

The exception is propagated until a section of code handles the exception

If no code handles the exception, the runtime will report an unhandled exception and the application will crash

Using a Try/Catch Block

```
try
{
    // Try block.
}
catch (DivideByZeroException ex)
{
    // Catch block, can access DivideByZeroException
    // exception in ex.
}
catch (Exception ex)
{
    // Catch block, can access exception in ex.
}
```

Logic that might cause an exception

Logic to run in the event of a **DivideByZeroException** exception

Logic to run in the event of any other exceptions

You can also nest try/catch blocks

Exception types can implement a hierarchy of exceptions

73

Using Exception Properties

Common properties include:

- **Message**
- **Source**
- **StackTrace**
- **TargetSite**
- **InnerException**
- **HelpLink**
- **Data**

```
try
{
    // Try block.
}
catch (DivideByZeroException ex)
{
    Console.WriteLine(ex.Message);
}
```

Get the message associated with the exception

74

Using a Finally Block

Enables you to release resources and specify code that will always run, whether an exception occurs or not

```
try
{
    OpenFile("MyFile"); // Open a file
    WriteToFile(...);   // Write some data to the file
}
catch (IOException ex)
{
    MessageBox.Show(ex.Message);
}
finally
{
    CloseFile("MyFile"); // Close the file
}
```

Logic that will always run

75

Using the Checked and Unchecked Keywords

C# applications run with integer numeric overflow checking disabled by default. You can change this project setting

You can control overflow checking for specific code by using the **checked** and **unchecked** keywords

```
checked
{
    int x = ...;
    int y = ...;
    int z = ...;
    ...
}
```

```
unchecked
{
    int x = ...;
    int y = ...;
    int z = ...;
    ...
}
```

checked and **unchecked** blocks

```
...
int z = checked(x* y);
...
```

```
...
int z = unchecked(x* y);
...
```

checked and **unchecked** operators

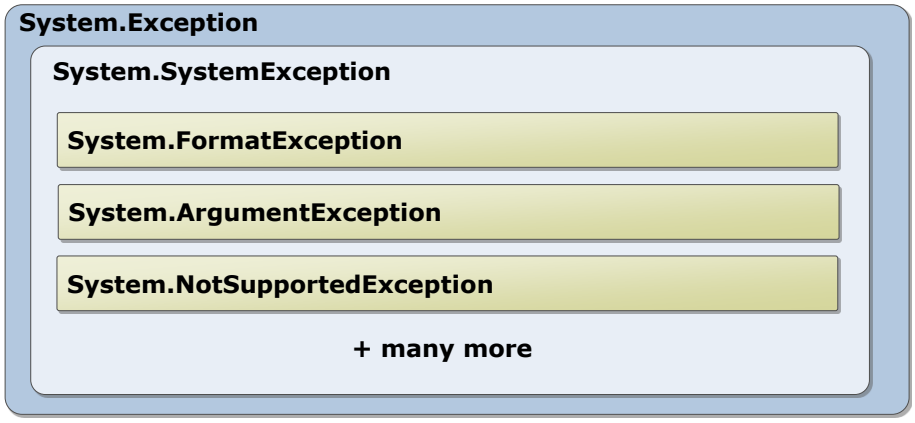
76

Lesson 2: Raising Exceptions

- Creating an Exception Object
- Throwing an Exception
- Best Practices for Handling and Raising Exceptions

77

Creating an Exception Object




```
catch (Exception e)
{
    FormatException ex =
        new FormatException("Argument has the wrong format", e);
}
```

78

Throwing an Exception

throw keyword

```
throw [exception object];
```

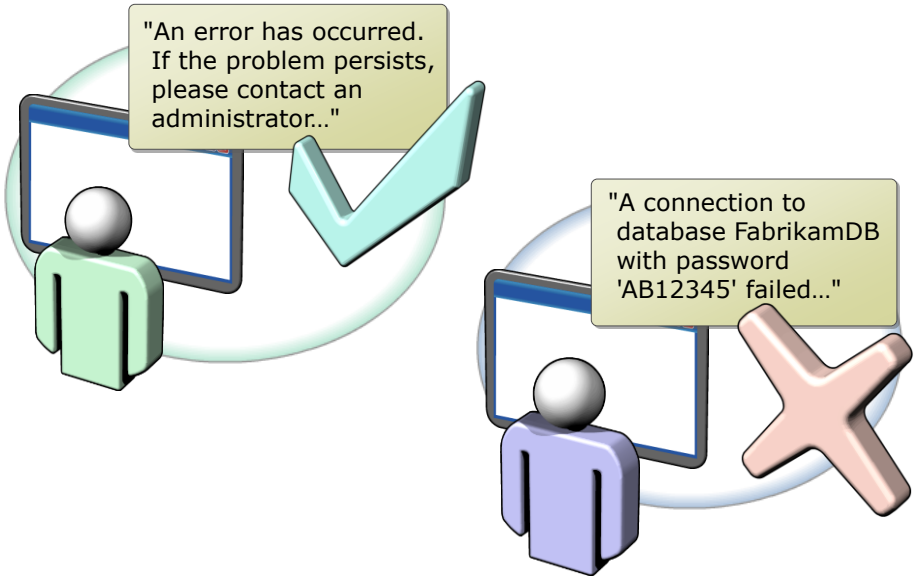
Exception object declaration

Example

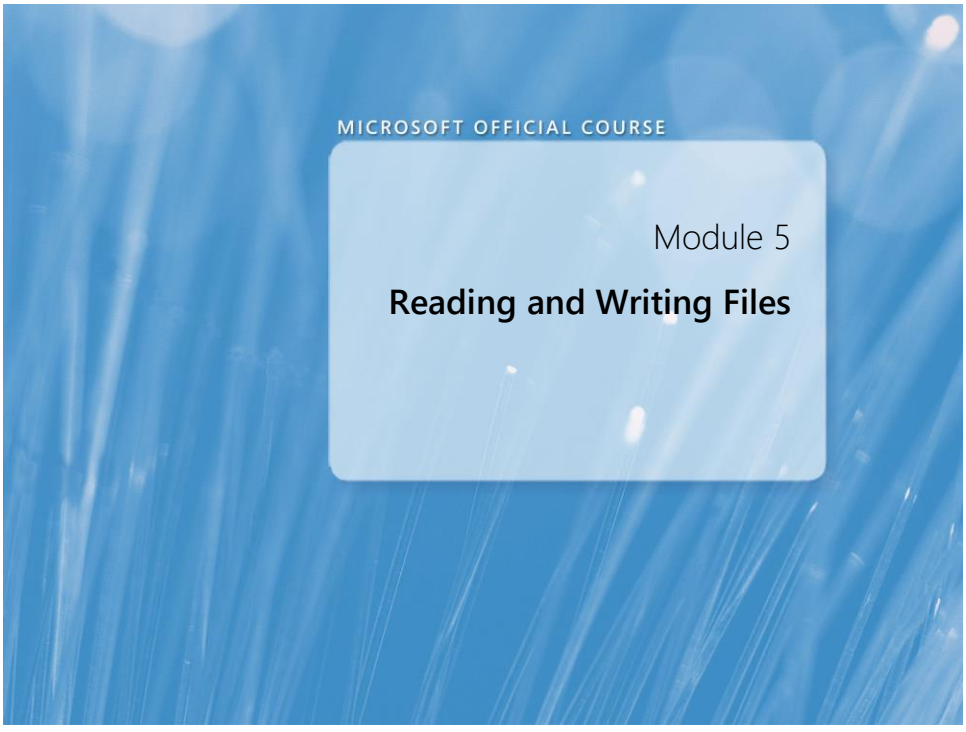
```
public int GetIntegerRoot(int operand)
{
    double root = Math.Sqrt(operand);
    if (root != (int)root)
    {
        throw new ArgumentException("No integer root found.");
    }
    return (int)root;
}
```

79

Best Practices for Handling and Raising Exceptions



80



81

Module Overview

- Accessing the File System
- Reading and Writing Files by Using Streams

82

Lesson 1: Accessing the File System

- Manipulating Files
- Reading from and Writing to Files
- Manipulating Directories
- Manipulating Paths
- Using the Common File System Dialog Boxes

83

Manipulating Files

Interacting with files is a common requirement for many applications

The **System.IO** namespace contains many classes that simplify interactions with the file system, such as the **File** and **FileInfo** classes

The **File** class includes:

- **Copy()**
- **Create()**
- **Delete()**
- **Exists()**

The **FileInfo** class includes:

- **CopyTo()**
- **Delete()**
- **Length**
- **Open()**

84

Reading from and Writing to Files

Reading data from files

```
string filePath = "myFile.txt";
...
byte[] data = File.ReadAllBytes(filePath); // Binary data.
string[] lines = File.ReadAllLines(filePath); // Lines from file.
string data = File.ReadAllText(filePath); // Entire file.
```

Writing data to files

```
string filePath = "myFile.txt";
...
string[] fileLines = {"Line 1", "Line 2", "Line 3"};
File.AppendAllLines(filePath, fileLines); // Append lines.
File.WriteAllLines(filePath, fileLines); // Write lines to new file.
...
string fileContents = "I am writing this text to a file ...“;
File.AppendAllText(filePath, fileContents); // Append all text..
File.WriteAllText(filePath, fileContents); // Write all lines to new
file.
```

85

Manipulating Directories

The **System.IO** namespace contains the **Directory** and **DirectoryInfo** classes to help simplify interactions with directories

Directory class

```
string dirPath = @"C:\Users\Student\MyDirectory";
...
Directory.CreateDirectory(dirPath);
Directory.Delete(dirPath);
string[] dirs = Directory.GetDirectories(dirPath);
string[] files = Directory.GetFiles(dirPath);
```

DirectoryInfo class

```
string dirPath = @"C:\Users\Student\MyDirectory";
DirectoryInfo dir = new DirectoryInfo(dirPath);
...
bool exists = dir.Exists;
DirectoryInfo[] dirs = dir.GetDirectories();
FileInfo[] files = dir.GetFiles();
string fullName = dir.FullName;
```

86

Manipulating Paths

The **System.IO** namespace contains the **Path** class, which can help to create and control path names, independent of the underlying file system

Path class includes:

- **GetDirectoryName()**
- **GetExtension()**
- **GetFileName()**
- **GetFileNameWithoutExtension()**
- **GetRandomFileName()**

87

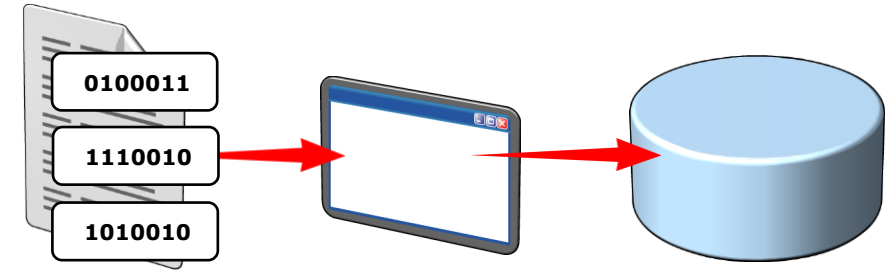
Lesson 2: Reading and Writing Files by Using Streams

- What Are Streams?
- Reading and Writing Binary Data
- Reading and Writing Text
- Reading and Writing Primitive Data Types

88

What Are Streams?

