Chapter 9

Exception Handling

In this chapter:
Exception Example. ................................................................. 326
Common Exception Model. ......................................................... 327
Structured Exception Handling .................................................. 327
System.Exception ................................................................. 335
Remote Exceptions ............................................................... 345
Unhandled Exceptions ......................................................... 347
Managing Exceptions in Visual Studio ........................................ 351
Metadata and Reflections ...................................................... 352

Exception handling is an important ingredient of a robust application and should be included in the application design of every .NET program. Exception handling helps applications identify and respond to exceptional events in a known and robust manner. This enhances the correctness of an application, which naturally improves customer satisfaction. Unfortunately, exception handling is often an afterthought, which leads to poorly implemented solutions for exception handling and a less robust application. Starting with application design, exception handling should be included in all aspects of application planning and development.

What is an exception? Exceptions are exceptional events or error conditions within an application and are categorized as system or application exceptions. System exceptions are raised by the Common Language Runtime (CLR) and include null reference, out of memory, divide by zero, and stack overflow exceptions. Application exceptions, considered custom exceptions, are thrown by the application, not by the CLR. Some exceptional events or error conditions are detectable by application logic, not by the runtime. Application exceptions are useful in those circumstances. As an example, constructors that fail are not always detectable by the CLR. In addition, constructors implicitly return void, which prevents returning an error code. For these reasons, throwing an application exception in the failed constructor is the best solution.

Exceptions should be used only for exceptions. Exception handling is not a substitute for transfer of control or a goto statement. Exceptions are expensive when compared with conventional transfer of control. Whenever possible, applications should preempt exceptions and avoid the accompanying costs. Performing data validation, where incorrect values can cause
exceptions, is one measure that prevents exceptions. Data validation and, if necessary, returning an error code is undoubtedly cheaper than raising an exception.

The overuse of exceptions makes code harder to read and maintain. Errors that frequently occur should not be handled with exception handling, but with error codes or returns. Remember, exceptions should be reserved for exceptional events.

Exception Example

A common exception is *dividing by zero*. Exceptions are generally linked to an action. For dividing by zero, the action is a divisor of zero. Integer division, where the divisor is zero, triggers a divide by zero exception. (However, floating-point division by zero does not cause an exception; instead, infinity is returned.) The following code causes an unhandled divide by zero exception, which terminates the application:

```csh
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            int var1=5, var2=0;
            var1 /= var2; // exception occurs
        }
    }
}
```

Place code that is likely to raise an exception in a *try* block because code in a *try* block is protected from exceptions. Exceptions raised in the *try* block are trapped. The stack is then walked by the CLR, searching for the appropriate exception handler. Code not residing in a *try* block is unprotected from exceptions. In this circumstance, the exception eventually evolves into an unhandled exception. As demonstrated in the previous code, an unhandled exception is apt to abort an application.

In the following code, the divide by zero exception is caught in a *try* block. Trapping, catching, and handling an exception are separate tasks. The *catch* statement consists of a *catch* filter and block. The *DivideByZeroException* filter catches the divide by zero exception. The *catch* block handles the exception, which is to display the stack trace. The proximity of the infraction is included in the stack trace. Execution then continues at the first statement after the *catch* block.

```csh
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            try {
                int var1=5, var2=0;
```

Common Exception Model

.NET offers a common exception model. Exception handling is implemented within the CLR and is not specific to a managed language. Each managed language exposes exception handling using language-specific syntax. The uniform exception model contributes to language independence, which is an important tenet of .NET. In addition, exceptions raised in one managed language can be caught in a different managed language. Prior to .NET, there were competing models for error handling: Microsoft Visual Basic 6.0, Win32 SDK, MFC, COM, and more. In a component-driven architecture, this disparity contributed to complexity and potential product instability. In .NET, exception handling between disparate components is consistent, simple, and stable.

As part of the common exception model, .NET advocates structured exception handling, which is essentially the try and catch statements. C-based and Java developers are probably familiar with structured exception handling. For Visual Basic developers, this is a seismic shift from unstructured exception handling. The unstructured model, such as on error goto and on error resume, is thankfully now obsolete.

Structured Exception Handling

A structured exception evaluates the stack to determine when code is protected and where an exception is caught and handled. When exceptions occur, the stack is walked in search of an exception handler. When a matching catch filter is located, the exception is handled in the exception handler, which is the catch block. This sets the scope of the exception on the stack. After the exception is handled, the CLR unwinds the stack to that scope.

Try Statement

Try blocks are observers; they watch for exceptions in protected code. Place code prone to raising exceptions in a try block. Do not attempt to protect an entire application in a try block—there are more convenient and practical means of accomplishing the same feat, as described later in this chapter.

