Exception handling

Sometimes application does not work as expected. To help frame the upcoming discussion of structured exception handling, allow me to provide definitions for three commonly used anomaly-centric terms.

Bugs: These are, simply put, errors made by the programmer. For example, suppose you are programming with unmanaged C++. If you fail to delete dynamically allocated memory, resulting in a memory leak, you have a bug.

User errors: User errors, on the other hand, are typically caused by the individual running your application, rather than by those who created it. For example, an end user who enters a malformed string into a text box could very well generate an error if you fail to handle this faulty input in your codebase.

Exceptions: Exceptions are typically regarded as runtime anomalies that are difficult, if not impossible, to account for while programming your application. Possible exceptions include attempting to connect to a database that no longer exists, opening a corrupted XML file, or trying to contact a machine that is currently offline. In each of these cases, the programmer (or end user) has little control over these "exceptional" circumstances.

Given these definitions, it should be clear that .NET structured exception handling is a technique for dealing with runtime exceptions. However, even for the bugs and user errors that have escaped your view, the CLR will often generate a corresponding exception that identifies the problem at hand.

Exceptions occur in processor, when the processor detects an error condition of a program, in response to signals from hardware.

The Building Blocks of .NET Exception Handling Programming with structured exception handling involves the use of four interrelated entities.

- A class type that represents the details of the exception
- A member that throws an instance of the exception class to the caller under the correct circumstances
- A block of code on the caller's side that invokes the exception-prone member
- A block of code on the caller's side that will process (or catch) the exception, should it occur

The C# programming language offers five keywords (*try, catch, throw, finally*, and *when*) that allow you to throw and handle exceptions. The object that represents the problem at hand is a class extending *System.Exception*.

The try-catch..finally block in .NET allows developers to handle runtime exceptions. The syntax has three variations, ***try-catch***, ***try-finally***, and ***try-catch-finally***.

Try/catch/finally blocks

The common language runtime provides an exception handling model that is based on the representation of exceptions as objects, and the separation of program code and exception handling code into **try** blocks and **catch** blocks. There can be one or more *catch* blocks, *each designed to handle a particular type of exception,* or one block designed to catch a more specific exception than another block.

The rule to keep in mind is to make sure your catch blocks are structured such that the first catch is the most specific exception (i.e., the most derived type in an exception-type inheritance chain), leaving the final catch for the most general (i.e., the base class of a given exception inheritance chain, in this case System.Exception).

In essence, a try block is a section of statements that may throw an exception during execution. If an exception is detected, the flow of program execution is sent to the appropriate catch block. On the other hand, if the code within a try block does not trigger an exception, the catch block is skipped entirely.

The **try-catch** statement consists of a try block followed by one or more catch clauses, which specify handlers for different exceptions.

When an exception is thrown, the common language runtime (CLR) looks for the catch statement that handles this exception. If the currently executing method does not contain such a catch block, the CLR looks at the method that called the current method, and so on up the call stack. If no catch block is found, then the CLR displays an unhandled exception message to the user and stops execution of the program.

A try/catch scope may also define an optional finally block. The purpose of a **finally** block is to ensure that a set of code statements will always execute, exception (of any type) or not. When you need to dispose of objects, close a file, or detach from a database (or whatever), a finally block ensures a location for proper cleanup.

By using a finally block, you can clean up any resources that are allocated in a try block, and you can run code even if an exception occurs in the try block. Typically, the statements of a finally block run when control leaves a try statement. The transfer of control can occur as a result of normal execution, of execution of a break, continue, goto, or return statement, or of propagation of an exception out of the try statement. Within a handled exception, the associated finally block is guaranteed to be run.

Rethrowing exceptions

Exception objects that describe an error are created and then *thrown* with the throw keyword. The runtime then searches for the most compatible exception handler. With C# 7, throw is available as an expression as well and can be called anywhere expressions are allowed.

When you catch an exception, it is permissible for the logic in a try block to *rethrow* the exception up the call stack to the previous caller. To do so, simply use the throw keyword within a catch block. This passes the exception up the chain of calling logic, which can be helpful if your catch block is only able to partially handle the error at hand.

```
static void Main(string[] args)
{
        ...
        try
        {
                // Speed up car logic...
        }
        catch (CarIsDeadException e)
        {
                // Do any partial processing of this error and pass the buck.
                throw;
        }
        ...
}
```

Be aware that in this example code, the ultimate receiver of CarIsDeadException is the CLR because it is the Main() method rethrowing the exception. Because of this, your end user is presented with a systemsupplied error dialog box.

Notice as well that you are not explicitly rethrowing the CarIsDeadException object but rather making use of the throw keyword with no argument. You're not creating a new exception object; you're just rethrowing the original exception object (with all its original information). Doing so preserves the context of the original target.

