CHAPTER 20

Exceptions and State Management

In this chapter:

This chapter is all about error handling. But it's not just about that. There are several parts to error handling. First, we'll define what an error actually is. Then, we'll talk about how to discover when your code is experiencing an error and about how to recover from this error. At this point, state becomes an issue because errors tend to come at inopportune times. It is likely that your code will be in the middle of mutating some state when it experiences the error, and your code likely will have to restore some state back to what it was prior to attempting to mutate it. Of course, we'll also talk about how your code can notify its callers that it has detected an error.

In my opinion, exception handling is the weakest area of the common language runtime (CLR) and therefore causes many problems for developers writing managed code. Over the years, Microsoft has made some significant improvements to help developers deal with errors, but I believe that there is much more that must be done before we can really have a good, reliable system. I will talk a lot about the various enhancements that have been made when dealing with unhandled exceptions, constrained execution regions, code contracts, runtime wrapped exceptions, uncatchable exceptions, and so on.

Defining "Exception"

When designing a type, you first imagine the various situations in which the type will be used. The type name is usually a noun, such as `FileStream` or `StringBuilder`. Then you define the properties, methods, events, and so on for the type. The way you define these members (property data types, method parameters, return values, and so forth) becomes the programmatic interface for your type. These members indicate actions that can be performed by the type itself or on an instance of the type. These action members are usually verbs such as `Read`, `Write`, `Flush`, `Append`, `Insert`, `Remove`, etc. When an action member cannot complete its task, the member should throw an exception.

Important An exception is when a member fails to complete the task it is supposed to perform as indicated by its name.

Look at the following class definition.

```
internal sealed class Account {
   public static void Transfer(Account from, Account to, Decimal amount) {
      from -= amount;
      to += amount;
   }
}
```

The `Transfer` method accepts two `Account` objects and a `Decimal` value that identifies an amount of money to transfer between accounts. Obviously, the goal of the `Transfer` method is to subtract money from one account and add money to another. The `Transfer` method could fail for many reasons: the `from` or `to` argument might be `null`; the `from` or `to` argument might not refer to an open account; the `from` account might have insufficient funds; the `to` account might have so much money in it that adding more would cause it to overflow; or the amount argument might be `0`, negative, or have more than two digits after the decimal place.

When the `Transfer` method is called, its code must check for all of these possibilities, and if any of them are detected, it cannot transfer the money and should notify the caller that it failed by throwing an exception. In fact, notice that the `Transfer` method's return type is `void`. This is because the `Transfer` method has no meaningful value to return; if it returns at all, it was successful. If it fails, it throws a meaningful exception.

Object-oriented programming allows developers to be very productive because you get to write code like this.

```
Boolean f = "Jeff".Substring(1, 1).ToUpper().EndsWith("E"); // true
```

Here I'm composing my intent by chaining several operations together.[1] This code was easy for me to write and is easy for others to read and maintain because the intent is obvious: take a string, grab

[1] In fact, C#'s extension method feature exists in the language to allow you to chain more methods together that would not have been chainable otherwise.

a portion of it, uppercase that portion, and see if it ends with an "E." This is great, but there is a big assumption being made here: no operation fails. But, of course, errors are always possible, so we need a way to handle those errors. In fact, there are many object-oriented constructs—constructors, getting/setting a property, adding/removing an event, calling an operator overload, calling a conversion operator—that have no way to return error codes, but these constructs must still be able to report an error. The mechanism provided by the Microsoft .NET Framework and all programming languages that support it is called *exception handling*.

Important Many developers incorrectly believe that an exception is related to how *frequently* something happens. For example, a developer designing a file Read method is likely to say the following: "When reading from a file, you will eventually reach the end of its data. Because reaching the end will *always* happen, I'll design my Read method so that it reports the end by returning a special value; I won't have it throw an exception." The problem with this statement is that it is being made by the developer designing the Read method, not by the developer calling the Read method.

When designing the Read method, it is impossible for the developer to know all of the possible situations in which the method gets called. Therefore, the developer can't possibly know how *often* the caller of the Read method will attempt to read past the end of the file. In fact, because most files contain structured data, attempting to read past the end of a file is something that *rarely* happens.

