Module Overview

Module Overview

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

Overview

Applications often consist of logical units of functionality that perform specific functions, such as providing access to data or triggering some logical processing. Visual C# is an object-orientated language and uses the concept of methods to encapsulate logical units of functionality. A method can be as simple or as complex as you like, and therefore it is important to consider what happens to the state of your application when an exception occurs in a method.

In this module, you will learn how to create and use methods and how to handle exceptions. You will also learn how to use logging and tracing to record the details of any exceptions that occur.

Objectives

After completing this module, you will be able to:

- Create and invoke methods.
- Create overloaded methods and use optional parameters.
- Handle exceptions.
- Monitor applications by using logging, tracing, and profiling.

Lesson 1: Creating and Invoking Methods

Lesson 1: Creating and Invoking Methods

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

Lesson Overview

Every application exists to execute some algorithm. Wikipedia describes an algorithm as *"an unambiguous specification of how to solve a class of problems"*. In simple terms, an algorithm is the description of every action necessary to perform some process. For example, the algorithm for clicking an icon could be described as:

1. Locate the icon.
2. Move the mouse cursor to that icon.
3. Double-click the left mouse button.

Notice that each of these actions can be split further. Every time small actions are combined into one cohesive action a new level of abstraction is added.

In object-oriented languages such as Visual C#, a method is a unit of code that performs a discrete piece of work. This allows you to create new levels of abstractions as you see fit and divide your solution into manageable logical components.

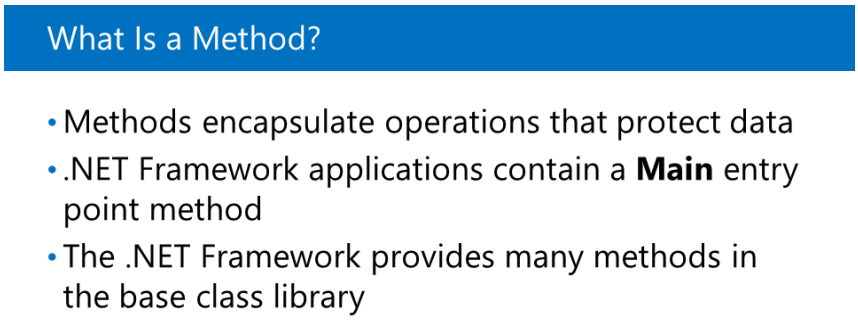
In this lesson, you will learn how to create and invoke methods.

OBJECTIVES

After completing this lesson, you will be able to:

- Describe the purpose of methods.
- Create methods.
- Invoke methods.
- Debug methods.

What Is a Method?



The ability to define and call methods is a fundamental component of object-oriented programming, because methods enable you to encapsulate operations that protect data that is stored inside a type.

Typically, any application that you develop by using the Microsoft .NET Framework and Visual C# will have many methods, each with a specific purpose. Some methods are fundamental to the operation of an application. For example, all Visual C# desktop applications must have a method called **Main** that defines the entry point for the application. When the user runs a Visual C# application, the common language runtime (CLR) executes the **Main** method for that application.

Methods can be designed for internal use by a type, and as such are hidden from other types. Public methods may be designed to enable other types to request that an object performs an action, and are exposed outside of the type. The .NET Framework itself is built from classes that expose methods that you can call from your applications to interact with the user and the computer.

Creating Methods

Creating Methods

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

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

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

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

A method comprises of two elements:

1. The method's specification.
2. The method's body.

The method specification defines the name of the method, the parameters that the method can take, the return type of the method, and the accessibility of the method. The combination of the name of the method and its parameter list are referred to as the method signature. The definition of the return value of a method is not regarded as part of the signature. Each method in a class must have a unique signature.

Naming Methods

A method name has the same syntactic restrictions as a variable name. A method must start with a letter or an underscore and can only contain letters, underscores, and numeric characters. Visual C# is case sensitive, so a class can contain two methods that have the same name and differ only in the casing of one or more letters—although this is not a good coding practice.

The following guidelines are recommended best practices when you choose the name of a method:

- Use verbs or verb phrases to name methods. This helps other developers to understand the structure of your code.
- Use UpperCamelCase.

Implementing a Method Body

The body of a method is a block of code that is implemented by using any of the available Visual C# programming constructs. The body is enclosed in braces.

You can define variables inside a method body, in which case they exist only while the method is running. When the method finishes, it is no longer in scope.

The following code example shows the body for the **StopService** method, which contains a variable named **isServiceRunning**. The **isServiceRunning** variable is only available inside the **StopService** code block. If you try to refer to the **isServiceRunning** variable outside the scope of the method, the compiler will raise a compile error with the message **The name 'isServiceRunning' does not exist in the current context.**

Variable Method Scope

```
void StopService()
{
var isServiceRunning = FourthCoffeeServices.Status;
...
}
```

Specifying Parameters

Parameters are local variables that are created when the method runs and are populated with values that are specified when the method is called. All methods must have a list of parameters. You specify the parameters in parentheses following the method name. Each parameter is separated by a comma. If a method takes no parameters, you specify an empty parameter list.

