Microsoft® Official Course

Module 2

Creating Methods, Handling Exceptions, and Monitoring Applications

Microsoft®

Module Overview

1. Creating and Invoking Methods
2. Creating Overloaded Methods and Using Optional and Output Parameters
3. Handling Exceptions
4. Monitoring Applications

Lesson 1:
Creating and Invoking Methods

What Is a Method?

- Methods **encapsulate** operations that protect data that is stored inside a type
  - "Don't repeat yourself" (DRY) – code reuse
  - Methods: "a collection of statements to perform a specific task"
  - Improves maintainability of programs
  - Simplifies the process of writing programs

- Method call: a statement that causes a function to execute
- Method definition: statements that make up a function

- In-class example: let's create a **rectangle** class that implements several methods (e.g. **ComputeArea** and **ComputePerimeter**).
  - public vs private
    - public methods are exposed outside of the type

What Is a Method?

- .NET Framework applications contain a **Main** entry point method
  - When the user runs a Visual C# application, the common language runtime (CLR) executes the **Main** method for that application.
- static methods are discussed in module 4 of this course
- 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.
  - Recommendations:
    - Use verbs or verb phrases to name methods. This helps other developers to understand the structure of your code.
    - Use Pascal case. Do not start public method names with an underscore or a lowercase letter
      - Example: StartService, WriteLine,...

Creating Methods

- Methods comprise two elements:
  - Method specification (return type, name, parameters)
  - Method body
    - The body is enclosed in braces
    - Variables declared inside a method body exist only while the method is running (…local vars).

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

- Use the **ref** keyword to pass parameter references
    - Example: swap method
    - (CLR will pass a reference to the parameter …) any changes to the parameter inside the method body will then be reflected in the underlying variable in the calling 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.

Invoking Methods

To call a method specify:

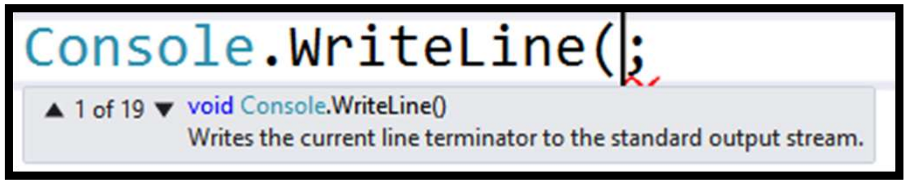
- Method name
- Any arguments to satisfy parameters

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

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

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

Creating Overloaded Methods

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



```
Console.WriteLine(;
```

▲ 1 of 19 ▼ void Console.WriteLine()
Writes the current line terminator to the standard output stream.

```
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);
```

- 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.

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 declare an output parameter, you must assign a value to the parameter before the method returns, otherwise the code will not compile.