As mentioned, exceptions are stack-based. The following code contains a fence-post error that happens when the bounds of the array are exceeded. The result is an exception in the unprotected MethodA method. Main calls MethodA, where the scope of Main encompasses MethodA.
For this reason, the \textit{try} block in \textit{Main} extends protection to \textit{MethodA}. The exception is trapped in \textit{Main}.

using System;

namespace Donis.CSharpBook{
  public class Starter{

    public static void Main(){
      try {
        MethodA();
      } catch(Exception except) {
        Console.WriteLine(except.Message);
      }
    }

    public static void MethodA() {
      int [] values={1,2,3,4};
      for(int count=0; count<=values.Length; ++count) {
        Console.WriteLine(values[count]);
      }
    }
  }
}

This is the syntax of the \textit{try} statement:

\begin{verbatim}
try { protected} catch\textsuperscript{1}(filter\textsuperscript{1}) { handler\textsuperscript{1}} catch\textsuperscript{n}(filter\textsuperscript{n}) { handler\textsuperscript{n}} finally {terminationhandler}
\end{verbatim}

\textit{Try} statements must be paired with either a \textit{catch} statement or a \textit{finally} statement. There can be zero or more \textit{catch} statements attached to a \textit{try} statement; there are zero or one \textit{finally} statements. If both \textit{catch} and \textit{finally} are present, the \textit{catch} statement should precede the \textit{finally} statement. The following code has various combinations of \textit{try}, \textit{catch}, and \textit{finally} statements:

\begin{verbatim}
// try..catch
try {
}
catch(Exception e) {
}

// try..finally
try {
}
finally{
}

// try..catch..finally
try {
}
catch(Exception e) {
}
finally{
}
\end{verbatim}
// try..catches..finally
try {
    try {  
    catch(Exception e) {  

catch {  

finally{
    }

---

**Catch Statement**

*Catch* statements filter and handle exceptions. When an exception is raised, the filter sets which exceptions are handled at that scope in the stack. If the filter matches the exception, the exception is suppressed, and control is transferred to the adjoining *catch* block to be handled. The CLR searches the stack for an appropriate filter. If a matching filter is not found, the exception becomes an unhandled exception.

Exceptions must be derived from *System.Exception*. The *catch* filter defines an exception type and catches exception of that type or any descendants. The exception object contains details of the exception, such as a user-friendly message describing the exception. The exception object caught in the filter is accessible only in the *catch* block. *System.Exception* is the base class of .NET exceptions and the generic filter.

A *System.Exception* filter catches all managed exceptions. Derived classes of *System.Exception* catch more-specific exceptions. In the previous code, the *DivideByZeroException* filter caught integer divide by zero exceptions and nothing else. A *try* block can be appended to multiple catches for catching distinct exceptions. The catches are ordered from specific to generic. The following code has several exception filters, from specific to generic:

```csharp
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            try {
                int var1=5, var2=0;
                var1/=var2;
            }
            catch(DivideByZeroException except) {
                Console.WriteLine("Exception "+except.Message);
            }
            catch(System.ArithmeticException except) {
                Console.WriteLine("Exception "+except.Message);
            }
            catch(Exception except) {
                Console.WriteLine("Exception "+except.Message);
            }
        }
    }
}
```
In the preceding code, `DivideByZeroException` is very specific and catches only divide by zero exceptions. `ArithmeticException` is less specific and catches a variety of arithmetic exceptions, including the divide by zero exception. `Exception` catches all managed exceptions, which includes divide by zero and arithmetic exceptions. In the preceding code, the exception is caught at the `DivideByZeroException` catch handler.

The `catch` filter is optional, and the default is `catch all`. Although `System.Exception` catches any managed exceptions, `catch all` catches both managed and unmanaged exceptions. In most circumstances, native exceptions are mapped to managed exceptions, which are thrown in the `RaiseTheException` native function. `RaiseTheException` uses the `RaiseException` API to construct the managed exception. Exceptions not mapping to managed exceptions are outside this normal mechanism. A catch with no `catch` filter is a `catch all`. This code has a `catch all`:

```csharp
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            try {
                int var1=5, var2=0;
                var1/=var2;
            }
            catch(DivideByZeroException except) {
                Console.WriteLine("Exception "+except.StackTrace);
            }
            catch {
                // catch remaining managed and unmanaged exceptions
            }
        }
    }
}
```

### Propagating Exceptions

Exceptions are not always handled locally where the exception is caught. It is sometimes beneficial to catch an exception and then propagate the exception. Propagating an exception is catching and then rethrowing the exception. Rethrowing an exception continues the search along the call stack to find an appropriate handler. Here are some reasons to propagate an exception:

- There is a centralized handler for the exception. There are several reasons for implementing centralized handlers, including code reuse. Instead of handling an exception in various locations in an application, concentrate code for certain exceptions in a centralized handler. Wherever the exception is raised, the proper response is to record the exception and then delegate to the centralized handler. A central handler can be used to handle all exceptions in a single place.
■ Resources required to handle the exception are not available locally. For example, an exception is raised because of an invalid database connection. However, the correct connection string is read from a file not available where the exception occurs. The solution is to propagate the exception to a handler that has access to the file resource.

■ Propagate unwanted exceptions caught in the umbrella of the exception filter. This would be useful for catching all DataException types with the exception of the DuplicateNameException. One solution would be to write 12 individual catches—one for each of the data exceptions except for the DuplicateNameException exception. A better solution is to catch the DataException type and propagate the DuplicateNameException when necessary. This is one catch statement versus 12 catch statements and eliminates redundant code.