For each parameter, you specify the type and the name. By convention, parameters are named by using lowerCamelCase.

The following code example shows a method that accepts an **int** parameter and a **Boolean** parameter.

Passing Parameters to a Method

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

When defining the parameters that a method accepts, you can also prefix the parameter definition with the **ref** keyword. By using the **ref** keyword, you instruct the CLR to pass a reference to the parameter and not just the value of the parameter. You must initialize the **ref** parameter, and any changes to the parameter inside the method body will then be reflected in the underlying variable in the calling method.

The following code example shows how to define a parameter by using the **ref** keyword.

Defining a Parameter by Using the ref Keyword

```
void StopAllServices(ref int serviceCount)
{
serviceCount = FourthCoffeeServices.ActiveServiceCount;
}
```

Additional Reading: For more information about the ref keyword, see the ref (C# Reference) page at http://go.microsoft.com/fwlink/?LinkID=267782.

Specifying a Return Type

All methods must have a return type. A method that does not return a value has the **void** return type. You specify the return type before the method name when you define a method. When you declare a method that returns data, you must include a **return** statement in the method block.

The following code example shows how to return a **string** from a method.

Returning Data from a Method

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

The expression that the **return** statement specifies must have the same type as the method. When the return statement runs, this expression is evaluated and passed back to the statement that called the method. The method then finishes, so any other statements that occur after a **return** statement has been executed will not run.

However, each execution path in the method must call the **return** keyword eventually (or throw an exception). The compiler will produce an error if there's a possibility to run the method without reaching a **return** statement.

The following code example will generate a compiler error since it's possible to execute the method without returning a value.

This code will not compile

```
// Error CS0161 'GetServiceName()': not all code paths return a value
string GetServiceName(string language)
{
If(language == "en")
return "FourthCoffee.SalesService";
}
```

Invoking Methods

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.
}
```

You call a method to run the code in that method from part of your application. You do not need to understand how the code in a method works. You may not even have access to the code, if it is in a class in an assembly for which you do not have the source, such as the .NET Framework class library.

To call a method, you specify the method name and provide any arguments that correspond to the method parameters in brackets.

The following code example shows how to invoke the **StartService** method, passing **int** and **Boolean** variables to satisfy the parameter requirements of the method's signature.

Invoking a Method Passing Parameters

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

If the method returns a value, you specify how to handle this value, typically by assigning it to a variable of the same type, in your calling code.

The following code example shows how to capture the return value of the **GetServiceName** method in a variable named **serviceName.**

Capturing a Method Return Value

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

Additional Reading: For more information about methods, see the Methods (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkID=267774.

Debugging Methods

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

When you are debugging your application, you can step through code one statement at a time. This is an extremely useful feature because it enables you to test the logic that your application uses one step at a time.

Visual Studio provides a number of debugging tools that enable you to step through code in exactly the way you want to. For example, you can step through each line in each method that is executed, or you can ignore the statements inside a method that you know are working correctly. You can also step over code completely, preventing some statements from executing.

When debugging methods, you can use the following three debug features to control whether you step over, step into, or step out of a method:

- The Step Into feature executes the statement at the current execution position. If the statement is a method call, the current execution position will move to the code inside the method. After you have stepped into a method you can continue executing statements inside the method, one line at a time. You can also use the Step Into button to start an application in debug mode. If you do this, the application will enter break mode as soon as it starts.

- The Step Over feature executes the statement at the current execution position. However, this feature does not step into code inside a method. Instead, the code inside the method executes and the executing position moves to the statement after the method call. The exception to this is if the code for the method or property contains a breakpoint. If this is the case, execution will continue up to the breakpoint. Using Step Over when the application is closed will also start it in the debug mode and break on the first line.
- The Step Out feature enables you to execute the remaining code in a method. Execution will continue to the statement that called the method, and then pause at that point.

Additional Reading: For more information about stepping through code, see the Tutorial: Learn to debug using Visual Studio page at https://aka.ms/moc-20483c-m2-pg1.

Demonstration: Creating, Invoking, and Debugging Methods

Demonstration: Creating, Invoking, and Debugging Methods

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

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

Demonstration Steps

You will find the steps in the "Demonstration: Creating, Invoking, and Debugging Methods" section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD02_DEMO.md.

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

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

Lesson Overview

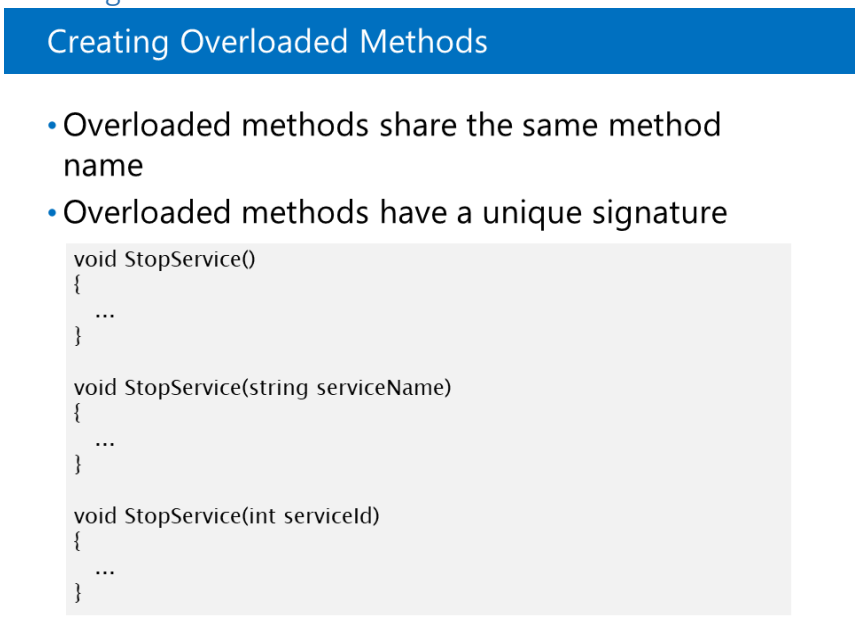
You have seen that you can define a method that accepts a fixed number of parameters. However, sometimes you might write one generic method that requires different sets of parameters depending on the context in which it is used. You can create overloaded methods with unique signatures to support this need. In other scenarios, you may want to define a method that has a fixed number of parameters, but enables an application to specify arguments for only the parameters that it needs. You can do this by defining a method that takes optional parameters and then using named arguments to satisfy the parameters by name.
In this lesson, you will learn how to create overloaded methods, define and use optional parameters, named arguments, and output parameters.

OBJECTIVES

After completing this lesson, you will be able to:

- Create an overloaded method.
- Use optional parameters.
- Use named arguments.
- Define output parameters.

Creating Overloaded Methods



When you define a method, you might realize that it requires different sets of information in different circumstances. You can define overloaded methods to create multiple methods with the same functionality that accept different parameters depending on the context in which they are called.
Overloaded methods have the same name as each other to emphasize their common intent. However, each overloaded method must have a unique signature, to differentiate it from the other overloaded versions of the method in the class.
The signature of a method includes its name and its parameter list. The return type is not part of the signature. Therefore, you cannot define overloaded methods that differ only in their return type.
The following code example shows three versions of the **StopService** method, all with a unique signature.

Overloaded Methods

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

When you invoke the **StopService** method, you have choice of which overloaded version you use. You simply provide the relevant arguments to satisfy a particular overload, and then the compiler works out which version to invoke based on the arguments that you passed.

Creating Methods that Use Optional Parameters

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);
```

