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?

- A piece of code with a name that can be executed
- Input data is specified by parameters
- A value can be returned

- The basic unit of encapsulation
- .NET Framework applications contain a **Main** entry point method

Creating Methods

- Methods comprise two elements:
  - Method specification (return type, name, parameters)
  - Method body

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

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

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

- Use the **ref** keyword to pass parameter references

Invoking Methods

To call a method specify:

- Method name
- Any arguments to satisfy parameters

```
var upTime = 2000;
var shutdownAutomatically = true;
StartService(upTime, shutdownAutomatically);

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

Debugging Methods

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

Demonstration: Creating, Invoking, and Debugging Methods

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

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

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

Creating Overloaded Methods

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

```
void StopService()
{
   ...
}

void StopService(string serviceName)
{
   ...
}

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

Creating Methods that Use Optional Parameters

- Define all mandatory parameters first

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

- Satisfy parameters in sequence

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

// OR

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

Calling a Method by Using Named Arguments

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

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

Creating Methods that Use Output Parameters

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

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

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

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

Lesson 3: Handling Exceptions

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

What Is an Exception?

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

Handling Exception by Using a Try/Catch Block

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

```
try
{
     // Code that could throw an exception
}
catch (FileNotFoundException ex)
{
    // Catch all FileNotFoundException exceptions.
}
catch (Exception ex)
{
    // Catch all other exceptions.
}
```

- Most specific first!

Using a Finally Block

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

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

Throwing Exceptions

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

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

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

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

Lesson 4: Monitoring Applications

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

Using Logging and Tracing

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

Using Application Profiling

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

Using Performance Counters

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

Demonstration: Extending the Class Enrollment Application Functionality Lab

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

Lab: Extending the Class Enrollment Application Functionality

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

Estimated Time: 55 minutes

Lab Scenario

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

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

Module Review and Takeaways

- Review Questions