Exception-Handling Mechanics

In this section, I'll introduce the mechanics and C# constructs needed in order to use exception handling, but it's not my intention to explain them in great detail. The purpose of this chapter is to offer useful guidelines for when and how to use exception handling in your code. If you want more information about the mechanics and language constructs for using exception handling, see the .NET Framework documentation and the C# language specification. Also, the .NET Framework exception-handling mechanism is built using the Structured Exception Handling (SEH) mechanism offered by Windows. SEH has been discussed in many resources, including the book, *Windows via C/C++*, Fifth Edition, by myself and Christophe Nasarre (Microsoft Press, 2007), which contains three chapters devoted to SEH.

The following C# code shows a standard usage of the exception-handling mechanism. This code gives you an idea of what exception-handling blocks look like and what their purpose is. In the subsections after the code, I'll formally describe the `try`, `catch`, and `finally` blocks and their purpose and provide some notes about their use.

```
private void SomeMethod() {

   try {
      // Put code requiring graceful recovery and/or cleanup operations here...
   }
   catch (InvalidOperationException) {
      // Put code that recovers from an InvalidOperationException here...
   }
   catch (IOException) {
      // Put code that recovers from an IOException here...
   }
   catch {
      // Put code that recovers from any kind of exception other than those preceding this...

      // When catching any exception, you usually re-throw the exception.
      // I explain re-throwing later in this chapter.
      throw;
   }
   finally {
      // Put code that cleans up any operations started within the try block here...
      // The code in here ALWAYS executes, regardless of whether an exception is thrown.
   }
   // Code below the finally block executes if no exception is thrown within the try block
   // or if a catch block catches the exception and doesn't throw or re-throw an exception.
}
```

This code demonstrates one possible way to use exception-handling blocks. Don't let the code scare you—most methods have simply a `try` block matched with a single `finally` block or a `try` block matched with a single `catch` block. It's unusual to have as many `catch` blocks as in this example. I put them there for illustration purposes.

The `try` Block

A *try block* contains code that requires common cleanup operations, exception-recovery operations, or both. The cleanup code should be placed in a single `finally` block. A `try` block can also contain code that might potentially throw an exception. The exception-recovery code should be placed in one or more `catch` blocks. You create one `catch` block for each kind of exception that your application can safely recover from. A `try` block must be associated with at least one `catch` or `finally` block; it makes no sense to have a `try` block that stands by itself, and C# will prevent you from doing this.

Important Sometimes developers ask how much code they should put inside a single `try` block. The answer to this depends on state management. If, inside a `try` block, you execute multiple operations that could all throw the same exception type and the way that you'd recover this exception type is different depending on the operation, then you should put each operation in its own `try` block so that you can recover your state correctly.

The `catch` Block

A `catch` *block* contains code to execute in response to an exception. A `try` block can have zero or more `catch` blocks associated with it. If the code in a `try` block doesn't cause an exception to be thrown, the CLR will never execute the code contained within any of its `catch` blocks. The thread will simply skip over all of the `catch` blocks and execute the code in the `finally` block (if one exists). After the code in the `finally` block executes, execution continues with the statement following the `finally` block.

The parenthetical expression appearing after the `catch` keyword is called the *catch type*. In C#, you must specify a catch type of `System.Exception` or a type derived from `System.Exception`. For example, the previous code contains `catch` blocks for handling an `InvalidOperationException` (or any exception derived from it) and an `IOException` (or any exception derived from it). The last `catch` block (which doesn't specify a catch type) handles any exception except for the exception type specified by earlier `catch` blocks; this is equivalent to having a `catch` block that specifies a catch type of `System.Exception` except that you cannot access the exception information via code inside the `catch` block's braces.

Note When debugging through a `catch` block by using Microsoft Visual Studio, you can see the currently thrown exception object by adding the special *$exception* variable name to a watch window.

The CLR searches from top to bottom for a matching `catch` type, and therefore you should place the more specific exception types at the top. The most-derived exception types should appear first, followed by their base types (if any), down to `System.Exception` (or an exception block that doesn't specify a catch type). In fact, the C# compiler generates an error if more specific `catch` blocks appear closer to the bottom because the `catch` block would be unreachable.