A stream is a mechanism that enables you to manipulate data in manageable chunks



89

Reading and Writing Binary Data

```
FileStream sourceFile = new FileStream(sourceFilePath);
BinaryReader reader = new BinaryReader(sourceFile);
int position = 0;
int length = (int)reader.BaseStream.Length;
byte[] dataCollection = new byte[length];
int returnedByte;
while ((returnedByte = reader.Read()) != -1)
{
    dataCollection[position] = (byte)returnedByte;
    position += sizeof(byte);
}
reader.Close();
sourceFile.Close();
```

BinaryReader class

```
byte[] dataCollection = { 1, 4, 6, 7, 12, 33, 26, 98, 82, 101 };
FileStream destFile = new FileStream(destinationFilePath);
BinaryWriter writer = new BinaryWriter(destFile);
foreach (byte data in dataCollection)
{
    writer.Write(data);
}
writer.Close();
destFile.Close();
```

BinaryWriter class

90

Reading and Writing Text

```
FileStream sourceFile = new FileStream(sourceFilePath);
StreamReader reader = new StreamReader(sourceFile);
StringBuilder fileContents = new StringBuilder();
while (reader.Peek() != -1)
{
    fileContents.Append((char)reader.Read());
}
string data = fileContents.ToString();
reader.Close();
sourceFile.Close();
```

StreamReader class

```
FileStream destFile = new FileStream("...");
StreamWriter writer = new StreamWriter(destFile);

writer.WriteLine("Hello, this will be written to the file");

writer.Close();
destFile.Close();
```

StreamWriter class

91

Reading and Writing Primitive Data Types

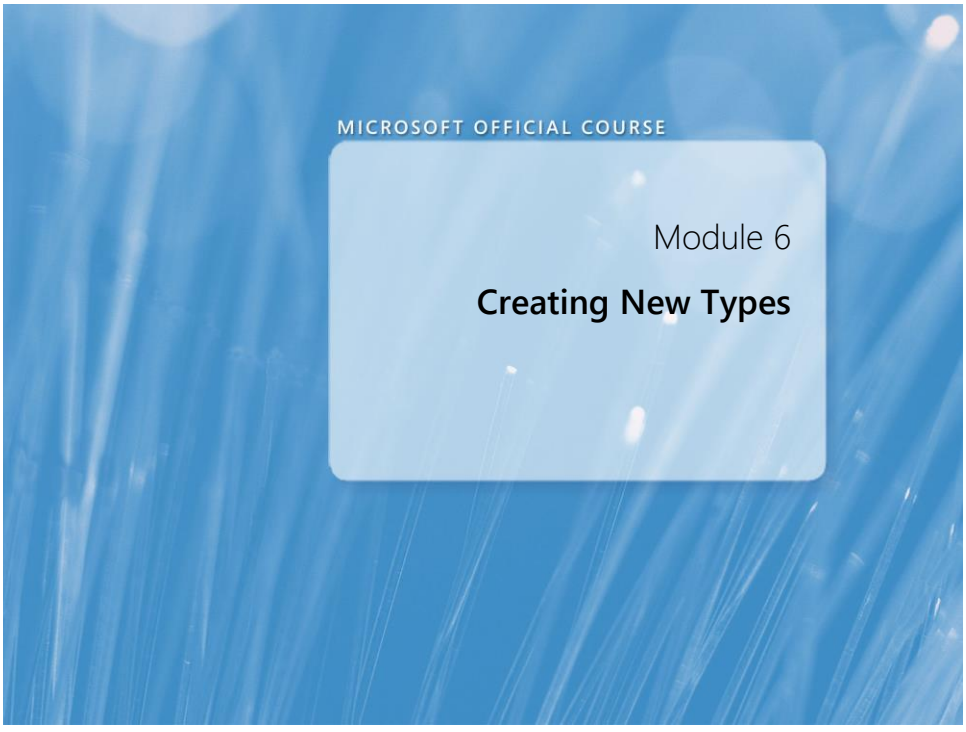
```
bool boolValue = reader.ReadBoolean();
byte byteValue = reader.ReadByte();
byte[] byteArrayValue = reader.ReadBytes(4);
char charValue = reader.ReadChar();
...
```

BinaryReader class

```
bool boolValue = true;
writer.Write(boolValue);

byte byteValue = 1;
writer.Write(byteValue);

byte[] byteArrayValue = { 1, 4, 6, 8 };
writer.Write(byteArrayValue);

char charValue = 'a';
writer.Write(charValue);
...
```

BinaryWriter class

92



93

Module Overview

- Creating and Using Enumerations
- Creating and Using Classes
- Creating and Using Structures
- Comparing References to Values

94

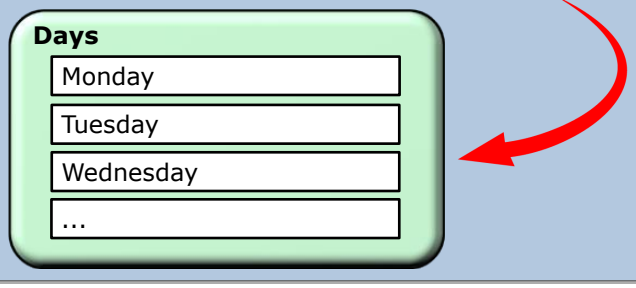
Lesson 1: Creating and Using Enumerations

- What Are Enumerations?
- Creating New Enum Types
- Initializing and Assigning Enum Variables

95

What Are Enumerations?

An enumeration type specifies a set of named constants



Benefits include:

- Code is easier to maintain because you assign only expected values to your variables
- Code is easier to read because you assign easily identifiable names to your values
- Code is easier to write because IntelliSense displays a list of the possible values that you can use

96

Creating New Enum Types

Define enumerations with the **enum** keyword

```
enum Name : DataType { Value1, Value2 . . . };
```

You can define enumerations in namespaces and classes, but not in methods

Enumerations values start at 0 by default, unless explicitly stated

```
namespace Fabrikam.CustomEnumerations
{
    enum Days : int { Monday = 1, Tuesday = 2, Wednesday = 3 };
    ...
}
```

97

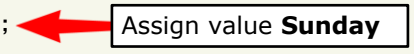
Initializing and Assigning Enum Variables

Instantiate enumeration variables in the same way as any other type

```
[EnumType] variableName = [EnumValue]
```

You can only assign an enumeration a value defined in its definition

```
enum Days : int
{
    Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4,
    Friday = 5, Saturday = 6, Sunday = 7
};
static void Main(string[] args)
{
    Days myDayOff = Days.Sunday;
}
```



You can perform simple operations on an enumeration

```
for (Days dayOfWeek = Days.Monday; dayOfWeek <= Days.Sunday;
    dayOfWeek++)
{
    ...
}
```

98

Lesson 2: Creating and Using Classes

- What Is a Class?
- Adding Members to Classes
- Defining Constructors and Initializing an Object
- Creating Objects
- Accessing Class Members
- Using Partial Classes and Partial Methods

What Is a Class?

A class is a blueprint from which you can create objects

A class defines the characteristics of an object

```
class House
{
   ...
}
```

Class definition with **class** keyword

An object is an instance of a class

myHouse **yourHouse**

Adding Members to Classes

Members define the data and behavior of the class

```
public class Residence
{
    public ResidenceType type;
    public int numberOfBedrooms;
    public bool hasGarage;
    public bool hasGarden;

    public int CalculateSalePrice()
    {
        // Code to calculate the sale value of
        // the residence.
    }

    public int CalculateRebuildingCost()
    {
        // Code to calculate the rebuilding costs
        // of the residence.
    }
}
```

Fields

Methods

101

Defining Constructors and Initializing an Object

A constructor is a special method that the runtime invokes implicitly

```
public class Residence
{
    public Residence(ResidenceType type, int numberOfBedrooms)
    {
    }
    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage)
    {
    }
    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage, bool hasGarden)
    {
    }
}
```

Three constructors

If you do not initialize a field in a class, it is assigned its default value

102

Creating Objects

Objects are initially unassigned

Before you can use a class, you must create an instance of that class

You can create a new instance of a class by using the **new** operator

The **new** operator does two things:

- Causes the CLR to allocate memory for the object
- Invokes a constructor to initialize the object

```
// Create a Flat with 2 bedrooms.
Residence myFlat = new Residence(ResidenceType.Flat, 2);

// Create a House with 3 bedrooms and a garage
Residence myHouse = new Residence(ResidenceType.House, 3, true);

// Create a Bungalow with 2 bedrooms, a garage, and a garden
Residence myBungalow =
   new Residence(ResidenceType.Bungalow, 2, true, true);
```

103

Accessing Class Members

Access object members by using the *instance name* followed by a period

```
InstanceName.MemberName
```

Example

```
// Create a 3 bedroom house.
Residence myHouse = new Residence(ResidenceType.House, 3);

// Indicate that the residence has a garden.
myHouse.hasGarden = true;

// Calculate the market value.
int salePrice = myHouse.CalculateSalePrice();

// Get the rebuilding costs.
int rebuildCost = myHouse.CalculateRebuildingCost();
```

104

Using Partial Classes and Partial Methods

Split a class definition across multiple source files

Prefix the class and method declarations with the **partial** keyword

```
public partial class Residence
{
    partial void SaleResidence(string name);
}
```

Definition of **SaleResidence** method

```
public partial class Residence
{
    partial void SaleResidence(string name)
    {
        //Logic goes here.
    }
}
```

Implementation of **SaleResidence** method

All parts of the partial class and method must be available during compilation

Compiled into a single entity

105

Lesson 3: Creating and Using Structures

- What Are Structures?
- Defining and Using a Structure
- Initializing a Structure

106

What Are Structures?

All structures are value types

System.Byte	System.Int64	System.Decimal
System.Int16	System.Single	System.Boolean
System.Int32	System.Double	System.Char

The data in structures is stored on the stack

Use structures to model items that contain small amounts of data

Structures can contain fields and methods just like a class

```
int x = 99;
string xAsString = x.ToString();
```

107

Defining and Using a Structure

Use the **struct** keyword to declare a structure

Define members in a structure in the same manner as a class

```
struct Currency
{
    public string currencyCode;
    public string currencySymbol;
    public int fractionDigits
}
```

Structures are automatically allocated memory on the stack

```
Currency unitedStatesCurrency;

unitedStatesCurrency.currencyCode = "USD";
unitedStatesCurrency.currencySymbol = "$";
unitedStatesCurrency.fractionDigits = 2;
```

You do not have to use the **new** operator when you create an instance of a structure

108

Initializing a Structure

The **new** operator only initializes fields in the structure

```
struct Currency
{
    public string currencyCode;
    public string currencySymbol;
    public int fractionDigits
    public Currency(string code, string symbol)
    {
        this.currencyCode = code;
        this.currencySymbol = symbol;
        this.fractionDigits = 2;
    }
};
...
Currency unitedKindgdomCurrency = new Currency("GBP", "£");
```

Always use a constructor to guarantee that the structure and fields are initialized

109

Lesson 4: Comparing References to Values

- Comparing Reference Types to Value Types
- Passing a Value Type by Reference into a Method
- Boxing and Unboxing
- Nullable Types

110

Comparing Reference Types to Value Types

Reference types (reference on stack, value on heap)

- When you assign a reference, you simply refer to an object in memory
- If you assign the same reference to two different variables, both variables refer to the same object
- If you change the data in the object, the changes will be reflected in all variables that reference that object