■ Catch an exception to gather information or to report on an exception, and then propagate the exception.

To propagate an exception, rethrow the same exception or another exception in the catch block. An empty throw statement propagates the caught exception. Alternatively, throw a different exception.

Exceptions might propagate through several layers of an application. Ultimately, the exception could percolate to the user interface level. As an exception percolates, the exception becomes less specific. Exceptions from the lower echelon of an application contain detailed information appropriate to the application developer, but probably not relevant to the user. Internal exceptions might contain security and other sensitive information not appropriate for a benign (or malicious) user. Record the facts of the internal exception if logging is preferable. Exceptions that reach the user should present user-relevant information: a user-friendly message, steps to resolve the exception, or even a customer support link.

When an original exception is rethrown, you can preserve the that exception in the InnerException attribute. Successive InnerException attributes form a chain of exceptions from the current exception to the original exception. The InnerException can be initialized in the constructor of the new exception. Here is sample code that propagates an exception and sets the inner exception:

```csharp
using System;
namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            try {
                MethodA();
            }
            catch(Exception except) {
                Console.WriteLine(except.Message);
            }
        }
    }
}
```
Finally Statement

The *finally* block is the termination handler. When an exception is raised, protected code after the infraction is not executed. What if it is cleanup code? If cleanup code is not executed, resources are left dangling. Termination handlers are the solution. Code that must execute, regardless of an exception occurring, is placed in a termination handler. When an exception is raised, the *finally* blocks within scope are called before the stack is unwound. Note that the termination handler is executed even when there is no occurrence of an exception. Execution simply falls through the *try* block into the attached *finally* block. In the termination handler, you could close a file, release a database connection, or otherwise manage resources.

Here is a typical termination handler:

```csharp
using System;
using System.IO;

namespace Donis.CSharpBook{
    public class FileWriter{
        public static void Main(){
            StreamWriter sw=null;
            try {
                sw=new StreamWriter("date.txt");
                sw.Write(DateTime.Now.ToString());
                throw new ApplicationException("exception");
                // dangling code
            }
            finally {
                sw.Close();
            }
        }
    }
}
```
Console.WriteLine("file closed");

Exception Information Table

The CLR refers to the Exception Information Table to track protected code in an efficient manner. Because of the Exception Information Table, there is no overhead associated with an exception unless an exception occurs.

An Exception Information Table is constructed for every managed application. The table has an entry for each method in the program. Each entry is an array, where the array elements describe the filters and handlers of that method. Entries in the array represent a catch filter, user-filtered handler, catch handler, or termination handler. User-filtered handlers use the when clause and are available in Visual Basic .NET, but not in C#.

When an exception occurs, the CLR consults the Exception Information Table. The entry for the method hosting the exception is searched for a filter that matches the exception. If the array is empty or a matching filter is not found, the entry of the outer method is examined next. When the boundary of the application is reached, the exception is considered unhandled.

Nested Try Blocks

Try blocks can be nested. The order of evaluation is more complex with nested try blocks. This is the order of execution when an exception occurs:

1. The first step is to find an appropriate try block. If an exception is raised outside a protected block, the exception is not trapped and is therefore unhandled.
2. From the try block, walk the stack until a matching catch filter is found. If a matching filter is not found, the exception is unhandled. This defines the scope of the exception.
3. Before the stack is unwound, finally blocks within the scope of the exception are run. The innermost finally blocks are executed first.
4. The catch block of the filter is executed as the exception handler.
5. Execution continues at the first statement after the catch block.
6. Finally blocks at scope of exception are executed.
Figure 9-1 diagrams the sequence when an exception is raised in a nested try block.

The following code has several nested try blocks:

```csharp
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            try {
                // protected code
                throw new Exception("Stack exhausted?");
            } catch (Exception e) {
                // appropriate catch filter
                if (e is StackOverflowException) {
                    // walk stack
                } else {
                    // unhandled exception
                }
            } finally {
                // execute any termination handler within scope
            }
            catch (Exception e) {
                // catch handler
            }
        }
    }
}
```
System.Exception

ArrayList is the base exception class. All exceptions in .NET are derived from System.Exception. System.SystemException is the base class for system exceptions raised by the CLR, such as System.Data.DataException or System.FormatException. SystemException is derived directly from System.Exception. System.SystemException does not refine System.Exception. However, it is an important marker that distinguishes between system and application exception, as demonstrated in the following code:

```
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            try {
                int var1=5, var2=0;
                var1/=var2; // exception occurs
            }
            catch(Exception except) {
                if(except is SystemException) {
                    Console.WriteLine("Exception thrown by runtime");
                } else {
                    Console.WriteLine("Exception thrown by application");
                }
            }
        }
    }
}
```
**System.Exception Functions**

*System.Exception* has four constructors:

- `public Exception()`
- `public Exception(string message)`
- `public Exception(string message, Exception innerException)`
- `protected Exception(Serialization info, StreamingContext context)`

`Exception` is the default constructor. `Exception` constructor sets the user-friendly message of the exception. `Exception` constructor also sets the inner exception, which is the originating exception. `Exception` deserializes an exception raised remotely.