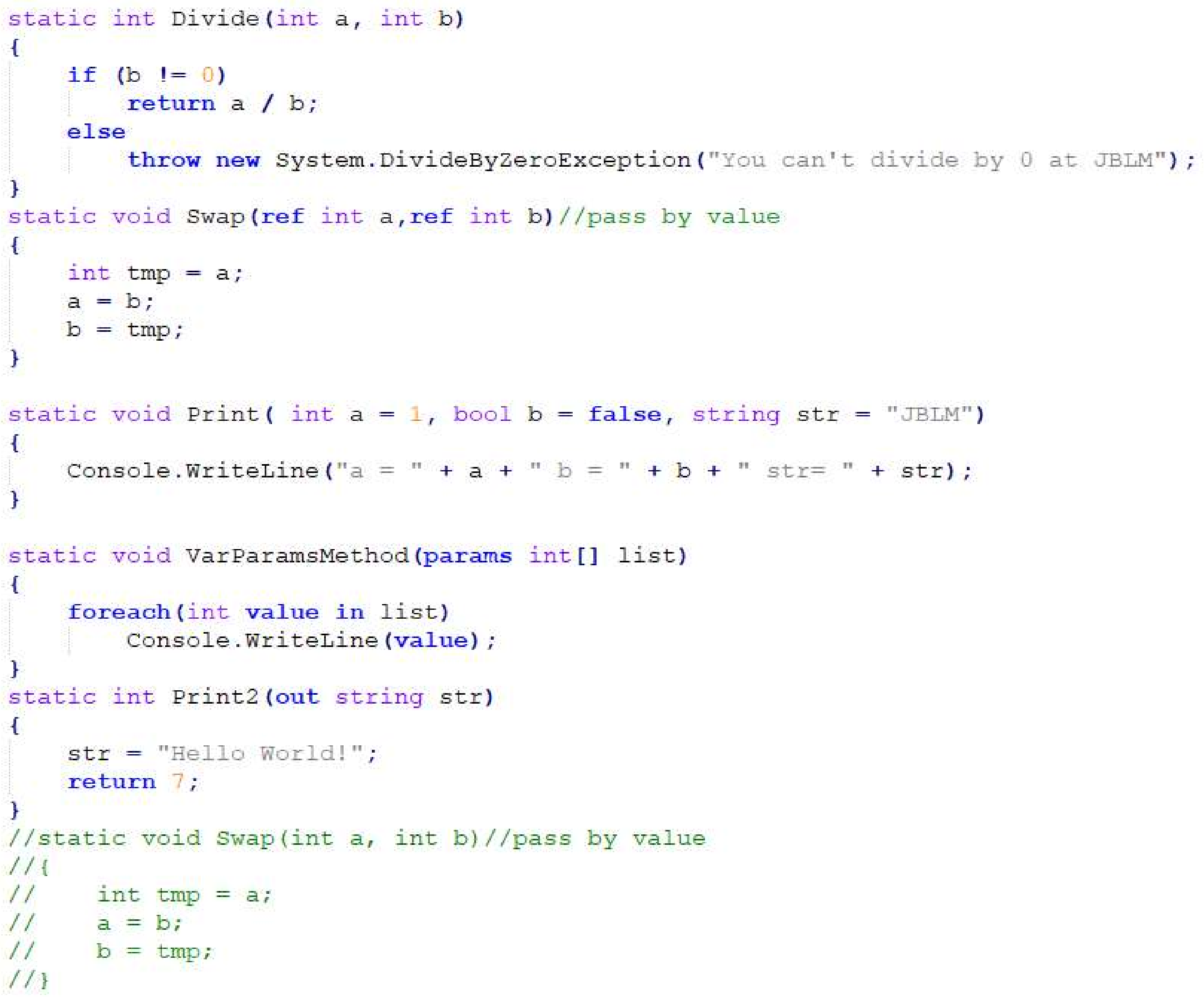
- Use the **out** keyword to define an output parameter

```
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);
```