Value types (value on stack)

- When you copy a value type, the variables do not refer to the same object
- If you change the data in either variable, the changes will not be reflected in other copies

111

Passing a Value Type by Reference into a Method

```
static void Main(string[] args)
{
    int myInt = 1005;
    ChangeInput(myInt);
}

static void ChangeInput(int input)
{
    input = 29910;
}
```

myInt still equals 1005

```
static void Main(string[] args)
{
    int myInt = 1005;
    ChangeInput(ref myInt);
}

static void ChangeInput(ref int input)
{
    input = 29910;
}
```

myInt now equals 29910

112

Nullable Types

You can set a reference variable to null to indicate that it has not been initialized

The null value is itself a reference, and there is no corresponding value for value types to indicate that they are uninitialized

```
Currency myCurrency = null; // illegal
```

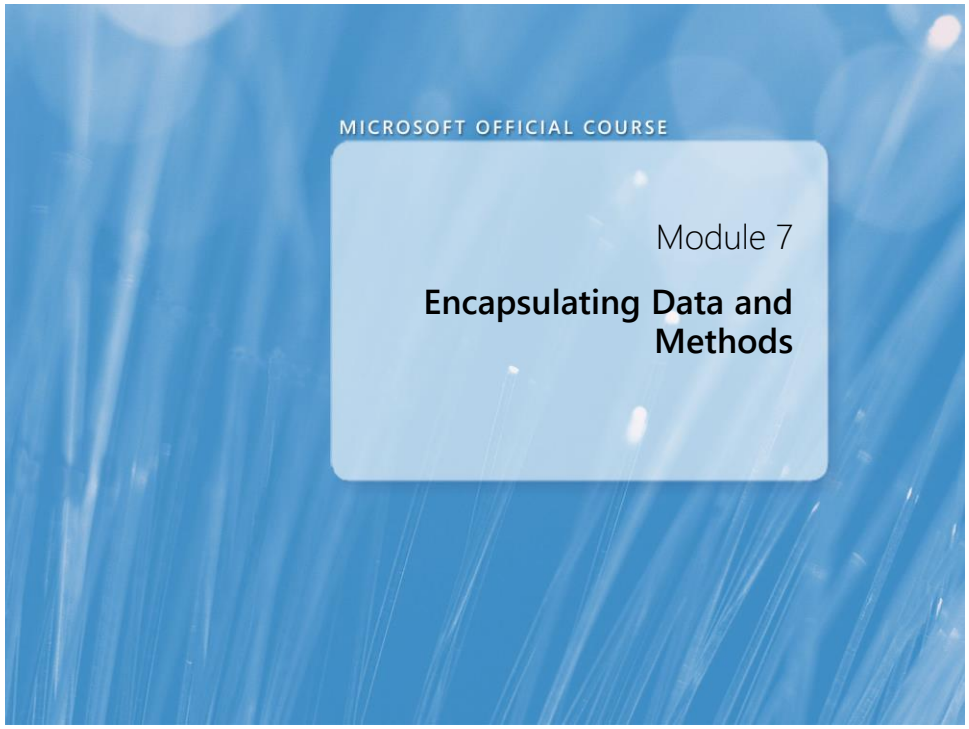
You use a question mark (?) to indicate that a value type is nullable

```
Currency? myCurrency = null; // legal
if (myCurrency == null)
{
}
```

Nullable types expose the **HasValue** and **Value** properties

```
Currency? myCurrency = null;
if (myCurrency.HasValue)
{
  Console.WriteLine(myCurrency.Value);
}
```

113



114

Module Overview

- Controlling Visibility of Type Members
- Sharing Methods and Data

115

Lesson 1: Controlling Visibility of Type Members

- What Is Encapsulation?
- Comparing Private and Public Members
- Comparing Internal and Public Types

116

What Is Encapsulation?

Definition

- An essential object-oriented principle
- Hides internal data and algorithms
- Provides a well-defined public operation

Benefits

- Makes external code simpler and more consistent
- Enables you to change implementation details later

117

Comparing Private and Public Members

private Least permissive access level

```
class Sales
{
    private double monthlyProfit;
    private void SetMonthlyProfit(double monthlyProfit)
    {
        this.monthlyProfit = monthlyProfit;
    }
}
```

Only accessible from within the **Sales** class

public Most permissive access level

```
class Sales
{
    private double monthlyProfit;
    public void SetMonthlyProfit(double monthlyProfit)
    {
        this.monthlyProfit = monthlyProfit;
    }
}
```

Accessible to any other type

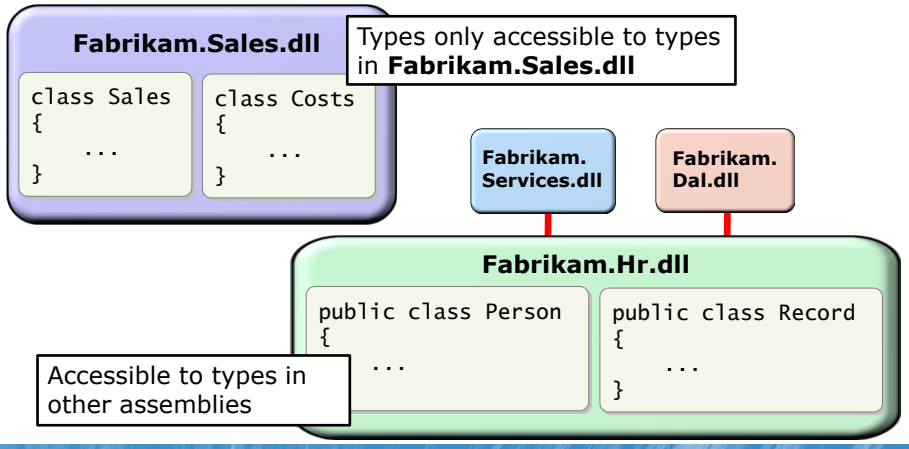
118

Comparing Internal and Public Types

Internal is the default access modifier for types

Internal permits access only to types in the same assembly

Public permits access to other types in other assemblies



119

Lesson 2: Sharing Methods and Data

- Creating and Using Static Fields
- Creating and Using Static Methods
- Creating Static Types and Using Static Constructors
- Creating and Using Extension Methods

120

Creating and Using Static Fields

Instance fields contain data pertinent to a specific instance of a type

Static fields contain data pertinent to the type itself

```
class Sales
{
    public static double salesTaxPercentage = 20;
}
```

To access a static field, you use the name of the class, followed by a period, followed by the name of the field

```
Sales.salesTaxPercentage = 32;
```

121

Creating and Using Static Methods

Static methods provide functionality pertinent to the type itself and not an instance of the type

You can use static methods to implement utility classes, like the **File** class in the **System.IO** namespace

```
class Sales
{
    public static double GetMonthlySalesTax(double
        monthlyProfit)
    {
        ...
    }
}
```

You do not need to create an instance of the type to use static methods

```
double monthlySalesTax = Sales.GetMonthlySalesTax(34267);
```

122

Creating Static Types and Using Static Constructors

You can declare types as static by using the **static** modifier

Static types can only contain static members

```
static class Sales
{
}
```

Defined with the **static** modifier

Both static and instance types can have constructors

You can explicitly invoke instance constructors with the **new** keyword

Static constructors are invoked implicitly by the CLR

```
static class Sales
{
    static Sales()
    {
    }
}
```

Defined with the **static** modifier

Defined without any parameters

Defined without an access modifier

123

MICROSOFT OFFICIAL COURSE

Module 8

Inheriting from Classes and Implementing Interfaces

124

Module Overview

- Using Inheritance to Define New Reference Types
- Defining and Implementing Interfaces
- Defining Abstract Classes

125

Lesson 1: Using Inheritance to Define New Reference Types

- What Is Inheritance?
- The .NET Framework Inheritance Hierarchy
- Overriding and Hiding Methods
- Calling Methods and Constructors in a Base Class
- Assigning and Referencing Classes in an Inheritance Hierarchy
- Understanding Polymorphism
- Defining Sealed Classes and Methods
- Demonstration: Using Inheritance to Construct New Reference Types

126

What Is Inheritance?

Inheritance enables you to define new types based on existing types:

- For example, **Manager** and **ManualWorker** classes might inherit from an **Employee** class
- Fields and methods in the **Employee** class are inherited by **Manager** and **ManualWorker**
- **Manager** and **ManualWorker** can define their own fields and behavior
- Define accessible members as **protected** in the base class

```
// Base class
class Employee
{
    protected string empNum;
    protected string empName;
    protected void DoWork()
    { ... }
}

// Inheriting classes
class Manager : Employee
{
    public void DoManagementWork()
    { ... }
}

class ManualWorker : Employee
{
    public void DoManualWork()
    { ... }
}
```

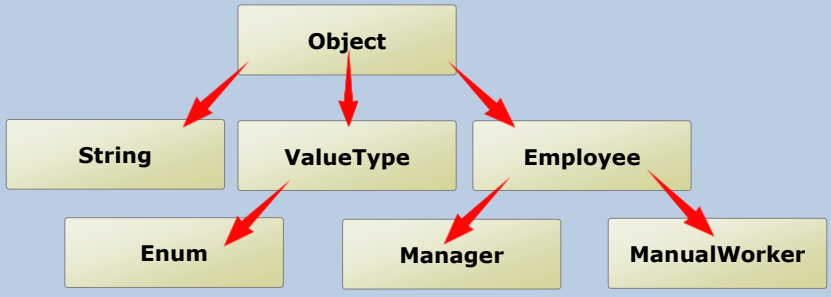
C# supports single inheritance only

127

The .NET Framework Inheritance Hierarchy

All types inherit directly or indirectly from the **System.Object** class

- No need to specify **: Object** in the class definition
- Structs and enums inherit from **ValueType**



You cannot define your own inheritance hierarchy by using structs and enums

128

Overriding and Hiding Methods

Overriding: Replace or extend functionality in a base class with semantically equivalent behavior (intentional)

```
class Employee
{
    protected virtual void
        DoWork()
    { ... }
}

class Manager : Employee
{
    protected override void
        DoWork()
    { ... }
}
```

```
class Employee
{
    protected void DoWork()
    { ... }
}

class Manager : Employee
{
    public new void DoWork()
    { ... }
}
```

Hiding: Replace functionality in a base class with new behavior (possibly an error)

129

Calling Methods and Constructors in a Base Class

Use the **base** keyword

```
class Employee
{
    protected virtual void
        DoWork()
    { ... }
}

class Manager : Employee
{
    protected override void
        DoWork()
    {
        ...
        base.DoWork();
    }
}
```

```
class Employee
{
    protected string empName;
    public Employee(string name)
    { this.empName = name; }
}

class Manager : Employee
{
    protected string empGrade;
    public Manager(string name,
                   string grade)
        : base(name)
    {
        this.empGrade = grade;
    }
}
```

Constructors automatically call the default constructor for the base type unless you specify otherwise

130

Assigning and Referencing Classes in an Inheritance Hierarchy

C# type-checking prevents you from assigning references of one type to variables of a different type …

```
Manager myManager = new
Manager(…);
ManualWorker myWorker =
    myManager;
```

… but you can assign a reference to a different type that is higher up an inheritance hierarchy

```
Manager myManager = new
Manager(…);
Employee myEmployee =
    myManager;
```