If an exception is thrown by code executing within the `try` block (or any method called from within the `try` block), the CLR starts searching for `catch` blocks whose catch type is the same type as or a base type of the thrown exception. If none of the catch types matches the exception, the CLR continues searching up the call stack looking for a catch type that matches the exception. If after reaching the top of the call stack, no `catch` block is found with a matching catch type, an unhandled exception occurs. I'll talk more about unhandled exceptions later in this chapter.

After the CLR locates a `catch` block with a matching catch type, it executes the code in all inner `finally` blocks, starting from within the `try` block whose code threw the exception and stopping with the `catch` block that matched the exception. Note that any `finally` block associated with the `catch` block that matched the exception is not executed yet. The code in this `finally` block won't execute until after the code in the handling `catch` block has executed.

After all the code in the inner `finally` blocks has executed, the code in the handling `catch` block executes. This code typically performs some operations to deal with the exception. At the end of the `catch` block, you have three choices:

- Re-throw the same exception, notifying code higher up in the call stack of the exception.
- Throw a different exception, giving richer exception information to code higher up in the call stack.
- Let the thread fall out of the bottom of the `catch` block.

Later in this chapter, I'll offer some guidelines for when you should use each of these techniques. If you choose either of the first two techniques, you're throwing an exception, and the CLR behaves just as it did before: it walks up the call stack looking for a `catch` block whose type matches the type of the exception thrown.

If you pick the last technique, when the thread falls out of the bottom of the `catch` block, it immediately starts executing code contained in the `finally` block (if one exists). After all of the code in the `finally` block executes, the thread drops out of the `finally` block and starts executing the statements immediately following the `finally` block. If no `finally` block exists, the thread continues execution at the statement following the last `catch` block.

In C#, you can specify a variable name after a catch type. When an exception is caught, this variable refers to the `System.Exception`-derived object that was thrown. The `catch` block's code can reference this variable to access information specific to the exception (such as the stack trace leading up to the exception). Although it's possible to modify this object, you shouldn't; consider the object to be read-only. I'll explain the `Exception` type and what you can do with it later in this chapter.

Note Your code can register with `AppDomain`'s `FirstChanceException` event to receive notifications as soon as an exception occurs within an AppDomain. This notification occurs before the CLR searches for any `catch` blocks. For more information about this event, see Chapter 22, "CLR Hosting and AppDomains."

The `finally` Block

A `finally` block contains code that's guaranteed to execute.[2] Typically, the code in a `finally` block performs the cleanup operations required by actions taken in the `try` block. For example, if you open a file in a `try` block, you'd put the code to close the file in a `finally` block.

```
private void ReadData(String pathname) {

   FileStream fs = null;
   try {
      fs = new FileStream(pathname, FileMode.Open);
```

[2] Aborting a thread or unloading an AppDomain causes the CLR to throw a `ThreadAbortException`, which allows the `finally` block to execute. If a thread is simply killed via the Win32 `TerminateThread` function, or if the process is killed via the Win32 `TerminateProcess` function or `System.Environment`'s `FailFast` method, then the `finally` block will not execute. Of course Windows cleans up all resources that a process was using when a process terminates.

```
      // Process the data in the file...
   }
   catch (IOException) {
      // Put code that recovers from an IOException here...
   }
   finally {
      // Make sure that the file gets closed.
      if (fs != null) fs.Close();
   }
}
```

If the code in the `try` block executes without throwing an exception, the file is guaranteed to be closed. If the code in the `try` block does throw an exception, the code in the `finally` block still executes, and the file is guaranteed to be closed, regardless of whether the exception is caught. It's improper to put the statement to close the file after the `finally` block; the statement wouldn't execute if an exception were thrown and not caught, which would result in the file being left open (until the next garbage collection).

A `try` block doesn't require a `finally` block associated with it; sometimes the code in a `try` block just doesn't require any cleanup code. However, if you do have a `finally` block, it must appear after any and all `catch` blocks. A `try` block can have no more than one `finally` block associated with it.

When a thread reaches the end of the code contained in a `finally` block, the thread simply starts executing the statements immediately following the `finally` block. Remember that the code in the `finally` block is cleanup code. This code should execute only what is necessary to clean up operations initiated in the `try` block. The code inside `catch` and `finally` blocks should be very short and should have a high likelihood of succeeding without itself throwing an exception. Usually the code in these blocks is just one or two lines of code.