By defining overloaded methods, you can implement different versions of a method that take different parameters. When you build an application that uses overloaded methods, the compiler determines which specific instance of each method it should use to satisfy each method call.

There are other languages and technologies that developers can use for building applications and components that do not follow these rules. A key feature of Visual C# is the ability to interoperate with applications and components that are written by using other technologies. One of the principal technologies that Windows uses is the Component Object Model (COM). COM does not support overloaded methods, but instead uses methods that can take optional parameters. To make it easier to incorporate COM libraries and components into a Visual C# solution, Visual C# also supports optional parameters.

Optional parameters are also useful in other situations. They provide a compact and simple solution when it is not possible to use overloading because the types of the parameters do not vary sufficiently to enable the compiler to distinguish between implementations.

The following code example shows how to define a method that accepts one mandatory parameter and two optional parameters.

Defining a Method with Optional Parameters

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

When defining a method that accepts optional parameters, you must specify all mandatory parameters before any optional parameters.

The following code example shows a method definition that uses optional parameters that throws a compile error.

Incorrect Optional Parameter Definition

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

You can call a method that takes optional parameters in the same way that you call any other method. You specify the method name and provide any necessary arguments. The difference with methods that take optional parameters is that you can omit the corresponding arguments, and the method will use the default value when the method runs.

The following code example shows how to invoke the **StopService** method, passing only an argument for the **forceStop** mandatory parameter.

Invoking a Method Specifying Only Mandatory Arguments.

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

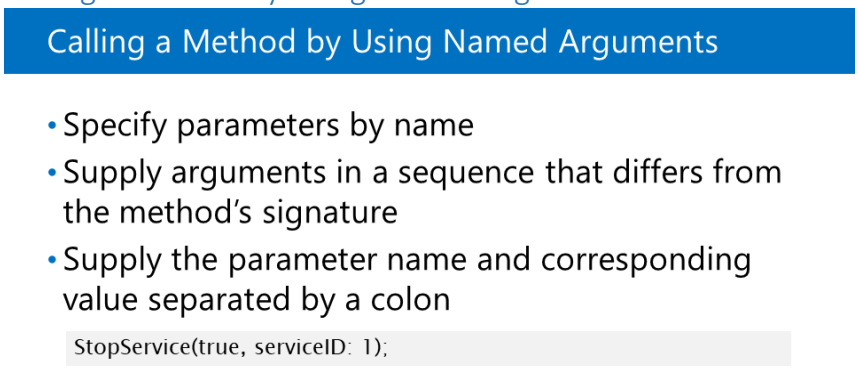
The following code example shows how to invoke the **StopService** method, passing an argument for the **forceStop** mandatory parameter, and an argument for the **serviceName** parameter.