Use the **is** and **as** operators to safely assign a reference to a type that is lower down in an inheritance hierarchy

```
Manager myManagerAgain =
     myEmployee as Manager;
```

131

Understanding Polymorphism

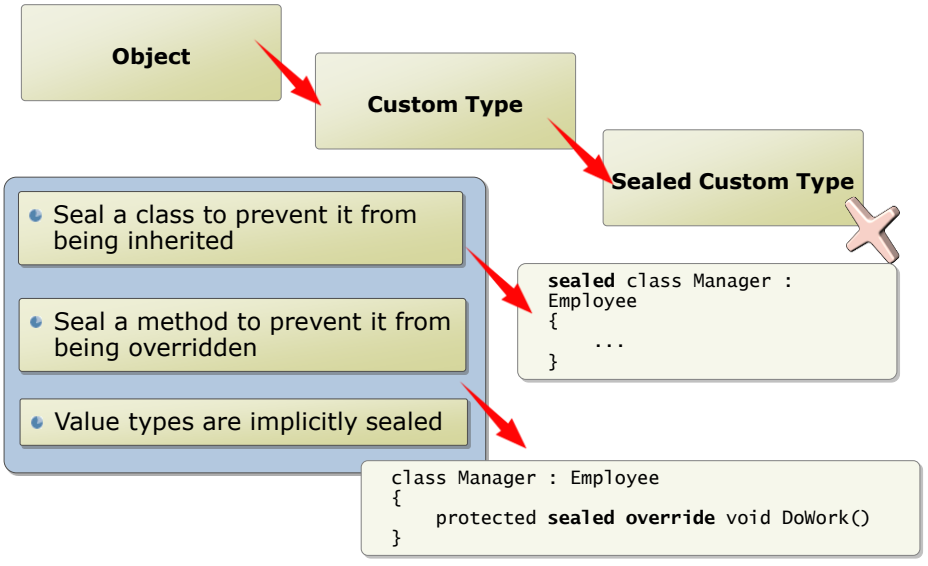
In an inheritance hierarchy, the same statement can call different implementations of the same method

```
Employee myEmployee;
Manager myManager =
    new Manager(...);
ManualWorker myWorker =
    new ManualWorker(...);

myEmployee = myManager;
Console.WriteLine(
myEmployee.GetTypeName());

myEmployee = myWorker;
Console.WriteLine(
myEmployee.GetTypeName());
```

```
class Employee
{
    public virtual string
GetTypeName()
    {
        return "This is an
            Employee";
    }
}

class Manager : Employee
{
    public override string
        GetTypeName()
    {
        return "This is a Manager";
    }
}

class ManualWorker : Employee
{
    // Does not override GetTypeName
}
```

132

Defining Sealed Classes and Methods



133

Lesson 2: Defining and Implementing Interfaces

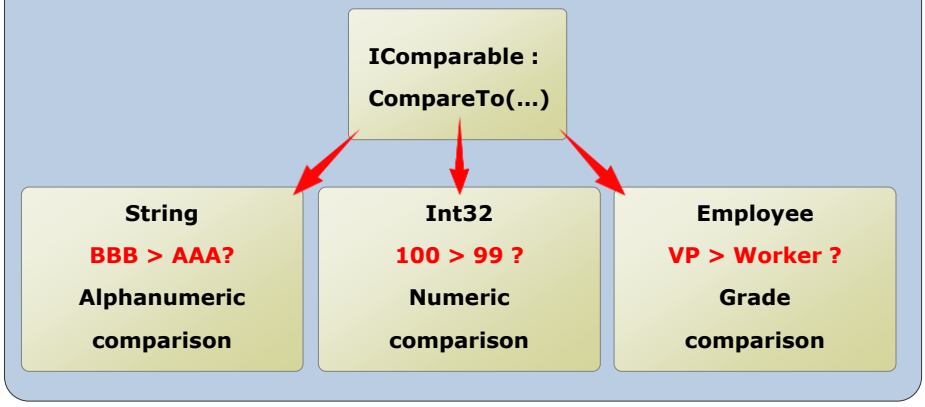
- What Is an Interface?
- Creating and Implementing an Interface
- Referencing an Object Through an Interface
- Explicitly and Implicitly Implementing an Interface
- Demonstration: Creating an Interface

134

What Is an Interface?

An interface is:

- A contract that specifies what methods must be exposed by an implementing class
- Independent of the implementation



135

Creating and Implementing an Interface

Use the **interface** keyword

Do not specify access modifiers

```
interface ICalculator
{
    double Add();
    double Subtract();
    double Multiply();
    double Divide();
}
```

```
class Calculator : ICalculator
{
    public double Add(){}
    public double Subtract(){}
    public double Multiply(){}
    public double Divide(){}
}
```

Add a colon followed by the interfaces being implemented

Ensure that methods are publicly accessible

C# classes can implement multiple interfaces

136

Lesson 3: Defining Abstract Classes

- What Is an Abstract Class?
- What Is an Abstract Method?
- Demonstration: Creating an Abstract Class

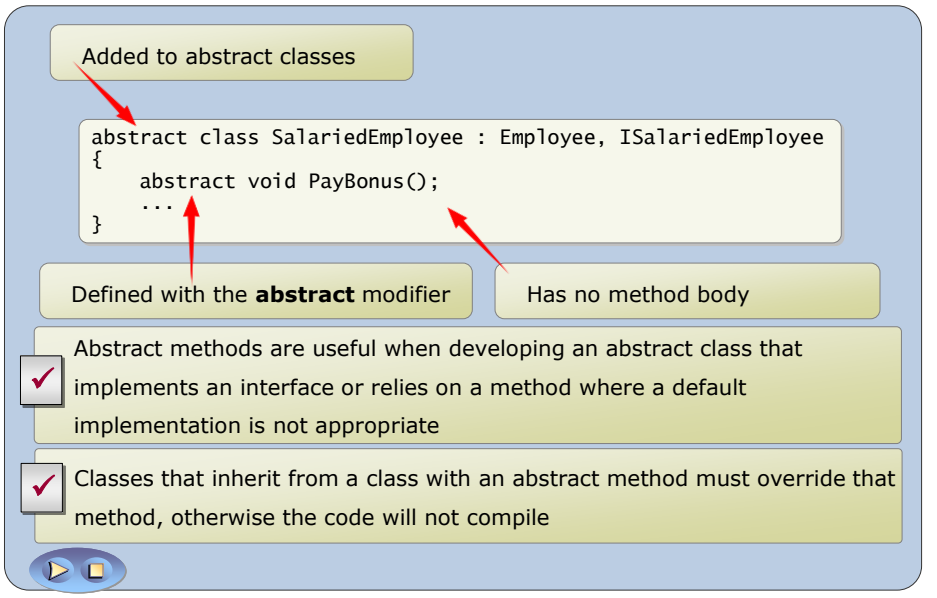
What Is an Abstract Class?

Abstract classes:

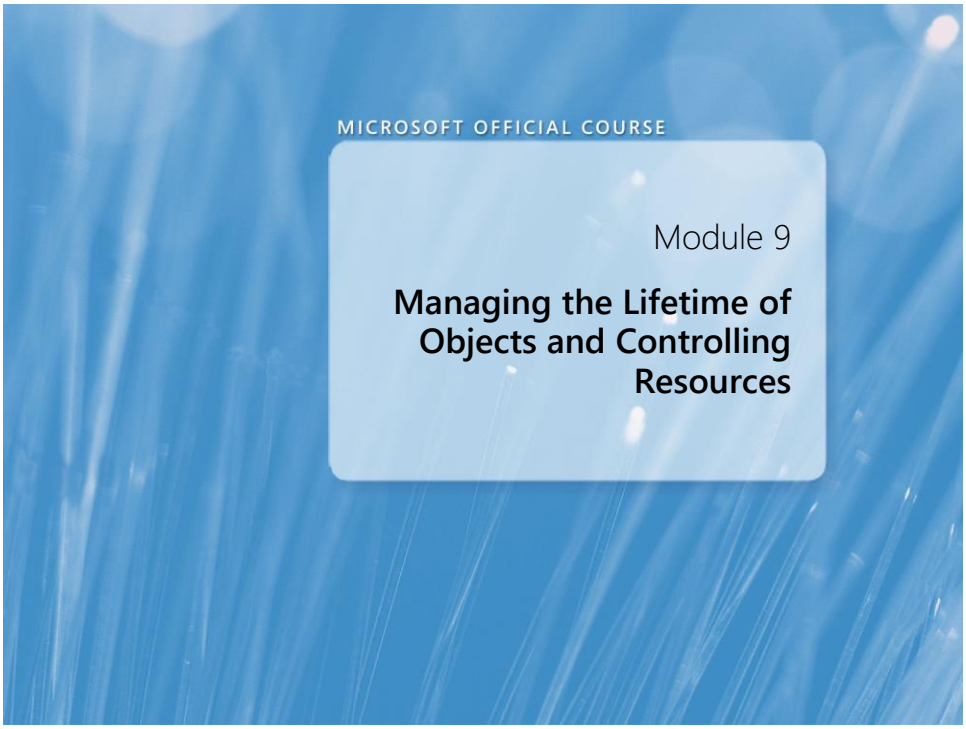
- Contain common code, thereby reducing code duplication
- Must be inherited
- Cannot be instantiated
- Can contain methods, fields, properties, and other members
- Use the **abstract** keyword

```
abstract class SalariedEmployee : Employee, ISalaried
{
    void ISalaried.PaySalary()
    {
        Console.WriteLine("Pay salary: {0}", currentSalary);
        // Common code for paying salary.
    }
    int currentSalary;
}

class ManualWorker : SalariedEmployee, ISalaried
{    ...    }

class Manager : SalariedEmployee, ISalaried
{    ...    }
```

What Is an Abstract Method?



139



140

Module Overview

- Introduction to Garbage Collection
- Managing Resources

141

Lesson 1: Introduction to Garbage Collection

- The Object Life Cycle
- Managed Resources in the .NET Framework
- How Does the Garbage Collector Work?
- Defining a Destructor
- The GC Class
- Demonstration: Implementing a Destructor

142

The Object Life Cycle

1. Block of memory allocated
2. Memory converted into an object (constructor)
3. Object used in application
4. Object tidied up (destructor)
5. Memory reclaimed

143

Managed Resources in the .NET Framework

Value Types

- Usually created on the stack
- Destroyed automatically when they go out of scope
- Never fragmented
- Fast to allocate and deallocate

```
private void MyMethod(…)
{
    MyStruct a = …;
    DateTime z = …;
    double y = …;
    int x = …;
    …

    // x, y, z, and a disappear here
}
```

Reference Types

- Created on the heap
- Support multiple references to the same object
- Can only be destroyed when the last reference disappears
- Can cause memory fragmentation
- Managed by the garbage collector
- More expensive to manage

```
string s = …;
MyClass c = …;

private void MyMethod2(…)
{
    string t = s;
    MyClass d = c;
    …
    // References t and d disappear here
    // Objects s and c remain on the heap
}
```

144

Defining a Destructor

```
class Employee
{
    ...
    ~Employee
    {
        // Destructor logic.
    }
}
```

Use a tilde (~) followed by the class name to define a destructor

Value types cannot define a destructor

```
class Employee
{
    ...
    protected override void Finalize()
    {
        try
        {
            // Destructor logic.
        }
        finally
        {
            base.Finalize();
        }
    }
}
```

Destructors never take parameters

The compiler converts the destructor to an override of the **Finalize** method

145

The GC Class

Name	Description
Collect	Forces garbage collection
WaitForPendingFinalizers	Suspends the current thread until all of the objects in the **freachable** queue have been finalized
SuppressFinalize	Prevents finalization of the object passed as the parameter
ReRegisterForFinalize	Requests finalization for an object that has either already been finalized or had finalization suppressed
AddMemoryPressure	Informs the runtime that you must allocate a large block of unmanaged memory
RemoveMemoryPressure	Informs the runtime that you have released a large block of unmanaged memory

146

Lesson 2: Managing Resources

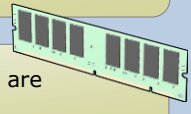
- Why Manage Resources in a Managed Environment?
- What Is the Dispose Pattern?
- Managing Resources in Your Applications
- Demonstration: Using the Dispose Pattern

147

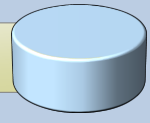
Why Manage Resources in a Managed Environment?

