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.
}
```

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.

Text Continuation

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)

Text Continuation

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.

Exercise 1

- Task 1: *Copy the code for editing a student into the studentsList_MouseDoubleClick*
- event handler
- Run SetupSchoolDB.cmd.
- Start Visual Studio and from the path folder, open the School.sln solution.
- In the code for the MainWindow.xaml.cs window, in the studentsList_KeyDown event, locate the code for editing student details which is in the case Key.Enter block.
- Copy the code in this block to the clipboard and then paste it into the StudentsList_MouseDoubleClick method.

Exercise 1

- Task 2: *Run the application and verify that the user can now double-click a student to edit their details*
- Build the solution and resolve any compilation errors.
- Change Kevin Liu’s last name to Cook by pressing Enter in the main application window.
- Verify that the updated data is copied back to the students list and that the Save Changes button is now enabled.
- Change George Li’s name to Darren Parker by double-clicking on his row in the main application window.
- Verify that the updated data is copied back to the student list.

Exercise 1

- Task 3: *Use the Analyze Solution for Code Clones wizard to detect the duplicated code*
- On the Analyze menu, click Analyze Solution for Code Clones.
- In the Code Clone Analysis Results window, expand Exact Match.
- Using the results of the analysis in the Code Clone Analysis Results window, refactor the duplicated code into a method called editStudent that takes a Student as a parameter.
- Call this method from the studentsList_MouseDoubleClick and studentsList_KeyDown methods

Exercise 1

- Task 4: *Refactor the logic that adds and deletes a student into the addNewStudent and deleteStudent methods*
- Refactor the code in the case Key.Insert code block in the studentsList_KeyDown method into a method called addNewStudent that takes no parameters.
- Call this method from the case Key.Insert code block in the studentsList_KeyDown method.
- Refactor the code in the case Key.Delete code block in the studentsList_KeyDown method into a method called removeStudent that takes a Student as a parameter.
- Call this method from the case Key.Delete code block in the studentsList_KeyDown method.
- ***Verify that students can still be added and removed from the application***.

Exercise 2

- Task 1: *Run the application and observe that student details that are not valid can be entered.*
- In the ok_Click method in StudentForm.xaml.cs code, add a statement to check if the First Name box is empty.
- If it is empty, display a message box with a caption of Error containing the text The student must have a first name, and then exit the method.
- In the ok_Click method in StudentForm.xaml.cs code, add a statement to check if the Last Name box is empty.
- If it is empty, display a message box with a caption of Error containing the text The student must have a last name, and then exit the click method.

Exercise 2

- Task 2: *Add code to validate the first name, last name and birthdate fields*
- In the ok_Click method in StudentForm.xaml.cs code, add a statement to check if the Date of Birth box is empty.
- If the entered date is invalid, display a message box with a caption of Error containing the text The date of birth must be a valid date, and then exit the method.
- In the ok_Click method in StudentForm.xaml.cs code, add a statement to calculate the student’s age in years, and check if the age is less than five years.
- If the age is less than five years, display a message box with a caption of Error containing the text The student must at least 5 years old, and then exit the method. Use the following formula to calculate the age in years.
- Age in years = age in days / 365.25
- ***Run the application and verify that student information is now validated correctly***

Exercise 3

- Task 1: Saving Changes to the Class List
- In the MainWindow.xaml.cs code bring the System.Data and System.Data.Objects namespaces into scope.
- Add code to perform the following tasks when a user clicks Save Changes:
  - Call the SaveChanges method of the schoolContext object.
  - Disable the Save Changes button.

Exercise 3

- Task 2 Add exception handling to the code to catch concurrency, update, and general exceptions:
  - Enclose the lines of code that call the SaveChanges method of the schoolContext object and disable the Save Changes button in a try block.
  - Below the try block, add a catch block to catch any OptimisticConcurrencyException exceptions that may occur.
  - In the catch block, add the following code:
    - this.schoolContext.Refresh(RefreshMode.StoreWins, schoolContext.Students);
    - this.schoolContext.SaveChanges();
  - Add another catch block to catch any UpdateException exceptions that may occur, storing the exception in a variable named uEx.
  - In the catch block, add the following code:
    - MessageBox.Show(uEx.InnerException.Message, "Error saving changes");
    - this.schoolContext.Refresh(RefreshMode.StoreWins, schoolContext.Students);

Exercise 3

- Task 2 Add exception handling to the code to catch concurrency, update, and general exceptions:
- Add another **catch** block to catch any other type of exception that might occur, storing the exception in a variable named **ex**.
- In the **catch** block, add the following code:
  - MessageBox.Show(ex.Message, "Error saving changes");
  - this.schoolContext.Refresh(RefreshMode.ClientWins, schoolContext.Students);
- *Run the application and verify that data changes are persisted to the database*

Text Continuation

Module Review and Takeaways

- Review Question(s)

Text Continuation