Invoking a Method Specifying Mandatory and Optional Arguments

```
var forceStop = true;
var serviceName = "FourthCoffee.SalesService";
```

```
StopService(forceStop, serviceName);
```

Calling a Method by Using Named Arguments



Traditionally, when calling a method, the order and position of arguments in the method call corresponds to the order of parameters in the method signature. If the arguments are misaligned and the types mismatched, you receive a compile error.

In Visual C#, you can specify parameters by name, and therefore supply arguments in a sequence that differs from that defined in the method signature. To use named arguments, you supply the parameter name and corresponding value separated by a colon.

The following code example shows how to invoke the **StopService** method by using named arguments to pass the **serviceID** parameter.

Using Named Arguments

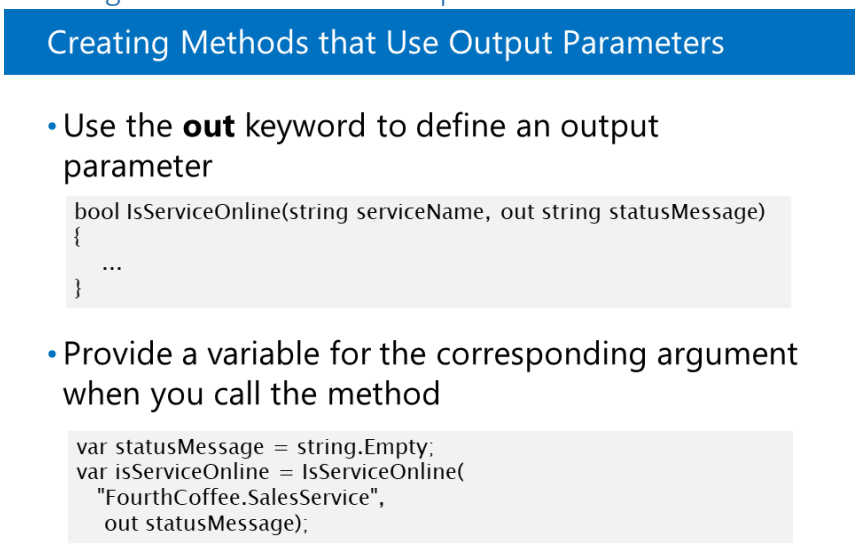
```
StopService(true, serviceID: 1);
```

When using named arguments in conjunction with optional parameters, you can easily omit parameters. Any optional parameters will receive their default value. However, if you omit any mandatory parameters, your code will not compile.

You can mix positional and named arguments. However, you must specify all positional arguments before any named arguments.

Additional Reading: For more information about using named arguments, see the Named and Optional Arguments (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkID=267784.

Creating Methods that Use Output Parameters



A method can pass a value back to the code that calls it by using a **return** statement. If you need to return more than a single value to the calling code, you can use output parameters to return additional data from the method. When you add an output parameter to a method, the method body is expected to assign a value to that parameter. When the method completes, the value of the output parameter is assigned to a variable that is specified as the corresponding argument in the method call.

To define an output parameter, you prefix the parameter in the method signature with the **out** keyword.

The following code example shows how to define a method that uses output parameters

Defining Output Parameters

```
bool IsServiceOnline(string serviceName, out string statusMessage)
{
var isOnline = FourthCoffeeServices.GetStatus(serviceName);
if (isOnline)
{
statusMessage = "Services is currently running.";
}
else
{
statusMessage = "Services is currently stopped.";
}
return isOnline;
}
```

A method can have as many output parameters as required. When you declare an output parameter, you must assign a value to the parameter before the method returns, otherwise the code will not compile.

To use an output parameter, you must provide a variable for the corresponding argument when you call the method, and prefix that argument with the **out** keyword. If you attempt to specify an argument that is not a variable or if you omit the **out** keyword, your code will not compile.

The following code example shows how to invoke a method that accepts an output parameter.

Invoking a Method that Accepts an Output Parameter

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

New versions of Visual C# let you define the variable in line with the **out** keyword. This allows the removal of the first line defining the empty variable. This also works with **ref** parameters.

The following code example shows how to invoke a method that accepts an output parameter, and define the variable in line

Invoking a Method that Accepts an Output Parameter and Define the Parameter in line

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

Additional Reading: For more information about output parameters, see the out parameter modifier (C# Reference) page at http://go.microsoft.com/fwlink/?LinkID=267785.

Lesson 3: Handling Exceptions

Lesson 3: Handling Exceptions

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

Lesson Overview

Exception handling is an important concept to ensure a good user experience and to limit data loss. Applications should be designed with exception handling in mind.

In this lesson, you will learn how to implement effective exception handling in your applications and how you can use exceptions in your methods to elegantly indicate an error condition to the code that calls your methods.