- Unmanaged resources are not subject to garbage collection:
  - You must take responsibility for releasing them when an object is finalized
  - Failing to dispose of unmanaged objects properly can result in data loss

Example: Files may be locked, preventing other applications from using them



Example: Data cached in memory may be lost if resources are not managed properly



Example: Connections to databases are often limited

148

What Is the Dispose Pattern?

```
class LogFileWriter : …, IDisposable
{
  private bool isDisposed = false;

  ~LogFileWriter()
  {
    Dispose(false);
  }

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

  protected virtual void
        Dispose(bool isDisposing)
  {
    if (!isDisposed)
    {
      if (isDisposing)
      {
        // clean up managed resources;
      }
        // clean up unmanaged resources
        isDisposed = false;
        base.Dispose(isDisposing)
    }
  }
}
```

- Implement the **IDisposable** interface
- Track the object status
- Add a destructor if you must ensure that the object is disposed of
- Add a public **Dispose** method that disposes of resources and suppresses subsequent finalization
- Add a virtual **Dispose** method that takes a **boolean** value indicating whether the object is currently disposing (or if false, finalizing)

149

MICROSOFT OFFICIAL COURSE

Module 10

Encapsulating Data and Defining Overloaded Operators

150

Module Overview

- Creating and Using Properties
- Creating and Using Indexers
- Overloading Operators

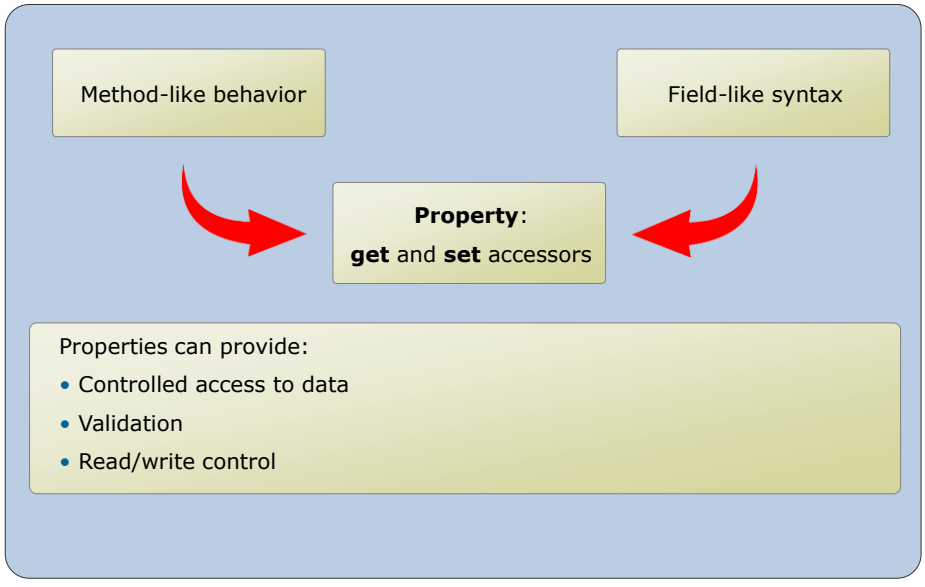
151

Lesson 1: Creating and Using Properties

- What Is a Property?
- Defining a Property
- Automatic Properties
- Instantiating an Object by Using Properties
- Defining Properties in an Interface
- Best Practices When Defining and Using Properties
- Demonstration: Using Properties

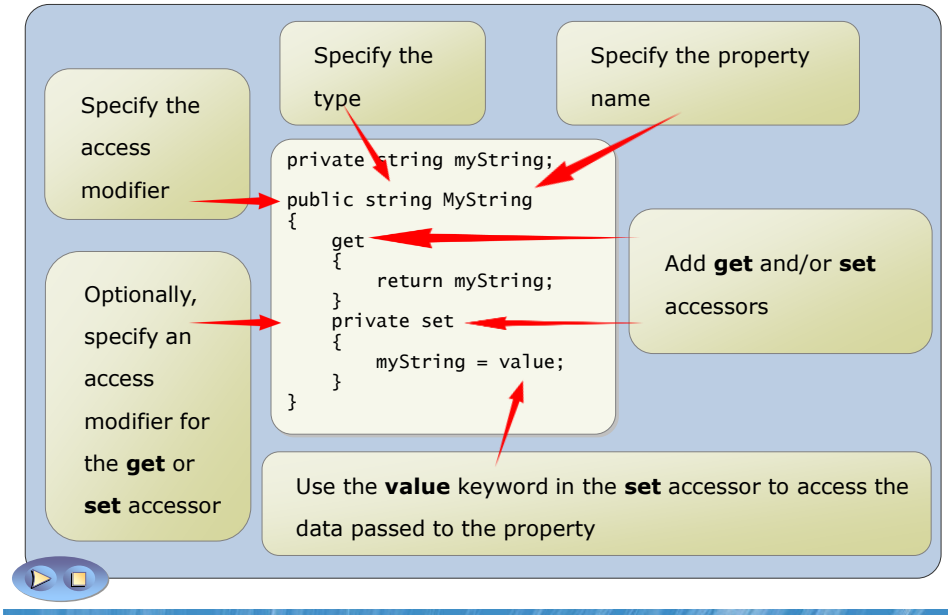
152

What Is a Property?



153

Defining a Property



154

Automatic Properties

```
public string Name { get; set; }
```

The automatic property shown above is converted by the compiler to code similar to:

```
private string _name;

public string Name
{
    get
    {
        return _name;
    }
    set
    {
        this._name = value;
    }
}
```

- Useful when you do not need to add custom logic to the property accessors
- Must specify both **get** and **set** accessors
- Important for forward compatibility
- No difference between automatic properties and normal properties to consuming applications

155

Instantiating an Object by Using Properties

```
Employee john = new Employee { Name = "John" };

Employee louisa = new Employee() { Department = "Technical" };

Employee mike = new Employee
                {
                    Name = "Mike",
                    Department = "Technical"
                };
```

- An object initializer avoids problems with defining several constructors
- A default constructor should instantiate properties to default values
- A constructor is called the object initializer
- A constructor is always run first, and properties are set after, so properties take precedence
- An object initializer can use no brackets to call the default constructor, brackets with no parameters to explicitly call the default constructor, or brackets with parameters to call a nondefault constructor

156

Defining Properties in an Interface

An interface is a contract between a type and a consuming application. It does not contain implementation details

Fields are considered implementation details; they cannot be defined in an interface

Properties are not implementation details; they are data exposed for consumption, so they can be defined in an interface

Do not specify an access modifier

```
interface IPerson
{
    string Name { get; set; }
    int Age { get; }
    DateTime DateOfBirth { set; }
}
```

Syntax is the same as for an automatic property except that you do not have to specify both accessors

157

Best Practices When Defining and Using Properties

Only expose properties where they are appropriate

BankAccount

- ✗ double Balance (get, ~~set~~)
- ✓ WithdrawMoney(double Amount)
- ✓ DepositMoney (double Amount)

Would you let an application directly set a bank balance?

Do not add code to a **get** accessor that has any side effects on data

Be careful to avoid accidentally recursive properties

158

Lesson 2: Creating and Using Indexers

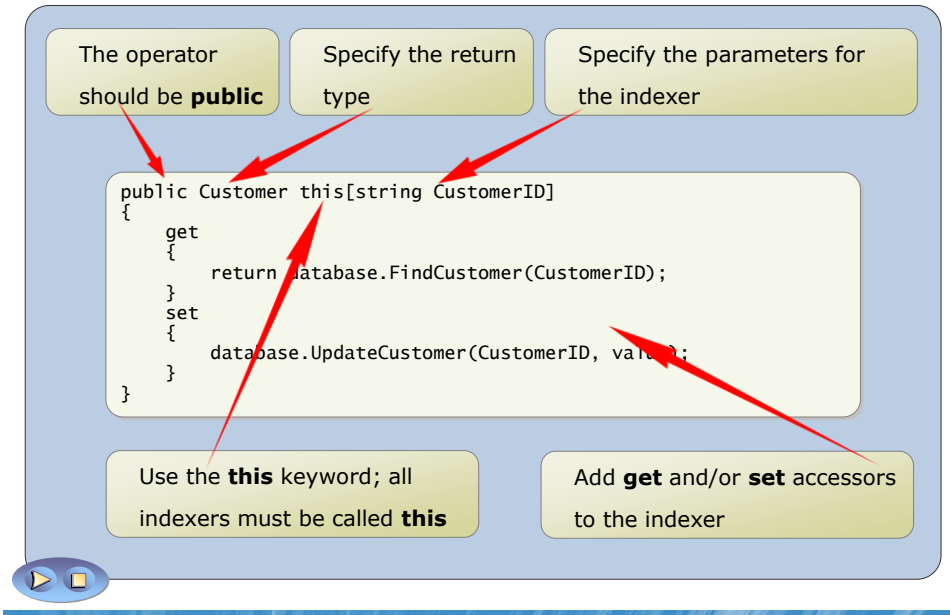
- What Is an Indexer?
- Creating an Indexer
- Comparing Indexers and Arrays
- Defining an Indexer in an Interface
- Demonstration: Creating and Using an Indexer

What Is an Indexer?