Exception Filters – when keyword

Here, we have added a when clause to the CarIsDeadException handler to ensure the catch block is never executed on a Friday.

```
catch (CarIsDeadException e) when (e.ErrorTimeStamp.DayOfWeek !=
DayOfWeek.Friday)
{
        // This new line will only print if the when clause evaluates to true.
        Console.WriteLine("Catching car is dead!");
        Console.WriteLine(e.Message);
}
```

System.Exception Base Class

All exceptions ultimately derive from the System.Exception base class, which in turn derives from System.Object.

This class is the base class for all exceptions. When an error occurs, either the system or the currently executing application reports it by throwing an exception that contains information about the error. After an exception is thrown, it is handled by the application or by the default exception handler.

```
public class Exception :
System.Runtime.InteropServices.
_Exception, System.Runtime.Serialization.ISerializable
```

The Exception class implements two .Net interfaces. although you have yet to examine interfaces, understand that the *_Exception* interface allows a .Net exception to be processed by an unmanaged codebase (such as a CoM application), while the *ISerializable* interface allows an exception object to be persisted across boundaries (such as a machine boundary).

Constructors

- Exception()

  Initializes a new instance of the Exception class.

- Exception(String)

  Initializes a new instance of the Exception class with a specified error message.

- Exception(SerializationInfo, StreamingContext)

  Initializes a new instance of the Exception class with serialized data.

- Exception(String, Exception)

  Initializes a new instance of the Exception class with a specified error message and a reference to the inner exception that is the cause of this exception.

Properties

- public virtual string Message { get; }

This read-only property returns the textual description of a given error. The error message itself is set as a constructor parameter. The text of the **Message** property should completely describe the error and, when possible, should also explain how to correct the error. Top-level exception handlers may display the message to end-users, so you should ensure that it is grammatically correct and that each sentence of the message ends with a period.

- public virtual string Source { get; set; }

This property gets or sets the name of the assembly, or the object, that threw the current exception. In other words we can say that this property gets or sets the name of the application or the object that causes the error. If the **Source** property is not set explicitly, the runtime automatically sets it to the name of the assembly in which the exception originated.

- public virtual string StackTrace { get; }

The execution stack keeps track of all the methods that are in execution at a given instant. A trace of the method calls is called a stack trace. The stack trace listing provides a way to follow the call stack to the line number in the method where the exception occurs. The **StackTrace** property returns the frames of the call stack that originate at the location where the exception was thrown. You can obtain information about additional frames in the call stack by creating a new instance of the System.Diagnostics.StackTrace class and using its StackTrace.ToString method. As you might guess, this property is useful during debugging or if you want to dump the error to an external error log. . Be aware that you never set the value of StackTrace, as it is established automatically at the time the exception is created.

- public System.Reflection.MethodBase TargetSite { get; }

This read-only property returns a MethodBase object (gets the method that throws the current exception), which describes numerous details about the method that threw the exception (invoking ToString() will identify the method by name). If the method that throws this exception is not available and the stack trace is not a null reference (Nothing in Visual Basic), **TargetSite** obtains the method from the stack trace. If the stack trace is a null reference, TargetSite also returns a null reference.

```
static void Main(string[] args)
{
        Exception exInstance = new Exception("User created Exception");
        try
        {
                throw exInstance;
        }
        catch(Exception e)
        {
                Console.WriteLine("\n*** Error! ***");
```

```
                Console.WriteLine($"Member name: (TargetSite) {e.TargetSite}");
                Console.WriteLine($"Class defining member (TargetSite.DeclaringType):
{e.TargetSite.DeclaringType}");
                Console.WriteLine($"Member type: {e.TargetSite.MemberType}");
                Console.WriteLine($"Message: {e.Message}");
                Console.WriteLine($"Source (project name) : {e.Source}");
                Console.WriteLine($"Stack: {e.StackTrace}");
            }
        }

        // *** Error! ***
        Member name: (TargetSite) Void Main(System.String[])
        Class defining member (TargetSite.DeclaringType): Sevagir.Program
        Member type: Method
        Message: User created Exception
        Source (project name) : Sevagir
        Stack:    at Sevagir.Program.Main(String[] args) in
    C:\Users\User\source\repos\Sevagir\Sevagir\Program.cs:line 23
```

- `public virtual System.Collections.IDictionary Data { get; }`

This read-only property retrieves a collection of key-value pairs (represented by an object implementing IDictionary) that provide additional, programmer-defined information about the exception. By default, this collection is empty. The key component of each key/value pair is typically an identifying string, whereas the value component of the pair can be any type of object.

The Data property is useful in that it allows you to pack in custom information regarding the error at hand, without requiring the building of a new class type to extend the Exception base class. As helpful as the Data property may be, however, it is still common for .NET developers to build strongly typed exception classes, which handle custom data using strongly typed properties.