OBJECTIVES

After completing this lesson, you will be able to:

- Describe the purpose of an exception.
- Handle exceptions by using a try/catch block.
- Use a finally block to run code after an exception.
- Throw an exception.

What Is an Exception?

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**

Many things can go wrong as an application runs. Some errors may occur due to flaws in the application logic, but others may be due to conditions outside the control of your application. For example, your application cannot guarantee that a file exists on the file system or that a required database is online. When you design an application, you should consider how to ensure that your application can recover gracefully when such problems arise. It is common practice to simply check the return values from methods to ensure that they have executed correctly, however, this methodology is not always sufficient to handle all errors that may occur because:

- Not all methods return a value.
- You need to know why the method call has failed, not just that it has failed.
- Unexpected errors such as running out of memory cannot be handled in this way.

Traditionally, applications used the concept of a global error object. When a piece of code caused an error, it would set the data in this object to indicate the cause of the error and then return to the caller. It was the responsibility of the calling code to examine the error object and determine how to handle it. However, this approach is not robust, because it is too easy for a programmer to forget to handle errors appropriately.

How Exceptions Propagate

The .NET Framework uses exceptions to help overcome these issues. An exception is an indication of an error or exceptional condition. A method can throw an exception when it detects that something unexpected has happened, for example, the application tries to open a file that does not exist.

When a method throws an exception, the calling code must be prepared to detect and handle this exception. If the calling code does not detect the exception, the code is aborted and the exception is automatically propagated to the code that invoked the calling code. This process continues until a section of code takes responsibility for handling the exception. Execution continues in this section of code after the exception-handling logic has completed. If no code handles the exception, then the process will terminate and display a message to the user.

The Exception Type

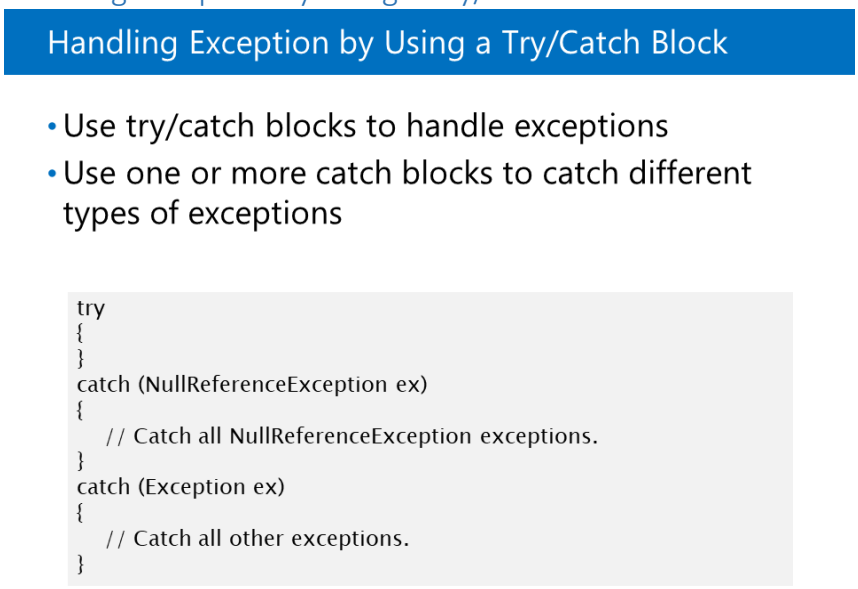
When an exception occurs, it is useful to include information about the original cause so that the method that handles the exception can take the appropriate corrective action. In the .NET Framework, exceptions are based on the **Exception** class, which contains information about the exception. When a method throws an exception, it creates an **Exception** object and can populate it with information about the cause of the error. The **Exception** object is then passed to the code that handles the exception.

The following table describes some of the exception classes provided by the .NET Framework.

Exception Class	Namespace	Description
Exception	**System**	Represents any exception that is raised during the execution of an application.
SystemException	**System**	Represents all exceptions raised by the CLR. The **SystemException** class is the base class for all the exception classes in the **System** namespace.
ApplicationException	**System**	Represents all non-fatal exceptions raised by applications and not the CLR.
NullReferenceException	**System**	Represents an exception that is caused when trying to use an object that is null.
FileNotFoundException	**System.IO**	Represents an exception caused when a file does not exist.
SerializationException	**System.Runtime.Serialization**	Represents an exception that occurs during the serialization or deserialization process.

Additional Reading: For more information about the Exception class, see the Exception Class page at https://aka.ms/moc-20483c-m2-pg2.

Handling Exception by Using a Try/Catch Block



The try/catch block is the key programming construct that enables you to implement Structured Exception Handling (SEH) in your applications. You wrap code that may fail and cause an exception in a try block, and add one or more catch blocks to handle any exceptions that may occur.
The following code example shows the syntax for defining a try/catch block.