- Provides array-like syntax for accessing members in a set
- Can use different subscripts from an array, for example, a **string** instead of an **int**
- Can be overloaded

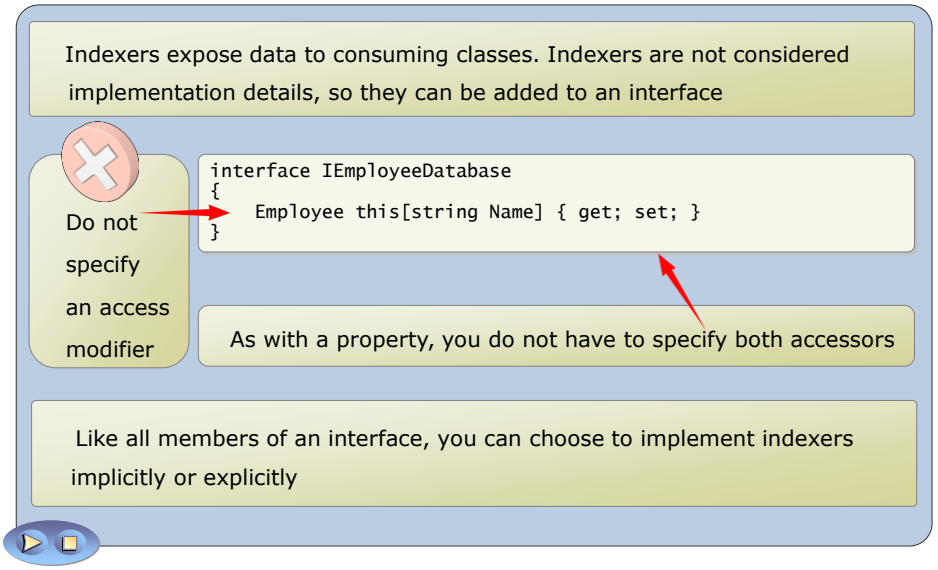
```
CustomerAddressBook addressBook = ...;

Address customerAddress = addressBook["a2332"];
...
customerAddress = addressBook[99];
```

Creating an Indexer



161

Defining an Indexer in an Interface

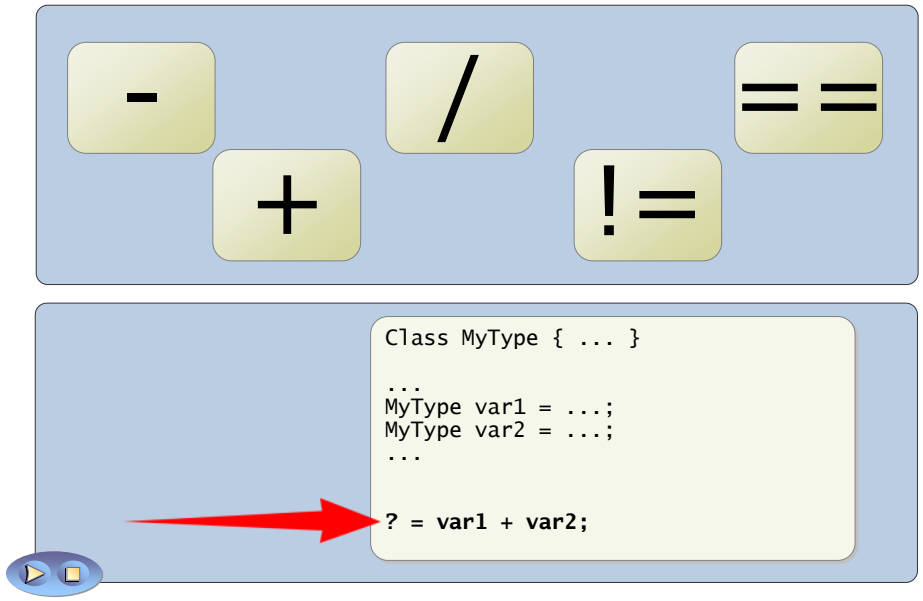


162

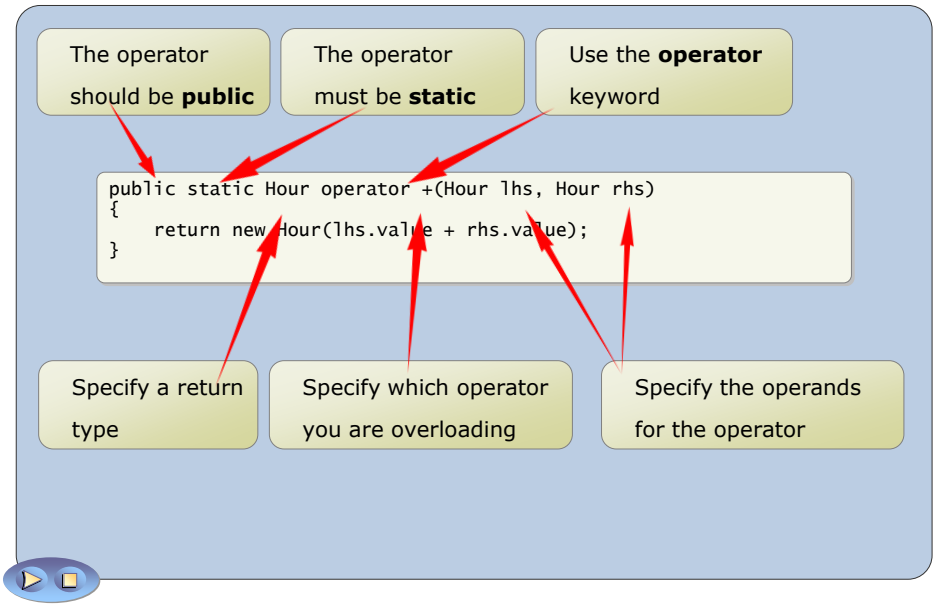
Lesson 3: Overloading Operators

- What Is Operator Overloading?
- Overloading an Operator
- Restrictions When Overloading Operators
- Best Practices When Overloading Operators
- Implementing and Using Conversion Operators
- Demonstration: Overloading an Operator

What Is Operator Overloading?



Overloading an Operator



165

MICROSOFT OFFICIAL COURSE

Module 11

Decoupling Methods and Handling Events

166

Module Overview

- Declaring and Using Delegates
- Using Lambda Expressions
- Handling Events

167

Lesson 1: Declaring and Using Delegates

- Why Decouple an Operation from a Method?
- Defining a Delegate
- Invoking a Delegate
- Defining Anonymous Methods

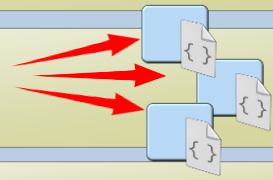
168

Why Decouple an Operation from a Method?

- Methods determined dynamically at run time
  - Method A or method B or method C

- Callback methods
  - Enable you to specify a method (or methods) to run when an asynchronous method call completes, particularly when you use third-party assemblies

- Multicast operations
  - Method A and method B and method C



Implemented by using delegates, which are basically references to methods

169

Defining a Delegate

Define a delegate by using the **delegate** keyword

```
public delegate bool isValidDelegate();
```

Create an instance of the delegate

```
public isValidDelegate isValid = null;
```

Add method references to the delegate by using the **+=** operator

```
isValid += CheckStateValid;
isValid += new isValidDelegate(CheckControl);
```

Remove method references from the delegate by using the **-=** operator

```
isValid -= CheckStateValid;
```

170

Lesson 2: Using Lambda Expressions

- What Is a Lambda Expression?
- Defining Lambda Expressions
- Variable Scope in Lambda Expressions

171

What Is a Lambda Expression?

A lambda expression is an expression that returns a method

Lambda expressions are defined by using the **=>** operator

Lambda expressions use a natural and concise syntax

```
x => x * x
```

This code example can be read as: Given x, calculate x * x
The parameter and return types are inferred

172

Defining Lambda Expressions

A simple expression where the type of the *x* parameter is inferred from the context

```
x => x * x
```

An expression that uses a Visual C# statement block instead of a simple expression

```
x => { return x * x ; }
```

An expression that calls a method and takes no parameters

```
() => myObject.MyMethod(0)
```

An expression where the parameter type is stated explicitly

```
(int x) => x / 2
```

An expression with multiple parameters and a parameter passed by reference

```
(ref int x,int y) => {x++; return x/y;}
```

Lesson 3: Handling Events

- What Is an Event?
- Defining an Event
- Using Events
- Best Practices for Using Events
- Using Events in Graphical Applications
- Demonstration: Using Events

What Is an Event?

An event provides a mechanism for informing consuming applications of a change of state or other occurrence in a type

Events are based on delegates

Unlike an instance of a delegate, an event can be raised (invoked) only by the containing class or a derivative of it

Consuming classes can subscribe to the event, adding references to methods that need to run when an event is raised

Events are used extensively in the .NET Framework; nearly all Windows Presentation Foundation (WPF) controls use them

175

Defining an Event

Define a delegate on which the event is based; the delegate should be at least as visible as the event

```
public delegate void MyEventDelegate(object sender, EventArgs e);

public event MyEventDelegate MyEvent = null;
```

Use the **event** keyword

Specify the delegate type

Specify a name for the event

You can define an event in an interface

176

Best Practices for Using Events

- ✓ Use the standard event signature
- ✓ Use a protected virtual method to raise an event
- ✓ Do not pass null to an event

177

MICROSOFT OFFICIAL COURSE

Module 12

Using Collections and Building Generic Types

178

Module Overview

- Using Collections
- Creating and Using Generic Types
- Defining Generic Interfaces and Understanding Variance
- Using Generic Methods and Delegates

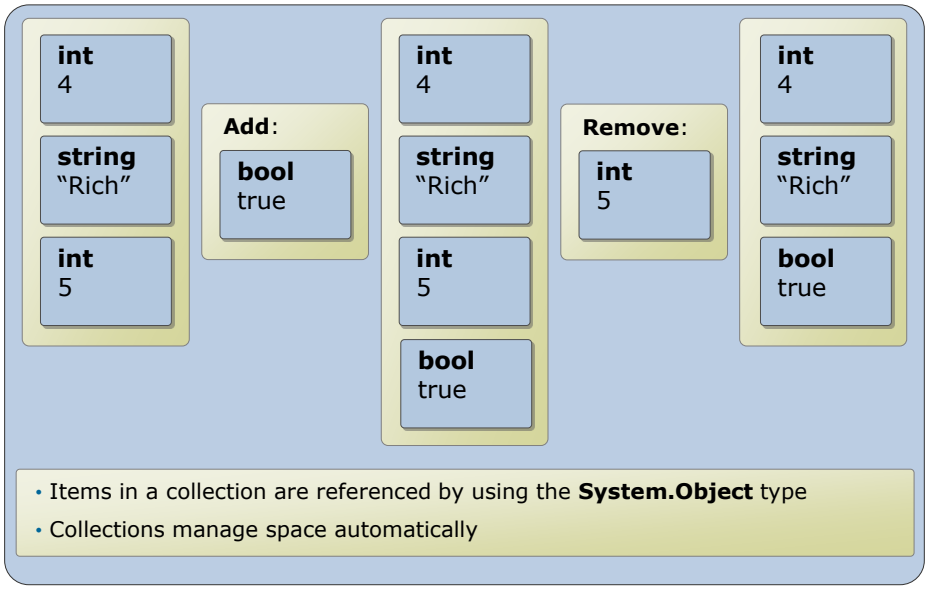
179

Lesson 1: Using Collections

- What Is a Collection?
- Using Collection Classes
- Iterating Through a Collection
- Common Collection Classes
- Using Collection Initializers
- Demonstration: Using Collections

180

What Is a Collection?