Some example …



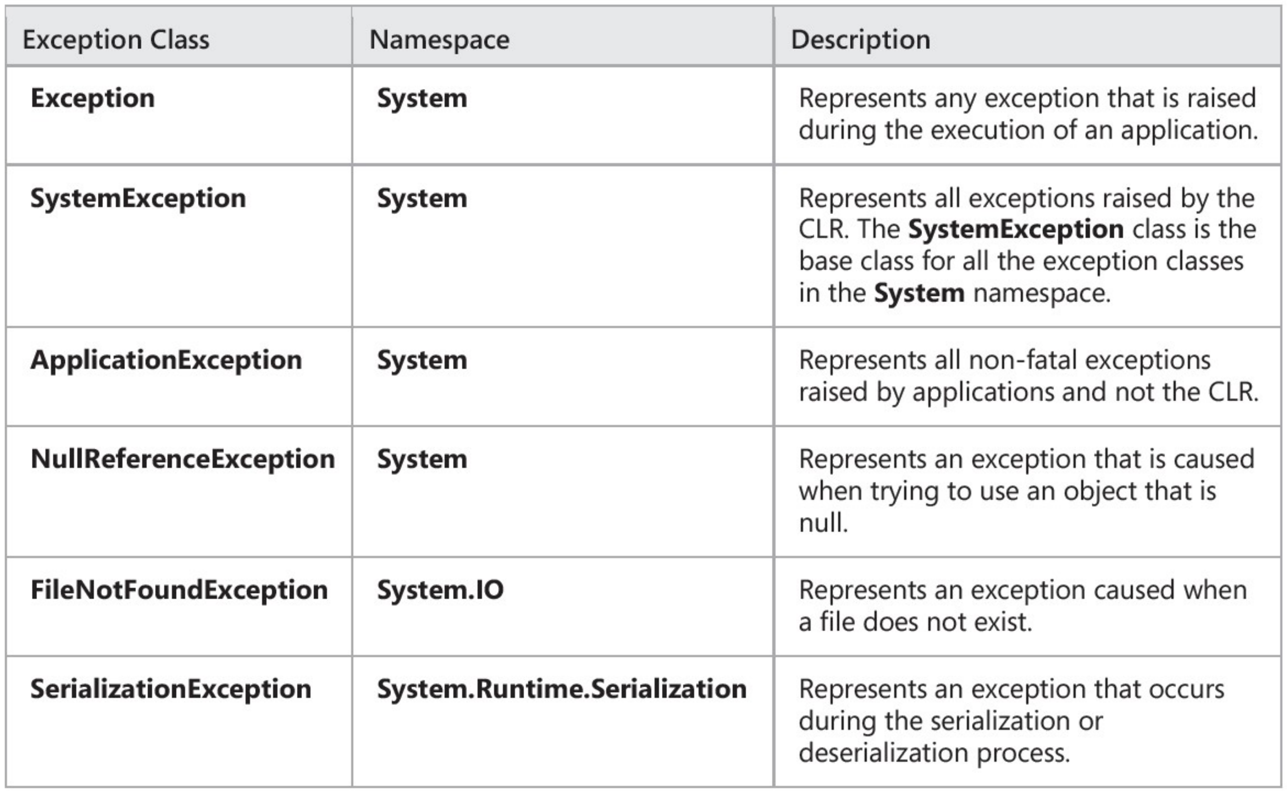
```
static int Divide(int a, int b)
{
    if (b != 0)
        return a / b;
    else
        throw new System.DivideByZeroException("You can't divide by 0 at JBLM");
}
static void Swap(ref int a,ref int b)//pass by value
{
    int tmp = a;
    a = b;
    b = tmp;
}

static void Print( int a = 1, bool b = false, string str = "JBLM")
{
    Console.WriteLine("a = " + a + " b = " + b + " str= " + str);
}

static void VarParamsMethod(params int[] list)
{
    foreach(int value in list)
        Console.WriteLine(value);
}
static int Print2(out string str)
{
    str = "Hello World!";
    return 7;
}
//static void Swap(int a, int b)//pass by value
//{
//    int tmp = a;
//    a = b;
//    b = tmp;
//}
```

Lesson 3:
Handling Exceptions

What Is an Exception?

- An exception is an indication of an error or exceptional condition

- The .NET Framework provides many exception classes:



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.

- Let's see what happens if divide by zero, or access out of range …

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.
}
```

- 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.

Using a Finally Block

- Use a finally block to run code whether or not an exception has occurred
  - Some methods may contain critical code that must always be run, even if an unhandled exception occurs.
    - E.g. closing a file if one was used …
  - 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.

```
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
  - 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.

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

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

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

Lesson 4:
Monitoring Applications

Using Logging

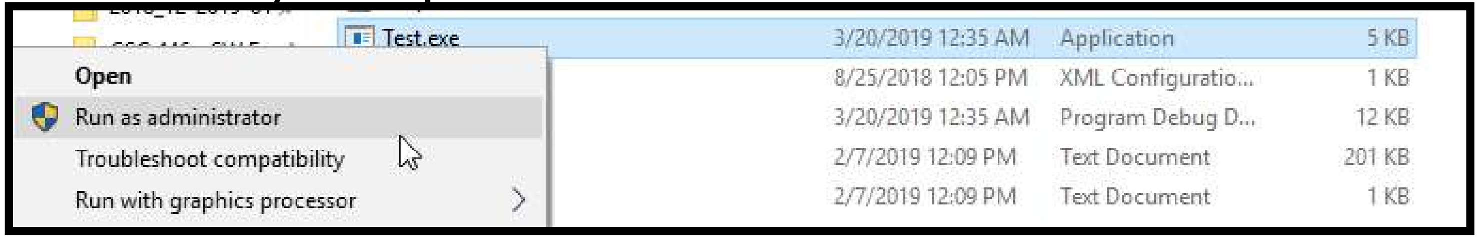
- *Logging* provides information to users and administrators
  - Windows event log ←**EventLog.WriteEntry** method
    - If your application does not run with sufficient permissions, it will throw a SecurityException when you attempt to create an event source or write to the event log.
  - Text files
  - Custom logging destinations

```
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);
```