Try/Catch Syntax

```
try
{
// Try block.
}
catch ([catch specification 1])
{
// Catch block 1.
}
catch ([catch specification n])
{
// Catch block n.
}
```

The statements that are enclosed in the braces in the try block can be any Visual C# statements, and can invoke methods in other objects. If any of these statements cause an exception to be thrown, execution passes to the appropriate catch block. The catch specification for each block determines which exceptions it will catch and the variable, if any, in which to store the exception. You can specify catch blocks for different types of exceptions. It is good practice to include a catch block for the general **Exception** type at the end of the catch blocks to catch all exceptions that have not been handled otherwise.
In the following code example, if the code in the try block causes a **NullReferenceException** exception, the code in the corresponding catch block runs. If any other type of exception occurs, the code in the catch block for the **Exception** type runs.

Handling NullReferenceException and Exception exceptions

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

When defining more than one catch block, you must ensure that you place them in the correct order. When an exception is thrown, the CLR attempts to match the exception against each catch block in turn. You must put more specific catch blocks before less specific catch blocks, otherwise your code will not compile.
Additional Reading: For more information about try/catch blocks, see the try-catch (C# Reference) page at https://aka.ms/moc-20483c-m2-pg3.

Using a Finally Block

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.
}
```

Some methods may contain critical code that must always be run, even if an unhandled exception occurs. For example, a method may need to ensure that it closes a file that it was writing to or releases some other resources before it terminates. A finally block enables you to handle this situation.
You specify a finally block after any catch handlers in a try/catch block. It specifies code that must be performed when the block finishes, irrespective of whether any exceptions, handled or unhandled, occur. If an exception is caught and handled, the exception handler in the catch block will run before the finally block.
You can also add a finally block to code that has no catch blocks. In this case, all exceptions are unhandled, but the finally block will always run.
The following code example shows how to implement a try/catch/finally block.

Try/Catch/Finally Blocks

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

Additional Reading: For more information about try/catch/finally blocks, see the try-catch-finally (C# Reference) page at https://aka.ms/moc-20483c-m2-pg4.

Throwing Exceptions

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;
}
```

You can create an instance of an exception class in your code and throw the exception to indicate that an exception has occurred. When you throw an exception, execution of the current block of code terminates and the CLR passes control to the first available exception handler that catches the exception.

To throw an exception, you use the **throw** keyword and specify the exception object to throw.

The following code example shows how to create an instance of the **NullReferenceException** class and then throw the **ex** object.

Creating and Throwing an Exception

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

A common strategy is for a method or block of code to catch any exceptions and attempt to handle them. If the catch block for an exception cannot resolve the error, it can rethrow the exception to propagate it to the caller.

The following code example shows how to rethrow an exception that has been caught in a catch block.

Rethrowing an Exception

```
try
{
}
catch (NullReferenceException ex)
{
// Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
// Attempt to handle the exception
...
// If this catch handler cannot resolve the exception,
// throw it to the calling code
throw;
}
```

Lesson 4: Monitoring Applications

Lesson 4: Monitoring Applications

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

Lesson Overview

When you develop real-world applications, writing code is just one part of the process. You are likely to spend a significant amount of time resolving bugs, troubleshooting problems, and optimizing the performance of your code. Visual Studio and the .NET Framework provide various tools that can help you to perform these tasks more effectively. In this lesson, you will learn how to use a range of tools and techniques to monitor and troubleshoot your applications.

OBJECTIVES

After completing this lesson, you will be able to:

- Use logging and tracing in your code.
- Use application profiling in Visual Studio.
- Use performance counters to monitor the performance of your application.

Using Logging and Tracing

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

Logging and tracing are similar, but distinct, concepts. When you implement logging in your application, you add code that writes information to a destination log, such as a text file or the Windows event log. Logging enables you to provide users and administrators with more information about what your code is doing. For example, if your application handles an exception, you might write the details to the Windows event log to enable the user or a system administrator to resolve any underlying problems.

By contrast, developers use tracing to monitor the execution of an application. When you implement tracing, you add code that writes messages to a trace listener, which in turn directs your output to a specified target. By default, your

trace messages are shown in the **Output** window in Visual Studio. You typically use tracing to provide information about variable values or condition results, to help you find out why your application behaves in a particular way. You can also use tracing techniques to interrupt the execution of an application in response to conditions that you define.

Writing to the Windows Event Log

Writing to the Windows event log is one of the more common logging requirements you might encounter. The **System.Diagnostics.EventLog** class provides various static methods that you can use to write to the Windows event log. In particular, the **EventLog.WriteEntry** method includes several overloads that you can use to log various combinations of information. To write to the Windows event log, you need to provide a minimum of three pieces of information:

- The *event log*. This is the name of the Windows event log you want to write to. In most cases you will write to the **Application** log.
- The *event source*. This identifies where the event came from, and is typically the name of your application. When you create an event source, you associate it with an event log.
- The *message*. This is the text that you want to add to the log.

You can also use the **WriteEntry** method to specify a category, an event ID, and an event severity if required.

Additional Reading: Writing to the Windows event log requires a high level of permissions. If your application does not run with sufficient permissions, it will throw a **SecurityException** when you attempt to create an event source or write to the event log.