- Items in a collection are referenced by using the **System.Object** type
- Collections manage space automatically

181

Using Collection Classes

ICollection interface:
CopyTo
GetEnumerator
Count
Implemented by all collection classes

IList interface:
Add
Remove

Implemented by some collection classes

Items are stored in collections as objects. You must cast items retrieved from the collection

Some collection classes use alternatives such as **Push** and **Pop** instead of **Add** and **Remove**

```
ArrayList list = new ArrayList();
list.Add(6);
list.Remove(6);
list.RemoveAt(1);
int temp = (int)list[0]);
```

182

Iterating Through a Collection

A **foreach** loop displays every item in a collection in turn

```
foreach(<type> <control_variable> in <collection>)
{
   <foreach_statement_body>
}
```

To use a **foreach** loop, the collection must expose an enumerator. The **ICollection** interface defines a **GetEnumerator** method

```
ArrayList list = new ArrayList();
list.Add(99);
list.Add(10001);
list.Add(25);
...
foreach (int i in list)
{
   Console.WriteLine(i);
}
// Output: 99
//         10001
//         25
```

183

Common Collection Classes

ArrayList

An unordered collection, similar to an array. Items are accessed by index

Queue

A first-in, first-out collection. Use the **Enqueue** method instead of **Add**

Stack

A first-in, last-out collection. Use the **Push** method instead of **Add**

Hashtable

A collection of key and value pairs. Suitable for large collections

SortedList

A collection of key and value pairs. Items are ordered based on the key

184

Using Collection Initializers

You can use the **Add** method to add items to a collection

```
ArrayList al = new ArrayList();
al.Add("Value");
al.Add("Another Value");
```

You can also use a collection initializer to add items to a collection when you define the collection

```
// Assume person1 and person 2 are instantiated
// Person objects.
ArrayList al2 = new ArrayList()
{
    person1,
    person2
};
```

You can combine collection initializers with object initializers

```
ArrayList al3 = new ArrayList()
{
    new Person() {Name="James", Age =45},
    new Person() {Name="Tom", Age =31}
};
```

185

Lesson 2: Creating and Using Generic Types

- What Are Generic Types?
- Compiling Generic Types and Type Safety
- Defining a Custom Generic Type
- Adding Constraints to Generic Types

186

What Are Generic Types?

A generic type is a type that specifies one or more type parameters

Type parameters are like other parameters except that they represent a type, not an instance of a type

Type parameters are defined by using angle brackets after the class name

```
public class List<T>
```

This code example shows the definition of a class named **List** that takes a single type parameter named **T**. You can use **T** like any other type in the class

187

Compiling Generic Types and Type Safety

The **List<T>** class is used several times with different type parameters

```
List<string> names = new List<string>();
names.Add("John");
...
string name = names[0];

List<List<string>> listOfLists = new List<List<string>>();
listOfLists.Add(names);
...
List<string> data = listOfLists[0];
```

The compiler generates a strongly typed equivalent of the generic class, effectively generating the following methods

```
public void Add(string item)

...

public void Add(List<string> item)
```

The compiler-generated classes and methods are generated when your application is compiled. You cannot call the strongly typed version directly

188

Defining a Custom Generic Type

Define a class and specify a type parameter in angle brackets after the class name

```
class PrintableCollection<TItem>
{
    TItem [] data;
    int index;
    ...

    public void Insert(TItem item)
    {
        ...
        data[index] = item;
        ...
    }
}
```

Use the type parameter as an alias for the type in fields and properties

You can also use the type parameter in methods

189

Defining a Generic Method

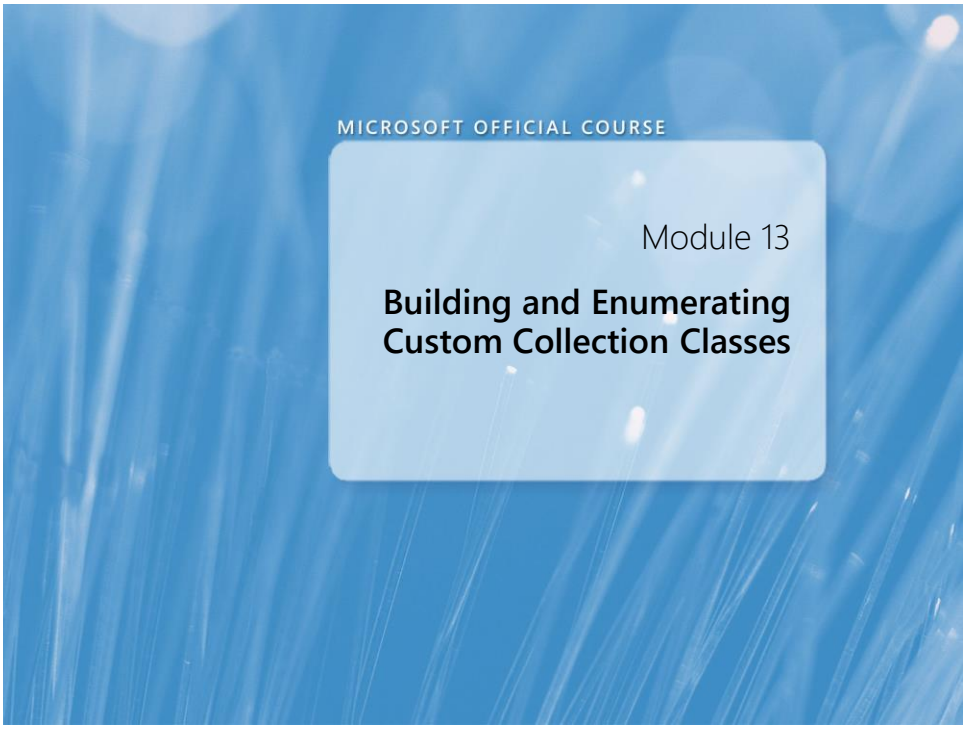
Define a method and specify a type parameter (or several) in angle brackets after the method name

```
ResultType MyMethod<Parameter1Type, ResultType>(Parameter1Type
param1)
    where ResultType : new()
{
    ResultType result = new ResultType();
    return result;
}
```

Specify any constraints on the type parameters

Use the type parameter(s) in the method parameters, return type, and method body

190



191

Module Overview

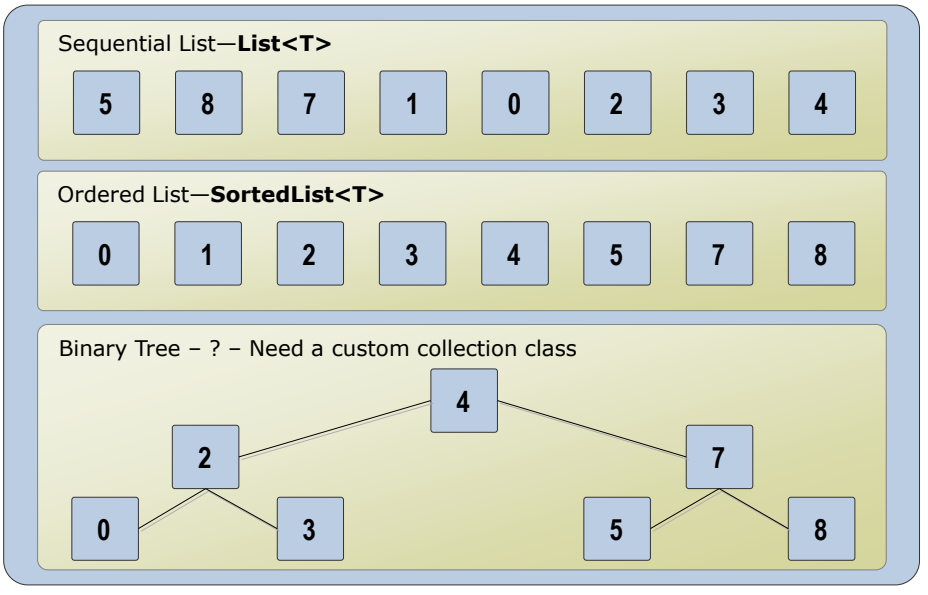
- Implementing a Custom Collection Class
- Adding an Enumerator to a Custom Collection Class

192

Lesson 1: Implementing a Custom Collection Class

- What Are Custom Collection Classes?
- Generic Collection Interfaces in the .NET Framework
- Implementing a Simple Custom Collection Class
- Implementing a Dictionary Collection Class

What Are Custom Collection Classes?



Generic Collection Interfaces in the .NET Framework

ICollection<T> : IEnumerable<T>

Methods:
- **Add**
- **Clear**
- **Contains**
- **CopyTo**
- **Remove**

Properties:
- **Count**
- **IsReadOnly**

IList<T> : ICollection<T>

Methods:
- **IndexOf**
- **Insert**
- **RemoveAt**

Properties:
- **Item**

IDictionary<TKey, TValue> : ICollection<KeyValuePair <TKey, TValue>>

Methods:
- **Add**
- **ContainsKey**
- **GetEnumerator**
- **Remove**
- **TryGetValue**

Properties:
- **Item**
- **Keys**
- **Values**

195

Implementing a Simple Custom Collection Class

```
class DoubleEndedQueue<T> :
ICollection<T>, IList<T>
{
    private List<T> items;

    // Type specific methods.
    public DoubleEndedQueue() {...}

    public EnqueueItemAtStart() {...}

    public DeqeueueItemFromStart()
    {...}

    public EnqueueItemAtEnd() {...}

    public DequeueItemFromEnd() {...}

    // Interface methods.
    public void Add(T item) {...}

    public void Clear() {...}

    public bool Contains(T item) {...}

    public void CopyTo(T[] array,
        int arrayIndex) {...}

    public int Count
    { get {...} }

    public bool IsReadOnly
    { get {...} }

    public bool Remove(T item) {...}

    public IEnumerator<T>
           GetEnumerator() {...}

    IEnumerator
        IEnumerable.GetEnumerator() {...}

    public int IndexOf(T item) {...}

    public void Insert
        (int index, T item) {...}

    public void RemoveAt(int index)
    {...}

    public T this[int index]
    {
      get {...}
      set {...}
    }
}
```

196

Implementing a Dictionary Collection Class

```
class IntelligentDictionary<TKey, TValue>
: IDictionary<TKey, TValue>
{
    public IntelligentDictionary() {...}

    public Tkey AddItem
        (Tkey key, Tvalue value) {...}

    public void Add
        (TKey key, TValue value) {...}

    public bool ContainsKey(TKey key)
    {...}

    public ICollection<TKey> Keys
    {
        get {...}
    }

    public bool Remove(TKey key) {...}

    public bool TryGetValue
        (TKey key, out TValue value)
    {...}

    public ICollection<TValue> Values
    { get {...} }

    public TValue this[TKey key]
    { get {...}
      set {...} }

     public void Add(KeyValuePair
         <TKey, TValue> item) {...}

    public void Clear() {...}

    public bool Contains
        (KeyValuePair<TKey, TValue> item)
    {...}

    public void CopyTo(KeyValuePair
        <TKey, TValue>[] array,
        int arrayIndex)
    {...}

    public int Count { get {...} }

    public bool IsReadOnly { get {...} }

    public bool Remove(KeyValuePair
        <TKey, TValue> item)
    {...}

    public IEnumerator<KeyValuePair
        <TKey, TValue>>
        GetEnumerator()
    {...}

    IEnumerator
        IEnumerable.GetEnumerator() {...}
}
```

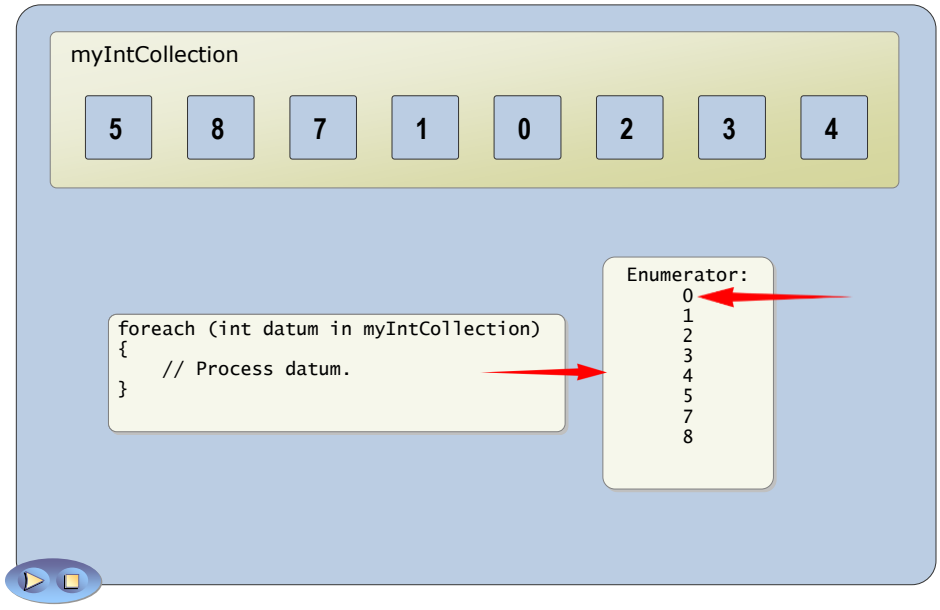
197

Lesson 2: Adding an Enumerator to a Custom Collection Class

- What Is an Enumerator?
- What Is the IEnumerable<T> Interface?
- What Is the IEnumerator<T> Interface?
- Implementing an Enumerator Manually
- Implementing an Enumerator by Using an Iterator

198

What Is an Enumerator?