Using Logging

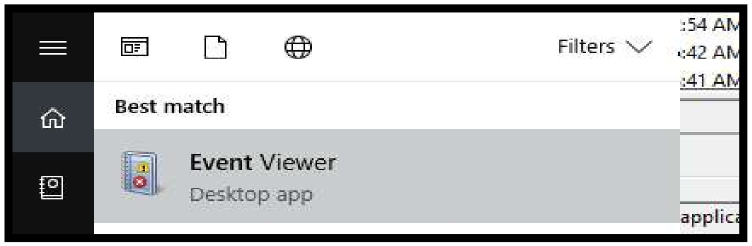
- To bypavoid the SecurityException run the **Run as administrator**:

  …

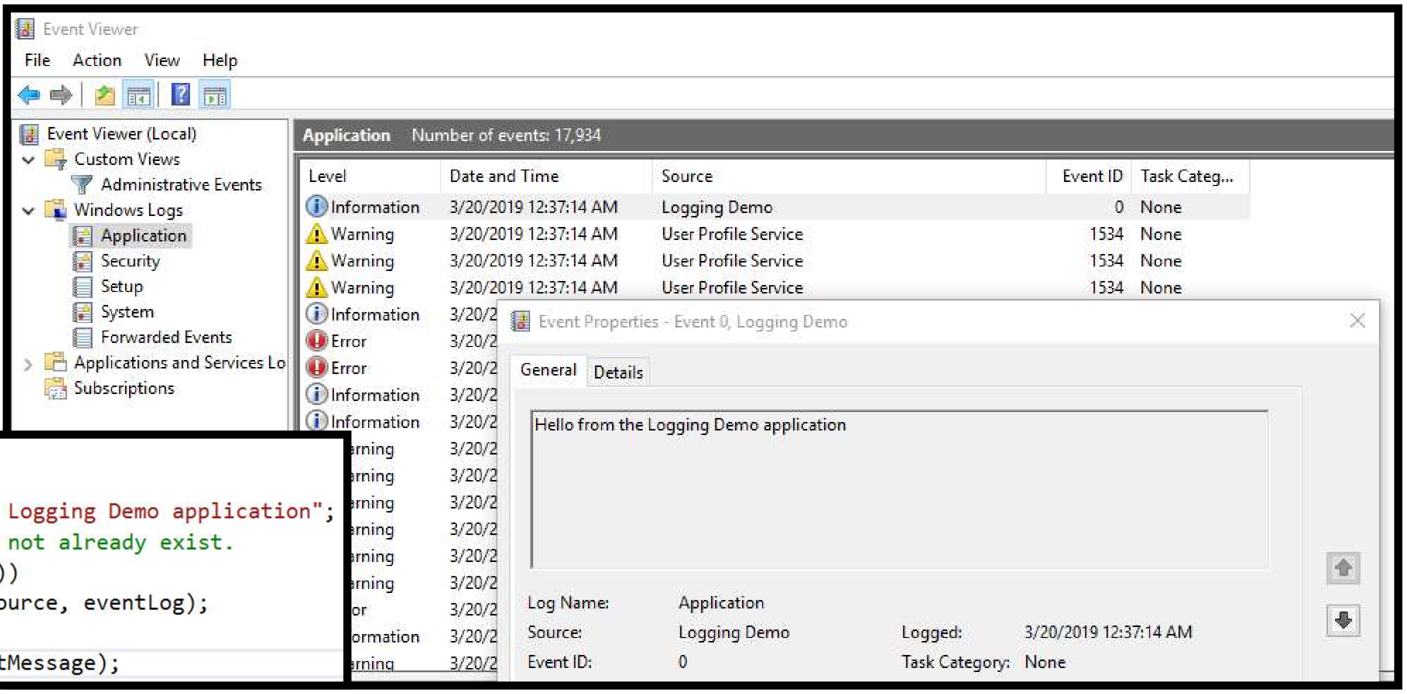


Test.exe	3/20/2019 12:35 AM	Application	5 KB
	8/25/2018 12:05 PM	XML Configuratio...	1 KB
	3/20/2019 12:35 AM	Program Debug D...	12 KB
	2/7/2019 12:09 PM	Text Document	201 KB
	2/7/2019 12:09 PM	Text Document	1 KB

- Then open event viewer:



- And see the logging entry:



Application Number of events: 17,934

Level	Date and Time	Source	Event ID	Task Categ...
Information	3/20/2019 12:37:14 AM	Logging Demo	0	None
Warning	3/20/2019 12:37:14 AM	User Profile Service	1534	None
Warning	3/20/2019 12:37:14 AM	User Profile Service	1534	None
Warning	3/20/2019 12:37:14 AM	User Profile Service	1534	None

Hello from the Logging Demo application

Log Name:	Application		
Source:	Logging Demo	Logged:	3/20/2019 12:37:14 AM
Event ID:	0	Task Category:	None

```
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);
```

Using Tracing

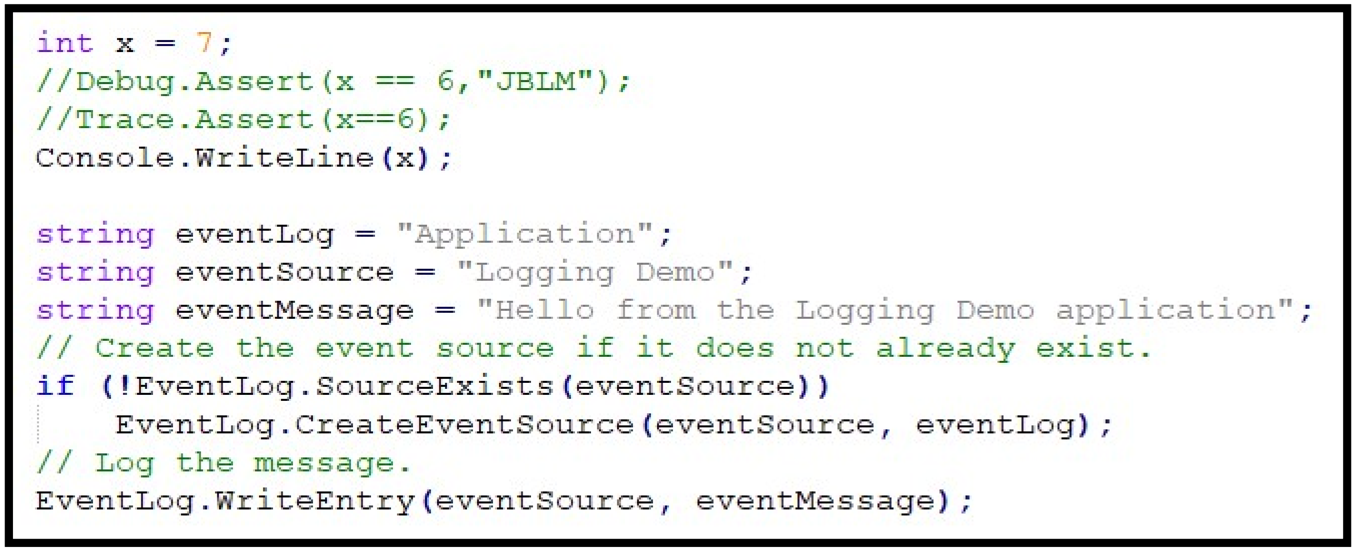
- *Tracing* provides information to developers
  - Visual Studio Output window
  - Custom tracing destinations

- Debug statements are only active if you build your solution in Debug mode,
- Trace statements are active in both Debug and Release mode builds.
  - Both Debug and Trace classes 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.

- Show a brief example if time …

Using Tracing

- Example:



```
int x = 7;
//Debug.Assert(x == 6,"JBLM");
//Trace.Assert(x==6);
Console.WriteLine(x);

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);
```

Module Review and Takeaways

- Review Question(s)
- **Question**
  - The return type of a method forms part of a methods signature.
    - (   )False
    - (   )True
- **Question**
  - When using output parameters in a method signature, which one of the following statements is true?
    - (   )Option 1: You cannot return data by using a return statement in a method that use output parameters.
    - (   )Option 2: You can only use the type object when defining an output parameter.
    - (   )Option 3: You must assign a value to an output parameter before the method returns.
    - (   )Option 4: You define an output parameter by using the output keyword.

Module Review and Takeaways

- Review Question(s)
- **Question**
  - A finally block enables you to run code in the event of an error occurring?
    - (   )False
    - (   )True
- **Question**
  - **Trace** statements are active in both **Debug** and **Release** mode builds.
    - (   )False
    - (   )True