The `Exception` class has several other helpful functions. Table 9-1 lists the important methods of the class.

### Table 9-1  Exception Methods

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetBaseException</td>
<td>Returns the root exception in a chain of exception objects</td>
</tr>
<tr>
<td>GetObjectData</td>
<td>Serializes data of the <em>Exception</em> class</td>
</tr>
<tr>
<td>GetType</td>
<td>Returns the type of the exception</td>
</tr>
<tr>
<td>ToString</td>
<td>Concatenates the name of the exception object with the user-friendly message</td>
</tr>
</tbody>
</table>

The following code calls `GetBaseException` and outputs the error message of the initial exception. If the current exception is the first exception in a chain of exceptions, `GetBaseException` returns null. Alternatively, you can walk `InnerException` properties back to the first exception.

```csharp
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            try {
                MethodA();
            }
            catch(Exception except) {
                Exception original=except.GetBaseException();
                Console.WriteLine(original.Message);
            }
        }
        public static void MethodA(){
            try {
                MethodB();
            }
        }
    }
}
```
```csharp
    catch(Exception except) {
        throw new ApplicationException( "Inner Exception", except);
    }
}

public static void MethodB(){
    throw new ApplicationException("Innermost Exception");
}
```

### System.Exception Properties

System.Exception has a full complement of attributes providing information on the exception. Table 9-2 describes the properties of the Exception class.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Type</th>
<th>Read/Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Returns a dictionary collection that provides optional user-defined details of exception.</td>
<td>IDictionary</td>
<td>R</td>
</tr>
<tr>
<td>HelpLink</td>
<td>Link to a help file for the exception.</td>
<td>string</td>
<td>R/W</td>
</tr>
<tr>
<td>HRESULT</td>
<td>The HRESULT, which is a 32-bit error code common to COM, assigned to the exception. This is a protected property.</td>
<td>int</td>
<td>R/W</td>
</tr>
<tr>
<td>InnerException</td>
<td>When exceptions are propagated, the inner exception represents the previous exception.</td>
<td>Exception</td>
<td>R</td>
</tr>
<tr>
<td>Message</td>
<td>User-friendly message describing the exception.</td>
<td>string</td>
<td>R</td>
</tr>
<tr>
<td>Source</td>
<td>Name of application or object where exception occurred.</td>
<td>string</td>
<td>R/W</td>
</tr>
<tr>
<td>StackTrace</td>
<td>String representation of the call stack when the exception occurred.</td>
<td>string</td>
<td>R</td>
</tr>
<tr>
<td>TargetSite</td>
<td>Reference to the method where exception is raised.</td>
<td>MethodBase</td>
<td>R</td>
</tr>
</tbody>
</table>

The Message and InnerException properties are settable in constructors of the Exception class.

The following code uses some of the properties of the Exception class. In Main, MethodA is called, and an exception is raised. The exception is caught and handled in Main. In the catch block, the exception flag is set to false. Leveraging the TargetSite property, MethodA is then called again successfully. The TargetSite property returns a MethodBase type, which can be used to late bind and invoke a method.
Part III: More C# Language

```csharp
using System;
using System.Reflection;

namespace Donis.CSharpBook{
    public class Starter{
        public static bool bException=true;
        public static void Main(){
            try {
                MethodA();
            }
            catch(Exception except) {
                Console.WriteLine(except.Message);
                bException=false;
                except.TargetSite.Invoke(null, null);
            }
        }
        public static void MethodA() {
            if(bException) {
                throw new ApplicationException("exception message");
            }
        }
    }
}
```

**Application Exceptions**

Application exceptions are custom exceptions and are thrown by the application, not by the CLR. Application exceptions are derived from `System.ApplicationException` or `System.Exception`. `System.ApplicationException` adds nothing to `System.Exception`. While `System.SystemException` is a marker for system exceptions, `System.ApplicationException` brands application exceptions. A custom exception derived from `System.Exception` accomplishes the same feat. When several custom exceptions are planned, create a custom base exception class to categorize the exceptions. For convenience and maintainability, deploy application exceptions together in a separate assembly.

Do not create an application exception for an existing exception. Research the available system exceptions to avoid replicating an existing exception.

These are the steps for creating an application exception:

1. Name the application exception. As a best practice, the class name should have the `Exception` suffix, as in `ApplicationException`.
2. Derive the application exception from `System.Exception`.
3. Define constructors that initialize the state of the application exception. This includes initializing members inherited from the base class.
4. Within the application exception, refine `System.Exception` as desired, such as by adding attributes that further delineate this specific exception.
To raise an application exception, use the `throw` statement. You can also throw system exceptions. Thrown exceptions are considered software exceptions. The CLR treats software exceptions as standard exceptions.

`throw` syntax:

```
    throw exceptioninstance1;
    throw2;
```

The second syntax is specialized: It is available in the `catch` block, but nowhere else. This version of the `throw` statement rethrows an exception caught in the `catch` block. However, the best policy is to add additional context to an exception before propagating the exception object. Propagating exceptions is reviewed later in this chapter.