199

What Is the IEnumerable<T> Interface?

The **IEnumerable<T>** interface is used with collection classes that support enumeration

The **IEnumerable<T>** interface defines the **GetEnumerator** method

The **IEnumerable<T>** interface inherits from the **IEnumerable** interface that also defines a **GetEnumerator** method

```
class CustomCollectionClass<T>
    : IEnumerable<T>
{
    public IEnumerator<T>
        GetEnumerator()
    {
        // Implementation not shown.
        ...
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException();
    }

    // Additional enumerators return
    // instances of the IEnumerable<T>
    // interface.
    public IEnumerable<T>  Backwards()
    {
        // Implementation not shown.
        ...
    }

    ...
}
```

200

What Is the IEnumerator<T> Interface?

IEnumerator<T>
: IEnumerator,
IDisposable

Properties:

- **T Current**
  Returns the current item from the collection
  The property is type-safe

IEnumerator

Methods:

- **void Reset**
  Resets the enumerator to the start of the collection
- **bool MoveNext**
  Moves the enumerator to the next point in the collection

Properties:

- **object Current**
  Returns the current item from the collection
  The property is not type-safe

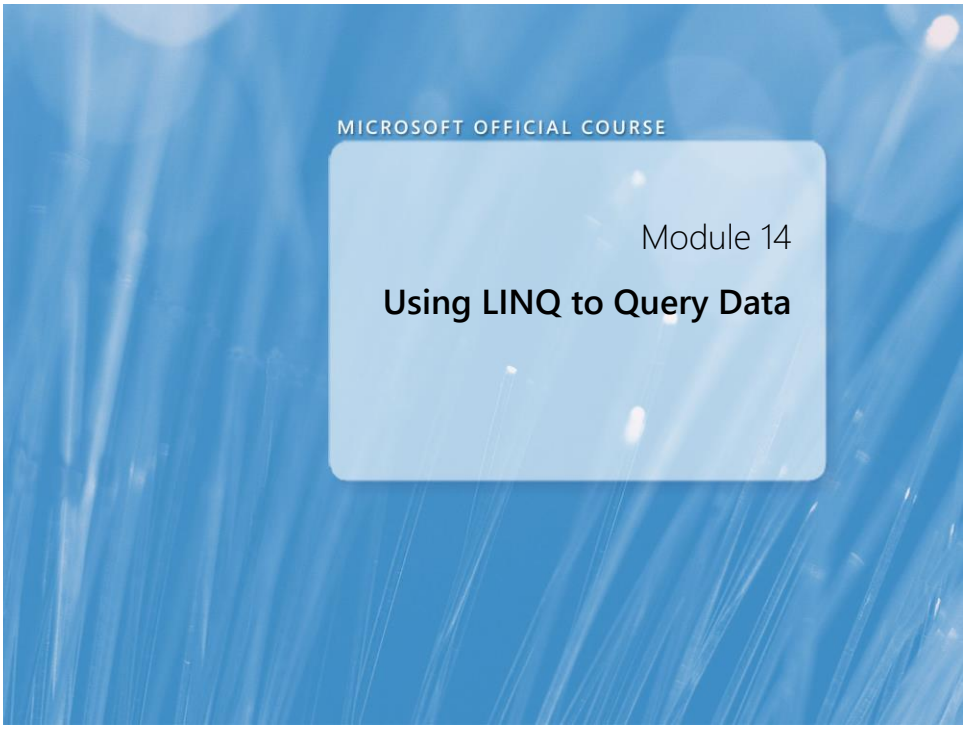
201

Implementing an Enumerator Manually

1. Implement the **IEnumerator<T>** interface
2. Define the **Current** property of type **T**
3. Define the **MoveNext** method that advances to the next item in the collection
4. Define the **Reset** method to move back to before the first item

```
partial class CustomCollectionClass<T> : IEnumerator<T>
{
   int pointer= -1;

   public T Current
   {
      get
      {
         if (pointer!= -1) {return pointer];}
         else {throw new InvalidOperationException();}
      }
   }

   public bool MoveNext()
   {
      if (pointer < (vals.Length - 1))
      {
         pointer ++;
         return true;
      }
      else { return false; }
   }

   public void Reset() { pointer = -1; }
}
```

202



203

Module Overview

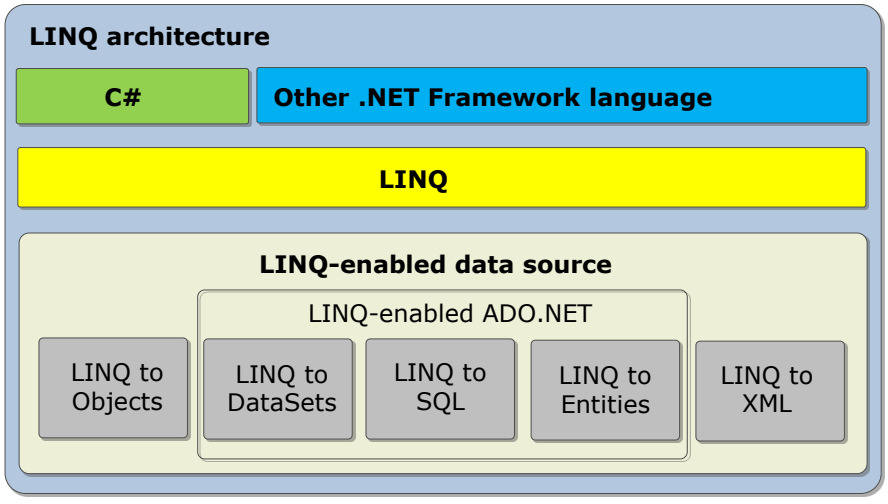
- Using the LINQ Extension Methods and Query Operators
- Building Dynamic LINQ Queries and Expressions

204

Lesson 1: Using the LINQ Extension Methods and Query Operators

- What Is the Purpose of LINQ?
- Querying Data and Building a Result Set
- Filtering Data
- Ordering Data
- Grouping Data and Performing Aggregate Calculations
- Joining Data from Different Data Sets

What Is the Purpose of LINQ?



Querying Data and Building a Result Set

Ability to select rows of data is a fundamental requirement

```
IEnumerable<Customer> customers = new[]
{
    new Customer{ FirstName = "…", LastName="…", Age = 41},
    ...
};

List<string> customerLastNames = new List<string>();
foreach (Customer customer in customers)
{
    customerLastNames.Add(customer.LastName);
}

...

IEnumerable<string> customerLastNames =
    customers.Select(cust => cust.LastName);
```

Traditional approach that uses the **foreach** statement

LINQ approach that uses the **Select** extension method

The **Select** extension method is available on any class that implements the generic **IQueryable<T>** or **IEnumerable<T>** interfaces

207

Filtering Data

Use the **Where** extension method to restrict the items returned

```
var customerLastNames =
    customers.Where(cust => cust.Age > 25).
    Select(cust => cust.LastName);
```

Get all customer last names for customers over 25

Use the **Where** extension method to filter data, and use the **Select** extension method to project data

Important: Use the methods in the correct order, otherwise you may get unexpected query results or compilation errors

208

Ordering Data

Order data with the **OrderBy**, **OrderByDescending**, **ThenBy**, and **ThenByDescending** extension methods

Use the **OrderBy** and **OrderByDescending** extension methods to apply a simple sort

```
var sortedCustomers =
    customers.OrderBy(cust => cust.FirstName);
```

Order customers by their first name

Use the **ThenBy** and **ThenByDescending** extension methods to sort by multiple keys

```
var sortedCustomers =
    customers.OrderBy(cust => cust.FirstName).
    ThenBy(cust => cust.Age);
```

Order customers by their first name and their age

209

Grouping Data and Performing Aggregate Calculations

Calculate an aggregated result across an enumerable collection

```
Console.WriteLine(
    "Count:{0}\t\tAverage age:{1}\t\tLowest:{2}\t\tHighest:{3}",
    customers.Count(),
    customers.Average(cust => cust.Age),
    customers.Min(cust => cust.Age),
    customers.Max(cust => cust.Age));
```

Group data by using the **GroupBy** extension method

```
var customersGroupedByAgeRange = customers.GroupBy( ... );
```

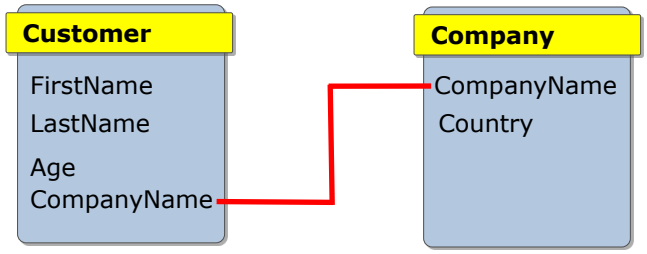
Eliminate duplicate values

```
Console.WriteLine("{0}",
    customers.Select(cust => cust.Age).Distinct().Count());
```

210

Joining Data from Different Data Sets

LINQ provides the **Join** extension method to join the data held in different sources together to perform composite queries




```
var customersAndCompanies = customers.Join(
    companies,
    custs => custs.CompanyName,
    comps => comps.CompanyName,
    (custs, comps) =>
        new { custs.FirstName, custs.LastName, comps.Country});
```

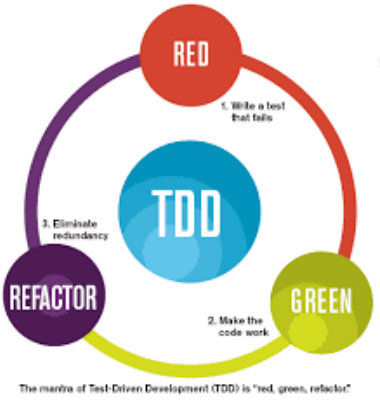
211

MICROSOFT OFFICIAL COURSE

Module 15

C# Test-Driven Development Concept

212



213

Demonstration Unit Test

214

… The End …

215