The following example shows how to write a message to the event log:

Writing to the Windows Event Log

```
string eventLog = "Application";
string eventSource = "Logging Demo";
string eventMessage = "Hello from the Logging Demo application";
// Create the event source if it does not already exist.
If (!EventLog.SourceExists(eventSource))
EventLog.CreateEventSource(eventSource, eventLog);
// Log the message.
EventLog.WriteEntry(eventSource, eventMessage);
```

Additional Reading: For more information on writing to the Windows event log, see the How to write to an event log by using Visual C# page at https://aka.ms/moc-20483c-m2-pg6.

Debugging and Tracing

The **System.Diagnostics** namespace includes two classes, **Debug** and **Trace**, which you can use to monitor the execution of your application. These two classes work in a similar way and include many of the same methods. However, **Debug** statements are only active if you build your solution in **Debug** mode, whereas **Trace** statements are active in both **Debug** and **Release** mode builds.

The **Debug** and **Trace** classes include methods to write format strings to the **Output** window in Visual Studio, as well as to any other listeners that you configure. You can also write to the **Output** window only when certain conditions are met, and you can adjust the indentation of your trace messages. For example, if you are writing details of every object within an enumeration to the **Output** window, you might want to indent these details to distinguish them from other output.

The **Debug** and **Trace** classes also include a method named **Assert**. The **Assert** method enables you to specify a condition (an expression that must evaluate to **true** or **false**) together with a format string. If the condition evaluates to **false**, the **Assert** method interrupts the execution of the program and displays a dialog box with the message you specify. This method is useful if you need to identify the point in a long-running program at which an unexpected condition arises.

The following example shows how to use the **Debug** class to write messages to the **Output** window, and to interrupt execution if unexpected conditions arise.

Using the Debug Class

```
int number;
Console.WriteLine("Please type a number between 1 and 10, and then press Enter");
string userInput = Console.ReadLine();
Debug.Assert(int.TryParse(userInput, out number),
string.Format("Unable to parse {0} as integer", userInput);
Debug.WriteLine(The current value of userInput is: {0}", userInput);
Debug.WriteLine(The current value of number is: {0}", number);
Console.WriteLine("Press Enter to finish");
Console.ReadLine();
```

Additional Reading: For more information on tracing, see the How to trace and debug in Visual C# page at http://go.microsoft.com/fwlink/?LinkID=267790.

Using Application Profiling

Using Application Profiling

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

When you develop applications, making your code work without bugs is only part of the challenge. You also have to ensure that your code runs efficiently. You need to review how long your code takes to accomplish tasks and whether it uses excessive processor, memory, disk, or network resources.

Visual Studio includes a range of tools, collectively known as the Visual Studio Profiling Tools, that can help you to analyze the performance of your applications. At a high level, running a performance analysis in Visual Studio consists of three high-level steps:

1. Create and run a *performance session*. All performance analysis takes place within a performance session. You can create and run a performance session by launching the Performance Wizard from the **Analyze** menu in Visual Studio. When the performance session is running, you run your application as you usually would. While your application is running, you typically aim to use functionality that you suspect may be causing performance issues.
2. Analyze the *profiling report*. When you finish running your application, Visual Studio displays the profiling report. This includes a range of information that can provide insights into the performance of your application. For example, you can:
   - See which functions consume the most CPU time.
   - View a timeline that shows what your application was doing when.
   - View warnings and suggestions on how to improve your code.
3. Revise your code and repeat the analysis. When your analysis is complete, you should make changes to your code to fix any issues that you identified. You can then run a new performance session and generate a new profiling report. The Visual Studio Profiling Tools enable you to compare two reports to help you identify and quantify how the performance of your code has changed.

Performance sessions work by sampling. When you create a performance session, you can choose whether you want to sample CPU use, .NET memory allocation, concurrency information for multi-threaded applications, or whether you want to use instrumentation to collect detailed timing information about every function call. In most cases you will want to start by using CPU sampling, which is the default option. CPU sampling uses statistical polling to determine which functions are using the most CPU time. This provides an insight into the performance of your application, without consuming many resources and slowing down your application.

Additional Reading: For more information on application profiling, see the Analyzing Application Performance by Using Profiling Tools page at https://aka.ms/moc-20483c-m2-pg5.

Using Performance Counters

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)

Performance counters are system tools that collect information on how resources are used. Viewing performance counters can provide additional insights into what your application is doing, and can help you to troubleshoot performance problems. Performance counters fall into three main groups:

- *Counters that are provided by the operating system and the underlying hardware platform*. This group includes counters that you can use to measure processor use, physical memory use, disk use, and network use. The details of the counters available will vary according to the hardware that the computer contains.
- *Counters that are provided by the .NET Framework*. The .NET Framework includes counters that you can use to measure a wide range of application characteristics. For example, you can look at the number of exceptions thrown, view details of locks and thread use, and examine the behavior of the garbage collector.
- *Counters that you create yourself*. You can create your own performance counters to examine specific aspects of the behavior of your application. For example, you can create a performance counter to count the number of calls to a particular method or to count the number of times a specific exception is thrown.