Application exceptions are typically prompted by an exceptional event. What is an exceptional event? A strict definition does not exist. You define the basis of the event using whatever criteria are appropriate. Remember, raising an exception simply for transfer of control or a nonexceptional event is bad policy. In an application, the following could be considered exceptional events where throwing an application exception is warranted:

- Constructor fails to initialize the state of an object.
- A property does not pass validation.
- Null parameters.
- An exceptional value is returned from a function.

`ConstructorException` is an application exception. Throw this exception when a constructor fails. It refines the `System.Exception` base class with name of the type and time of exception. In addition, the `Message` property is assigned a congruous message. This is the code for the `ConstructorException` class:

```csharp
using System;

namespace Donis.CSharpBook{

    public class ConstructorException: Exception{
        public ConstructorException(object origin)
            : this(origin, null) {
        }

        public ConstructorException(object origin, Exception innerException)
            : base("Exception in constructor", innerException) {
            prop_Typename=origin.GetType().Name;
            prop_Time=DateTime.Now.ToLongDateString() + " " +
            DateTime.Now.ToShortTimeString();
        }
    }

    public class Program{
        static void Main(string[] args){
        }
    }
}
```
protected string prop_Typename=null;
public string Typename {
    get {
        return prop_Typename;
    }
}

protected string prop_Time=null;
public string Time {
    get {
        return prop_Time;
    }
}

This code uses the ConstructorException class:

using System;
namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            try {
                ZClass obj=new ZClass();
            }
            catch(ConstructorException except) {
                Console.WriteLine(except.Message);
                Console.WriteLine("Typename: "+except.Typename);
                Console.WriteLine("Occured: "+except.Time);
            }
        }
    }
}

class ZClass {
    public ZClass() {
        // initialization fails
        throw new ConstructorException(this);
    }
}

Exception Translation

In some circumstances, the CLR catches an exception and rethrows a different exception. The inner exception of the translated exception contains the original exception. Invoking a method dynamically using reflection is one such circumstance. Exceptions raised in methods invoked by MethodInfo.Invoke are automatically trapped and converted to TargetInvocationException. .NET documentation in MSDN will always confirm when exception translation will happen. Here is an example of exception translation:

using System;
using System.Reflection;
namespace Donis.CSharpBook{

    public class ZClass {
        public static void MethodA() {
            Console.WriteLine("ZClass.MethodA");
            throw new Exception("MethodA exception");
        }
    }

    public class Starter{
        public static void Main(){
            try {
                Type zType=typeof(ZClass);
                MethodInfo method=zType.GetMethod("MethodA");
                method.Invoke(null, null);
            }
            catch(Exception except) {
                Console.WriteLine(except.Message);
                Console.WriteLine("original: "+
                        except.InnerException.Message);
            }
        }
    }
}

COM Interoperability Exceptions

.NET applications often host COM components or expose managed components to COM clients. These applications must be prepared to handle and possibly throw COM exceptions, respectively. The prevalence of COM components makes COM interoperability an important consideration for managed applications into the foreseeable future.

COM Exceptions

COM components should sandbox exceptions, which protects COM clients from potential language-specific exceptions. COM methods return an HRESULT structure, which is the result code of the method. An HRESULT is a 32-bit structure, where the severity bit is the high-order bit. The severity bit is set if an exception is raised. The Win32 SDK defines constants representing various HRESULT codes. E_NOINTERFACE, E_INVALIDARG, E_OUTOFMEMORY, S_OK, and S_FALSE are common HRESULT codes. E_codes are error codes indicating that an exception was raised or some other exceptional event happened. S_codes are success codes where no failure occurred.

When managed components call methods on COM objects, the CLR consumes the resulting HRESULT. If the HRESULT represents a known COM error (E_code), the CLR maps the HRESULT to a managed exception. For example, E_POINTER maps to the NullReference-Exception, which is a managed exception. An error code from an unknown HRESULT is mapped to a COMException object. No exception is thrown if the HRESULT is a success code (S_code). Table 9-3 shows the common translations of HRESULT to managed exceptions.
The `COMException` is derived from `System.Runtime.InteropServices.ExternalException`, which indirectly derives from `System.Exception`. The `COMException` class has additional properties providing the details of the unknown COM exception. The `ErrorCode` property contains the unrecognized HRESULT from the COM method.

COM components implement `Error` objects to provide extended error information to clients. An `Error` object implements the `IErrorInfo` interface. Members of the `IErrorInfo` interface correlate to members of the `COMException` class and are therefore accessible to the managed client. Table 9-4 maps members of the `Error` object to the `COMException` class.