- `public virtual string HelpLink { get; set; }`

This property gets or sets a link to the help file or web site describing the error in full detail. The return value, which represents a help file, is a URN or URL.

```
        Exception exInstance = new Exception("User created Exception");
        exInstance.HelpLink = "https://msdn.microsoft.com/en-us/";
        try
        {
            throw exInstance;
        }
        catch(Exception e)
        {
            Console.WriteLine($"Help link: {exInstance.HelpLink}");
        }
```

- public int HResult { get; protected set; }

Gets or sets **HRESULT**, a coded numerical value that is assigned to a specific exception. HRESULT is a 32-bit value, divided into three different fields: a severity code, a facility code, and an error code. The severity code indicates whether the return value represents information, warning, or error. The facility code identifies the area of the system responsible for the error. The error code is a unique number that is assigned to represent the exception. Each exception is mapped to a distinct HRESULT. When managed code throws an exception, the runtime passes the HRESULT to the COM client. When unmanaged code returns an error, the HRESULT is converted to an exception, which is then thrown by the runtime.

- public Exception InnerException { get; }

This read-only property can be used to obtain information about the previous exceptions that caused the current exception to occur. The previous exceptions are recorded by passing them into the constructor of the most current exception. So, this property gets the *Exception* instance that caused the current exception. When an exception X is thrown as a direct result of a previous exception Y, the **InnerException** property of X should contain a reference to Y.

Methods

- public virtual Exception GetBaseException();

A chain of exceptions consists of a set of exceptions such that each exception in the chain was thrown as a direct result of the exception referenced in its InnerException property. For a given chain, there can be exactly one exception that is the root cause of all other exceptions in the chain. This exception is called the base exception and its InnerException property always contains a null reference.

For all exceptions in a chain of exceptions, the GetBaseExceptionmethod must return the same object (the base exception).

Use the GetBaseException method when you want to find the root cause of an exception but do not need information about exceptions that may have occurred between the current exception and the first exception.

- public virtual void GetObjectData(System.Runtime.Serialization.SerializationInfo info, System.Runtime.Serialization.StreamingContext context);

GetObjectData sets a SerializationInfo with all the exception object data targeted for serialization. During deserialization, the exception is reconstituted from the SerializationInfo transmitted over the stream.

System-Level Exceptions (System.SystemException)

The .NET base class libraries define many classes that ultimately derive from System.Exception. For example, the System namespace defines core exception objects such as ArgumentOutOfRangeException, IndexOutOfRangeException, StackOverflowException, and so forth. Other namespaces define exceptions that reflect the behavior of that namespace. For example, System.Drawing.Printing defines printing exceptions, System.IO defines input/output-based exceptions, System.Data defines database-centric exceptions, and so forth.

Exceptions that are thrown by the .NET platform are (appropriately) called ***system exceptions****.* These exceptions are generally regarded as nonrecoverable, fatal errors. System exceptions derive directly from a base class named System.SystemException, which in turn derives from System.Exception (which derives from System.Object).

Given that the System.SystemException type does not add any additional functionality beyond a set of custom constructors, you might wonder why SystemException exists in the first place. Simply put, when an exception type derives from System.SystemException, you are able to determine that the .NET runtime is the entity that has thrown the exception, rather than the codebase of the executing application. You can verify this quite simply using the is keyword.

```
// True! NullReferenceException is-a SystemException.
NullReferenceException nullRefEx = new NullReferenceException();
Console.WriteLine($"NullReferenceException is-a SystemException? : {nullRefEx is SystemException}");
```

Application-Level Exceptions (System.ApplicationException)

Given that all .NET exceptions are class types, you are free to create your own application-specific exceptions. However, because the System.SystemException base class represents exceptions thrown from the CLR, you might naturally assume that you should derive your custom exceptions from the System.Exception type. You could do this, but you could instead derive from the ***System.ApplicationException*** class.

Like SystemException, ApplicationException does not define any additional members beyond a set of constructors. Functionally, the only purpose of System.ApplicationException is to identify the source of the error. When you handle an exception deriving from System.ApplicationException, you can assume the exception was raised by the codebase of the executing application, rather than by the .NET base class libraries or .NET runtime engine.

Building Custom Exceptions

If you want to build a truly prim-and-proper custom exception class, you want to make sure your type adheres to .NET best practices. Specifically, this requires that your custom exception does the following:

- Derives from Exception/ApplicationException
- Is marked with the [System.Serializable] attribute
- Defines a default constructor
- Defines a constructor that sets the inherited Message property
- Defines a constructor to handle "inner exceptions"
- Defines a constructor to handle the serialization of your type

```
[Serializable]
public class MyException : Exception
{
        public MyException() { }
        public MyException(string message) : base(message) { }
        public MyException(string message, Exception inner) : base(message, inner) { }
        protected MyException(
                SerializationInfo info,
                StreamingContext context) : base(info, context) { }
}
```

Visual Studio provides a code snippet template named Exception, that will autogenerate a new exception class that adheres to .NET best practices (pressing exception and the Tab key twice).