It is always possible that exception-recovery code or cleanup code could fail and throw an exception. Although possible, it is unlikely and if it does happen it usually means that there is something very wrong somewhere. Most likely some state has gotten corrupted somewhere. If an exception is inadvertently thrown within a `catch` or `finally` block, the world will not come to an end—the CLR's exception mechanism will execute as though the exception were thrown after the `finally` block. However, the CLR does not keep track of the first exception that was thrown in the corresponding `try` block (if any), and you will lose any and all information (such as the stack trace) available about the first exception. Probably (and hopefully), this new exception will not be handled by your code and the exception will turn into an unhandled exception. The CLR will then terminate your process, which is good because all the corrupted state will now be destroyed. This is much better than having your application continue to run with unpredictable results and possible security holes.

Personally, I think the C# team should have chosen different language keywords for the exception-handling mechanism. What programmers want to do is try to execute some piece of code. And then, if something fails, either recover from the failure and move on or compensate to undo some state change and continue to report the failure up to a caller. Programmers also want to have guaranteed cleanup no matter what happens.

The code on the left is what you have to write to make the C# compiler happy, but the code on the right is the way I prefer to think about it.

```
void Method() {                      void Method() {
   try {                                try {
      ...                                  ...
   }                                    }
   catch (XxxException) {               handle (XxxException) {
      ...                                  ...
   }                                    }
   catch (YyyException) {               handle (YyyException) {
      ...                                  ...
   }                                    }
   catch {                              compensate {
      ...; throw;                          ...
   }                                    }
   finally {                            cleanup {
      ...                                  ...
   }                                    }
}                                    }
```

CLS and Non-CLS Exceptions

All programming languages for the CLR must support the throwing of `Exception`-derived objects because the Common Language Specification (CLS) mandates this. However, the CLR actually allows an instance of any type to be thrown, and some programming languages will allow code to throw non–CLS-compliant exception objects such as a `String`, `Int32`, or `DateTime`. The C# compiler allows code to throw only `Exception`-derived objects, whereas code written in some other languages allow code to throw `Exception`-derived objects in addition to objects that are not derived from `Exception`.

Many programmers are not aware that the CLR allows any object to be thrown to report an exception. Most developers believe that only `Exception`-derived objects can be thrown. Prior to CLR 2.0, when programmers wrote `catch` blocks to catch exceptions, they were catching CLS-compliant exceptions only. If a C# method called a method written in another language, and that method threw a non–CLS-compliant exception, the C# code would not catch this exception at all, leading to some security vulnerabilities.

In CLR 2.0, Microsoft introduced a new `RuntimeWrappedException` class (defined in the `System.Runtime.CompilerServices` namespace). This class is derived from `Exception`, so it is a CLS-compliant exception type. The `RuntimeWrappedException` class contains a private field of type `Object` (which can be accessed by using `RuntimeWrappedException`'s `WrappedException` read-only property). In CLR 2.0, when a non–CLS-compliant exception is thrown, the CLR automatically constructs an instance of the `RuntimeWrappedException` class and initializes its private field to refer to the object that was actually thrown. In effect, the CLR now turns all non–CLS-compliant exceptions into CLS-compliant exceptions. Any code that now catches an `Exception` type will catch non–CLS-compliant exceptions, which fixes the potential security vulnerability problem.

Although the C# compiler allows developers to throw `Exception`-derived objects only, prior to C# 2.0, the C# compiler did allow developers to catch non–CLS-compliant exceptions by using code like this.

```
private void SomeMethod() {
   try {
      // Put code requiring graceful recovery and/or cleanup operations here...
   }
   catch (Exception e) {
      // Before C# 2.0, this block catches CLS-compliant exceptions only
      // Now, this block catches CLS-compliant & non–CLS-compliant exceptions
      throw; // Re-throws whatever got caught
   }
   catch {
      // In all versions of C#, this block catches CLS-compliant
      // & non–CLS-compliant exceptions
      throw; // Re-throws whatever got caught
   }
}
```

Now, some developers were aware that the CLR supports both CLS-compliant and non–CLS-compliant exceptions, and these developers might have written the preceding two `catch` blocks in order to catch both kinds of exceptions. If the preceding code is recompiled for CLR 2.0 or later, the second `catch` block will never execute, and the C# compiler will indicate this by issuing a warning: `CS1058: A previous catch clause already catches all exceptions. All non-exceptions thrown will be wrapped in a System.Runtime.CompilerServices.RuntimeWrappedException.`