### Table 9-4   `IErrorInfo` to `COMException` Mapping

<table>
<thead>
<tr>
<th><code>IErrorInfo</code> Member</th>
<th><code>COMException</code> Member</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IErrorInfo::GetDescription</code></td>
<td><code>COMException.Message</code></td>
</tr>
<tr>
<td><code>IErrorInfo::GetSource</code></td>
<td><code>COMException.Source</code></td>
</tr>
<tr>
<td>If <code>IErrorInfo::GetHelpFile</code> is non-zero,</td>
<td></td>
</tr>
<tr>
<td><code>IErrorInfo::GetHelpFile + &quot;#&quot; + IErrorInfo::HelpContext</code></td>
<td><code>COMException.HelpLink</code></td>
</tr>
</tbody>
</table>

The following code is a partial listing from an ATL project that publishes a COM component. The COM component exposes the `CComponentZ::MethodA`. Using the `AtlReportError` API, `CComponentZ::MethodA` builds an `Error` object to return extended error information to the client. The method also returns a custom error code in the HRESULT, which will be unknown to the CLR.

```
// ComponentZ.cpp : Implementation of CComponentZ

#include "stdafx.h"
#include "ComponentZ.h"
#include ".\componentz.h"

// CComponentZ

STDMETHODIMP CComponentZ::MethodA(void) {
    // TODO: Add your implementation code here

    HRESULT hResult=MAKE_HRESULT( 1, FACILITY_NULL, 12 );

    MessageBox(NULL, "COM component", "Hello from", MB_OK);
```
return AtlReportError (GetObjectCLSID(), "My error message", 5, "http://error.asp", GUID_NULL, hResult);
}

The following code is managed code, in which the ATL component is called from a managed COM client. Because the HRESULT is unknown, the error code appears as a COMException exception.

using System;
using System.Runtime.InteropServices;

namespace COMClient {
    class Program {
        static void Main(string[] args) {
            try {
                COMLib.CComponentZClass com_object = new COMLib.CComponentZClass();
                com_object.MethodA();
            } catch (COMException except) {
                Console.WriteLine(except.ErrorCode);
                Console.WriteLine(except.HelpLink);
                Console.WriteLine(except.Message);
            } catch (Exception) {} 
        }
    }
}

Generating COM Exceptions

Managed components have an HRESULT property that translates a managed exception to a COM error result. System exceptions are already assigned an appropriate HRESULT. For application exceptions, you should initialize the HRESULT property in the constructor. It is important that managed components that expect COM clients set the HRESULT to a known COM error code.

TypeException is an application exception, which should be thrown when an object is the wrong type. TypeException has two overloaded constructors. Both constructors set the HResult property of the exception to the E_NOTIMPL error code (0x80004001). The one-argument constructor accepts a type object, which is the required type that was not implemented. The name of the type is added to the error message of the exception. This is the code for the TypeException class:

using System;

namespace Donis.CSharpBook {
    public class TypeException: Exception{

The `Delegator` class uses the `TypeException` class. `Delegator` delegates method calls of `Delegator.MethodA` to an external object. For the delegation to be successful, the external object must also implement the `MethodA` method, which is defined in the `ZInterface`. Appropriately, the code in `Delegator.MethodA` confirms that the external object implements the `ZInterface`. If not, the `TypeException` is thrown. Otherwise, `Delegator.MethodA` proceeds with the delegation:

```csharp
using System;

namespace Donis.CSharpBook{
    interface ZInterface {
        void MethodA();
    }

    public class Delegator {
        public Delegator(object obj) {
            externalobject = obj;
        }

        public void MethodA() {
            if (externalobject is ZInterface) {
                ((ZInterface)externalobject).MethodA();
            } else {
                throw new TypeException(
                    typeof(ZInterface));
            }
        }
    }
}
```
YClass creates an instance of the Delegator class. A ZClass object is passed into the Delegator constructor as the external object. ZClass does not implement MethodA. Delegator.MethodA is called in YClass.UseDelegator. A TypeException is thrown because the ZClass does not implement the appropriate interface:

```csharp
using System;
using System.Runtime.InteropServices;
using Donis.CSharpBook;

class ZClass {
}

[ClassInterface(ClassInterfaceType.AutoDual)]
public class YClass {
    public void UseDelegator() {
        ZClass obj = new ZClass();
        Delegator del = new Delegator(obj);
        del.MethodA();
    }
}
```

COM clients can access managed code through COM Callable Wrappers (CCWs). The following unmanaged code creates an instance of YClass and invokes YClass.UseDelegator. As expected, an exception occurs, which translates to E_NOTIMPL in unmanaged code. The COM client checks for this exception and displays the appropriate message.

```csharp
#import "..\yclass.tlb" no_namespace, raw_interfaces_only, named_guids
#include "objbase.h"

void main() {
    CoInitialize(NULL);
    _YClassPtr obj(CLSID_YClass);
    HRESULT hResult = obj->UseDelegator();
    if (hResult == E_NOTIMPL) {
        MessageBox(NULL, "Required interface not implemented", "In Managed Component", MB_OK);
    } else {
        MessageBox(NULL, "Managed Component", "No error", MB_OK);
    }
}
```

## Remote Exceptions

Exceptions sometime occur in remote code. An exception that is raised in a different application domain is a remote exception. Remote exceptions include exceptions thrown in a .NET Remoting application or a Web service application. Exceptions that cross application domains
must be serialized to maintain the state. System exceptions are serializable. However, you need to make application exceptions serializable.