Browsing and Using Performance Counters

Performance counters are organized into *categories*. This helps you to find the counters you want when you are capturing and reviewing performance data. For example, the **PhysicalDisk** category typically includes counters for the percentage of time spent reading and writing to disk, amounts of data read from and written to disk, and the queue lengths to read data from and write data to disk.

Note: You can browse the performance counters available on your computer from Visual Studio. In Server Explorer, expand **Servers**, expand the name of your computer, and then expand **Performance Counters**.

Typically, you capture and view data from performance counters in Performance Monitor (perfmon.exe). Performance Monitor is included in the Windows operating system and enables you to view or capture data from performance counters in real time. When you use Performance Monitor, you can browse performance counter categories and add multiple performance counters to a graphical display. You can also create *data collector sets* to capture data for reporting or analysis.

Creating Custom Performance Counters

You can use the **PerformanceCounter** and **PerformanceCounterCategory** classes to interact with performance counters in a variety of ways. For example, you can:

- Iterate over the performance counter categories available on a specified computer.
- Iterate over the performance counters within a specified category.
- Check whether specific performance counter categories or performance counters exist on the local computer.
- Create custom performance counter categories or performance counters.

You typically create custom performance counter categories and performance counters during an installation routine, rather than during the execution of your application. After a custom performance counter is created on a specific computer, it remains there. You do not need to recreate it every time you run your application. To create a custom performance counter, you must specify a base counter type by using the **PerformanceCounterType** enumeration.

The following example shows how to programmatically create a custom performance counter category. This example creates a new performance counter category named **FourthCoffeeOrders**. The category contains two performance

counters. The first performance counter tracks the total number of coffee orders placed, and the second tracks the number of orders placed per second.

Programmatically Creating Performance Counter Categories and Performance Counters

```
if (!PerformanceCounterCategory.Exists("FourthCoffeeOrders"))
{
CounterCreationDataCollection counters = new CounterCreationDataCollection();
CounterCreationData totalOrders = new CounterCreationData();
totalOrders.CounterName = "# Orders";
totalOrders.CounterHelp = "Total number of orders placed";
totalOrders.CounterType = PerformanceCounterType.NumberOfItems32;
counters.Add(totalOrders);
CounterCreationData ordersPerSecond = new CounterCreationData();
ordersPerSecond.CounterName = "# Orders/Sec";
ordersPerSecond.CounterHelp = "Number of orders placed per second";
ordersPerSecond.CounterType = PerformanceCounterType.RateOfCountsPerSecond32;
counters.Add(ordersPerSecond);
PerformanceCounterCategory.Create("FourthCoffeeOrders", "A custom category for demonstration",
PerformanceCounterCategoryType.SingleInstance, counters);
}
```

Note: You can also create performance counter categories and performance counters from Server Explorer in Visual Studio.

When you have created custom performance counters, your application must provide the performance counters with data. Performance counters provide various methods that enable you to update the counter value, such as the **Increment** and **Decrement** methods. How the counter processes the value will depend on the base type you selected when you created the counter.

The following example shows how to programmatically update custom performance counters.

Using Custom Performance Counters

```
// Get a reference to the custom performance counters.
PerformanceCounter counterOrders = new PerformanceCounter("FourthCoffeeOrders", "# Orders", false);
PerformanceCounter counterOrdersPerSec = new PerformanceCounter("FourthCoffeeOrders", "# Orders/Sec",
false);
// Update the performance counter values at appropriate points in your code.
public void OrderCoffee()
{
counterOrders.Increment();
counterOrdersPerSec.Increment();
// Coffee ordering logic goes here.
}
```

When you have created a custom performance counter category, you can browse to your category and select individual performance counters in Performance Monitor. When you run your application, you can then use Performance Monitor to view data from your custom performance counters in real time.

Demonstration: Extending the Class Enrollment Application Functionality Lab

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

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

Demonstration Steps

You will find the steps in the **Demonstration: Extending the Class Enrollment Application Functionality Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD02_DEMO.md.

Lab Scenario

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.

Lab: Extending the Class Enrollment Application Functionality

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

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.

Objectives

After completing this lab, you will be able to:

- Refactor code to facilitate reusability.
- Write Visual C# code that validates data entered by a user.
- Write Visual C# code that saves changes back to a database.

Lab Setup

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD02_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD02_LAK.md.

Module Review and Takeaways

Module Review and Takeaways

- Review Questions

In this module, you learned how to create and use methods, and how to handle exceptions. You also learned how to use logging and tracing to record the details of any exceptions that occur.

Review Questions

1. The return type of a method forms part of a methods signature.

( )True

( )False

2. When using output parameters in a method signature, which one of the following statements is true?

( )You cannot return data by using a return statement in a method that use output parameters.

( )You can only use the type object when defining an output parameter.

( )You must assign a value to an output parameter before the method returns.

( )You define an output parameter by using the output keyword.

3. A finally block enables you to run code in the event of an error occurring?

( )True

( )False

4. Trace statements are active in both **Debug** and **Release** mode builds.

( )True

( )False