There are two ways for developers to migrate code from a version of the .NET Framework prior to version 2.0. First, you can merge the code from the two `catch` blocks into a single `catch` block and delete one of the `catch` blocks. This is the recommended approach. Alternatively, you can tell the CLR that the code in your assembly wants to play by the old rules. That is, tell the CLR that your `catch (Exception)` blocks should not catch an instance of the new `RuntimeWrappedException` class. And instead, the CLR should unwrap the non–CLS-compliant object and call your code only if you have a `catch` block that doesn't specify any type at all. You tell the CLR that you want the old behavior by applying an instance of the `RuntimeCompatibilityAttribute` to your assembly like this.

```
using System.Runtime.CompilerServices;
[assembly:RuntimeCompatibility(WrapNonExceptionThrows = false)]
```

Note This attribute has an assembly-wide impact. There is no way to mix wrapped and unwrapped exception styles in the same assembly. Be careful when adding new code (that expects the CLR to wrap exceptions) to an assembly containing old code (in which the CLR didn't wrap exceptions).

The System.Exception Class

The CLR allows an instance of any type to be thrown for an exception—from an Int32 to a String and beyond. However, Microsoft decided against forcing all programming languages to throw and catch exceptions of any type, so they defined the System.Exception type and decreed that all CLS-compliant programming languages must be able to throw and catch exceptions whose type is derived from this type. Exception types that are derived from System.Exception are said to be CLS-compliant. C# and many other language compilers allow your code to throw only CLS-compliant exceptions.

The System.Exception type is a very simple type that contains the properties described in Table 20-1. Usually, you will not write any code to query or access these properties in any way. Instead, when your application terminates due to an unhandled exception, you will look at these properties in the debugger or in a report that gets generated and written out to the Windows Application event log or crash dump.

TABLE 20-1 Public Properties of the System.Exception Type

Property	Access	Type	Description
Message	Read-only	String	Contains helpful text indicating why the exception was thrown. The message is typically written to a log when a thrown exception is unhandled. Because end users do not see this message, the message should be as technical as possible so that developers viewing the log can use the information in the message to fix the code when producing a new version.
Data	Read-only	IDictionary	A reference to a collection of key-value pairs. Usually, the code throwing the exception adds entries to this collection prior to throwing it; code that catches the exception can query the entries and use the information in its exception-recovery processing.
Source	Read/write	String	Contains the name of the assembly that generated the exception.
StackTrace	Read-only	String	Contains the names and signatures of methods called that led up to the exception being thrown. This property is invaluable for debugging.
TargetSite	Read-only	MethodBase	Contains the method that threw the exception.
HelpLink	Read-only	String	Contains a URL (such as file://C:\MyApp\Help.htm #MyExceptionHelp) to documentation that can help a user understand the exception. Keep in mind that sound programming and security practices prevent users from ever being able to see raw unhandled exceptions, so unless you are trying to convey information to other programmers, this property is seldom used.

Property	Access	Type	Description
`InnerException`	Read-only	`Exception`	Indicates the previous exception if the current exception were raised while handling an exception. This read-only property is usually `null`. The `Exception` type also offers a public `GetBaseException` method that traverses the linked list of inner exceptions and returns the originally thrown exception.
`HResult`	Read/write	`Int32`	A 32-bit value that is used when crossing managed and native code boundaries. For example, when COM APIs return failure HRESULT values, the CLR throws an `Exception`-derived object and maintains the HRESULT value in this property.

I'd like to say a few words about `System.Exception`'s read-only `StackTrace` property. A `catch` block can read this property to obtain the stack trace indicating what methods were called that led up to the exception. This information can be extremely valuable when you're trying to detect the cause of an exception so that you can correct your code. When you access this property, you're actually calling into code in the CLR; the property doesn't simply return a string. When you construct a new object of an `Exception`-derived type, the `StackTrace` property is initialized to `null`. If you were to read the property, you wouldn't get back a stack trace; you would get back `null`.

When an exception is thrown, the CLR internally records where the `throw` instruction occurred. When a `catch` block accepts the exception, the CLR records where the exception was caught. If, inside a `catch` block, you now access the thrown exception object's `StackTrace` property, the code that implements the property calls into the CLR, which builds a string identifying all of the methods between the place where the exception was thrown and the filter that caught the exception.