Follow these steps to serialize an application exception:

1. Adorn the application exception with the `serializable` attribute.

2. Implement a two-argument constructor with a `SerializationInfo` and `StreamingContext` parameter. Deserialize the exception with the `SerializationInfo` parameter, which is a state bag. Retrieve state information of the exception using the `Get` methods of the `SerializationInfo` object. The `StreamingContext` parameter provides additional data about the source or target of the serialization process. In addition, call the same constructor in the base class to allow the base class to deserialize its state.

3. Implement the `GetObjectData` method to serialize the exception. The method also has two parameters: `Serialization` and `StreamingContext`. Use the `Serialization.AddValue` to serialize the state of the exception. Invoke `GetObjectData` on the base class to allow it to serialize itself.

4. For the exception to be available in the client assembly, share the assembly through a global assembly cache or an application configuration file. If the assembly is not shared, the assembly must be copied into the private directory of the client application.

`CustomException` is an application exception that supports remoting. There is one property, `prop_Time`, which is serialized in `GetObjectData` and deserialized in the two-argument constructor:

```csharp
using System;
using System.Reflection;
using System.Runtime.Serialization;

[assembly:AssemblyVersion("1.1.0.0")]
[assembly:AssemblyCultureAttribute('')]

namespace Donis.CSharpBook{
    [Serializable]
    public class CustomException : Exception{

        public CustomException()
            : base("custom exception", null) {
            prop_Time=DateTime.Now.ToLongDateString() + " + " +
            DateTime.Now.ToShortTimeString();
        }

        protected CustomException(SerializationInfo info,.streamingContext) : base(info, context){
            prop_Time=info.GetString("Time");
        }
    }
}
public override void GetObjectData( SerializationInfo info,
    StreamingContext context ){
    info.AddValue("Time", prop_Time, typeof(string));
    base.GetObjectData(info,context);
}

protected string prop_Time=null;
public string Time {
    get {
        return prop_Time;
    }
}
}

In Microsoft Visual Studio, the assembly attributes, such as AssemblyVersion, are found in the AssemblyInfo.cs file.

**Unhandled Exceptions**

Unhandled exceptions are not handled directly in application code; they are handled in a global handler. The global handler reports the exception in an error box that offers the Debug, Send Error Report, and Don’t Send buttons. For applications within the realm of a debugger, a potential unhandled exception is handled in the debugger.

What is the life cycle of an exception? Exceptions are initially categorized as first chance exceptions. If the application is attached to a debugger, the debugger is first consulted about the exception. Debuggers typically ignore first chance exceptions, and the exception is forwarded to the application next. When no debugger is present, the first chance exception is immediately sent to the application. If the application does not handle the first chance exception, the exception becomes a high-priority second chance exception. If a debugger is attached, the second chance exception is handled by the debugger. Upon finding a second chance exception, the Visual Studio debugger transfers the user to the location in the source code where the exception happened. If no debugger is present, execution is transferred to a global exception handler, which displays an error dialog box and then terminates the application. Figure 9-2 shows the life cycle of an exception.

Applications can trap unhandled exceptions. The mechanism is different for Windows Forms and for Console applications. For Windows Forms, add a handler to the Application.ThreadException event. For Console applications, the handler is added to the AppDomain.UnhandledException event. Methods added to the Application.ThreadException event chain catch and handle the exception. This is an advantage when compared with AppDomain.UnhandledException. Methods added to the AppDomain.UnhandledException event can respond to an unhandled exception, but the exception is not caught. Therefore, the exception will resurface after the handlers have completed.
Figure 9-2  Life cycle of an exception

Do not use the unhandled exception handler to catch all exceptions. Proper application design identifies specific exceptions that an application should anticipate. Those exceptions should be caught and handled within the confines of structured exception handling. Reserve the unhandled exception method for unanticipated exceptions.

**Application.ThreadException**

In a Windows Forms application, the windows procedure raises the `Application.ThreadException` event upon an unhandled exception. Subscribe to the `ThreadException` event to handle the unhandled exception. The subscriber is an exception handler, which prevents the application from being terminated. Do not propagate an exception caught in this manner in the unhandled exception handler. The new exception is unprotected and will likely terminate the application. After the unhandled exception handler completes, execution continues at the next message in the message pump.
Subscribe to the `ThreadException` event with a `ThreadExceptionEventHandler` delegate, which has two parameters. The object parameter is the thread object of the thread that raised the exception. The `ThreadExceptionEventArgs` parameter of the `System.Threading` namespace contains the exception that was unhandled. This is the signature of the `ThreadExceptionEventHandler`:

```
ThreadExceptionEventHandler syntax:

    void ThreadExceptionEventHandler(object sender,
                                           ThreadExceptionEventArgs e)
```

In the following code, the `OnThreadException` handler is added to the `ThreadException` event. The `btnException_Click` method raises an unhandled divide by zero exception. The unhandled exception is then handled in the `OnThreadException` method, which displays an informative message. Run the application in release mode for the expected results. Otherwise, the Visual Studio debugger intercedes the exception.