Important When you throw an exception, the CLR resets the starting point for the exception; that is, the CLR remembers only the location where the most recent exception object was thrown.

The following code throws the same exception object that it caught and causes the CLR to reset its starting point for the exception.

```
private void SomeMethod() {
   try { ... }
   catch (Exception e) {
      ...
      throw e;  // CLR thinks this is where exception originated.
                // FxCop reports this as an error
   }
}
```

In contrast, if you re-throw an exception object by using the `throw` keyword by itself, the CLR doesn't reset the stack's starting point. The following code re-throws the same exception object that it caught, causing the CLR to not reset its starting point for the exception.

```
private void SomeMethod() {
   try { ... }
   catch (Exception e) {
      ...
      throw;  // This has no effect on where the CLR thinks the exception
              // originated. FxCop does NOT report this as an error
   }
}
```

In fact, the only difference between these two code fragments is what the CLR thinks is the original location where the exception was thrown. Unfortunately, when you throw or re-throw an exception, Windows does reset the stack's starting point. So if the exception becomes unhandled, the stack location that gets reported to Windows Error Reporting is the location of the last throw or re-throw, even though the CLR knows the stack location where the original exception was thrown. This is unfortunate because it makes debugging applications that have failed in the field much more difficult. Some developers have found this so intolerable that they have chosen a different way to implement their code to ensure that the stack trace truly reflects the location where an exception was originally thrown.

```
private void SomeMethod() {
   Boolean trySucceeds = false;
   try {
      ...
      trySucceeds = true;
   }
   finally {
      if (!trySucceeds) { /* catch code goes in here */ }
   }
}
```

The string returned from the `StackTrace` property doesn't include any of the methods in the call stack that are above the point where the `catch` block accepted the exception object. If you want the complete stack trace from the start of the thread up to the exception handler, you can use the `System.Diagnostics.StackTrace` type. This type defines some properties and methods that allow a developer to programmatically manipulate a stack trace and the frames that make up the stack trace.

You can construct a `StackTrace` object by using several different constructors. Some constructors build the frames from the start of the thread to the point where the `StackTrace` object is constructed. Other constructors initialize the frames of the `StackTrace` object by using an `Exception`-derived object passed as an argument.

If the CLR can find debug symbols (located in the .pdb files) for your assemblies, the string returned by `System.Exception`'s `StackTrace` property or `System.Diagnostics.StackTrace`'s `ToString` method will include source code file paths and line numbers. This information is incredibly useful for debugging.

Whenever you obtain a stack trace, you might find that some methods in the actual call stack don't appear in the stack trace string. There are two reasons for this. First, the stack is really a record of where the thread should return to, not where the thread has come from. Second, the just-in-time (JIT) compiler can inline methods to avoid the overhead of calling and returning from a separate method. Many compilers (including the C# compiler) offer a /debug command-line switch. When this switch is used, these compilers embed information into the resulting assembly to tell the JIT compiler not to inline any of the assembly's methods, making stack traces more complete and meaningful to the developer debugging the code.

Note The JIT compiler examines the System.Diagnostics.DebuggableAttribute custom attribute applied to the assembly. The C# compiler applies this attribute automatically. If this attribute has the DisableOptimizations flag specified, the JIT compiler won't inline the assembly's methods. Using the C# compiler's /debug switch sets this flag. By applying the System.Runtime.CompilerServices.MethodImplAttribute custom attribute to a method, you can forbid the JIT compiler from inlining the method for both debug and release builds. The following method definition shows how to forbid the method from being inlined.

```
using System;
using System.Runtime.CompilerServices;

internal sealed class SomeType {

   [MethodImpl(MethodImplOptions.NoInlining)]
   public void SomeMethod() {
      ...
   }
}
```