```csharp
private void btnException_Click(object sender, EventArgs e) {
    int varA = 5, varB = 0;
    varA /= varB;
}

private void Form1_Load(object sender, EventArgs e) {
    Application.ThreadException += new
    System.Threading.ThreadExceptionEventHandler(
    OnThreadException);
}

void OnThreadException(object sender, ThreadExceptionEventArgs e) {
    Thread t = (Thread) sender;
    Exception threadException = e.Exception;
    string errorMessage = "Thread ID: " +
    t.ManagedThreadId.ToString() +
    " [ " + threadException.Message + " ]";
    MessageBox.Show(errorMessage);
}
```

`AppDomain.UnhandledException`

When an unhandled exception is manifested in a Console application, the `AppDomain.UnhandledException` is raised. Subscribe to the event to clean up the resources of the application, such as closing files and relinquishing data connections. You might also record the unhandled exception in the event log or another location. It is important to note that the exception is not caught in the `AppDomain.UnhandledException` handler. After the handler finishes, the unhandled exception causes the application to be terminated. The `AppDomain.UnhandledException` event is triggered only in the initial application domain; it is ignored in other application domains.
Subscribe to the `AppDomain.UnhandledException` event with an `UnhandledExceptionEventHandler` delegate. The delegate has two parameters. The `object` parameter is the `AppDomain` object of the initial application domain. The `UnhandledExceptionEventArgs` parameter contains the specifics of the unhandled exception. This is the syntax of the `UnhandledExceptionEventHandler`:

**UnhandledExceptionEventHandler** syntax:

```csharp
void UnhandledExceptionEventHandler(object sender, UnhandledExceptionEventArgs e)
```

`UnhandledExceptionEventArgs` offers the `IsTerminating` and `ExceptionObject` properties. `IsTerminating` is a Boolean property indicating the status of the application. If true, the application is terminating because of the exception. If false, the application survives the exception. In .NET 2.0, this property is always true. Unhandled exceptions on both managed and unmanaged threads terminate an application. This is cleaner than the .NET 1.1 unhandled exception model, where exceptions raised on managed threads were nonfatal. The `Exception` property is an exception object for the unhandled exception. Inexplicably, this property is an object type, not an `Exception` type. Cast the property to the `Exception` type to access the details of the exception.

In the following `Console` application, the `OnUnhandledException` method is added to the `AppDomain.UnhandledException` event. When the subsequent divide by zero exception occurs, the `OnUnhandledException` method is called.

```csharp
using System;

namespace Donis.CSharpBook{
    public class Starter{
        public static void Main(){
            AppDomain.CurrentDomain.UnhandledException+=
                new UnhandledExceptionEventHandler(
                    OnUnhandledException);
            int vara = 5, varb = 0;
            vara /= varb;
        }
        public static void OnUnhandledException( object sender, UnhandledExceptionEventArgs e) {
            string application_name=sender.ToString();
            Exception except=(Exception) e.ExceptionObject;
            string errmsg = "Application " +application_name+ " [ Exception " + except.Message + " ]";
            Console.WriteLine(errmsg);
        }
    }
}
```
Managing Exceptions in Visual Studio

Visual Studio 2005 can configure exceptions for debugging. The Exception Assistant provides helpful information to developers when an unhandled exception is raised. The Exceptions dialog box alters how the Visual Studio debugger handles exceptions.

Exception Assistant

The Exception Assistant appears in Visual Studio when an unhandled exception occurs. It displays a translucent frame that partially obfuscates the application code. The source code that prompted the exception is highlighted and tethered to the Exception Assistant window. The Exception Assistant is shown in Figure 9-3.

![Figure 9-3 The Exception Assistant](image)

The Exception Assistant header identifies the unhandled exception and offers a brief explanation. The Troubleshooting Tips section offers hints to diagnose the exception. Each hint is also a link to further information. The Actions pane specifies two actions:

- New Detail displays the properties of the exception object.
- Copy Exceptional Detail To The Clipboard copies basic information on the exception onto the Clipboard.

The Exception Assistant can be disabled or otherwise configured from the Tools menu. Select the Options menu item. Configure the Exception Assistant in the Debugging pane.

Exceptions Dialog Box

In the Exceptions dialog box, the Visual Studio debugger can be instructed to interrupt on first chance exceptions. Open the Exceptions dialog box from the Debug menu. Choose the Exceptions menu item. Figure 9-4 shows the Exceptions dialog box.
Figure 9-4  The Exceptions dialog box

The Thrown and User-unhandled columns have a series of option boxes that are organized into categories and specific exceptions. Visual Studio debugger interrupts on first chance exceptions for exceptions selected in the Thrown column. For protected code, the debugger intercedes before the exception is caught. In the development phase of an application, it can be instructive to be notified of exceptions that would otherwise be consumed in an exception handler. The second column selects specific user-handled exceptions to break on. The Add button appends application exceptions to the list of available exceptions that are selectable in the dialog box. These exceptions can later be deleted.

**Metadata and Reflections**

.NET assemblies adhere to the Portable Executable Common Object File Format (PE COFF). PE COFF files have headers and sections, such as a PE and CLR Header. .NET assemblies consist of metadata and MSIL code. Metadata describes the types and other characteristics of the assembly. You can inspect the metadata of an assembly using reflection. Tools, such as ILDASM, browse an assembly using reflection. Reflection also provides support for late binding. The next chapter explores metadata and reflection.