FCL-Defined Exception Classes

The Framework Class Library (FCL) defines many exception types (all ultimately derived from System.Exception). The following hierarchy shows the exception types defined in the MSCorLib.dll assembly; other assemblies define even more exception types. (The application used to obtain this hierarchy is shown in Chapter 23, "Assembly Loading and Reflection.")

```
System.Exception
   System.AggregateException
   System.ApplicationException
      System.Reflection.InvalidFilterCriteriaException
      System.Reflection.TargetException
      System.Reflection.TargetInvocationException
      System.Reflection.TargetParameterCountException
      System.Threading.WaitHandleCannotBeOpenedException
   System.Diagnostics.Tracing.EventSourceException
   System.InvalidTimeZoneException
   System.IO.IsolatedStorage.IsolatedStorageException
   System.Threading.LockRecursionException
   System.Runtime.CompilerServices.RuntimeWrappedException
   System.SystemException
      System.Threading.AbandonedMutexException
      System.AccessViolationException
      System.Reflection.AmbiguousMatchException
      System.AppDomainUnloadedException
      System.ArgumentException
         System.ArgumentNullException
         System.ArgumentOutOfRangeException
         System.Globalization.CultureNotFoundException
         System.Text.DecoderFallbackException
         System.DuplicateWaitObjectException
         System.Text.EncoderFallbackException
      System.ArithmeticException
         System.DivideByZeroException
         System.NotFiniteNumberException
         System.OverflowException
      System.ArrayTypeMismatchException
      System.BadImageFormatException
      System.CannotUnloadAppDomainException
      System.ContextMarshalException
      System.Security.Cryptography.CryptographicException
         System.Security.Cryptography.CryptographicUnexpectedOperationException
      System.DataMisalignedException
      System.ExecutionEngineException
      System.Runtime.InteropServices.ExternalException
         System.Runtime.InteropServices.COMException
         System.Runtime.InteropServices.SEHException
      System.FormatException
         System.Reflection.CustomAttributeFormatException
      System.Security.HostProtectionException
      System.Security.Principal.IdentityNotMappedException
      System.IndexOutOfRangeException
      System.InsufficientExecutionStackException
      System.InvalidCastException
      System.Runtime.InteropServices.InvalidComObjectException
      System.Runtime.InteropServices.InvalidOleVariantTypeException
      System.InvalidOperationException
         System.ObjectDisposedException
      System.InvalidProgramException
      System.IO.IOException
         System.IO.DirectoryNotFoundException
         System.IO.DriveNotFoundException
         System.IO.EndOfStreamException
         System.IO.FileLoadException
         System.IO.FileNotFoundException
         System.IO.PathTooLongException
      System.Collections.Generic.KeyNotFoundException
      System.Runtime.InteropServices.MarshalDirectiveException
      System.MemberAccessException
         System.FieldAccessException
         System.MethodAccessException
         System.MissingMemberException
```

```
            System.MissingFieldException
            System.MissingMethodException
      System.Resources.MissingManifestResourceException
      System.Resources.MissingSatelliteAssemblyException
      System.MulticastNotSupportedException
      System.NotImplementedException
      System.NotSupportedException
         System.PlatformNotSupportedException
      System.NullReferenceException
      System.OperationCanceledException
         System.Threading.Tasks.TaskCanceledException
      System.OutOfMemoryException
         System.InsufficientMemoryException
      System.Security.Policy.PolicyException
      System.RankException
      System.Reflection.ReflectionTypeLoadException
      System.Runtime.Remoting.RemotingException
         System.Runtime.Remoting.RemotingTimeoutException
      System.Runtime.InteropServices.SafeArrayRankMismatchException
      System.Runtime.InteropServices.SafeArrayTypeMismatchException
      System.Security.SecurityException
      System.Threading.SemaphoreFullException
      System.Runtime.Serialization.SerializationException
      System.Runtime.Remoting.ServerException
      System.StackOverflowException
      System.Threading.SynchronizationLockException
      System.Threading.ThreadAbortException
      System.Threading.ThreadInterruptedException
      System.Threading.ThreadStartException
      System.Threading.ThreadStateException
      System.TimeoutException
      System.TypeInitializationException
      System.TypeLoadException
         System.DllNotFoundException
         System.EntryPointNotFoundException
         System.TypeAccessException
      System.TypeUnloadedException
      System.UnauthorizedAccessException
         System.Security.AccessControl.PrivilegeNotHeldException
      System.Security.VerificationException
      System.Security.XmlSyntaxException
   System.Threading.Tasks.TaskSchedulerException
   System.TimeZoneNotFoundException
```

The original idea was that `System.Exception` would be the base type for all exceptions and that two other types, `System.SystemException` and `System.ApplicationException`, would be the only two types immediately derived from `Exception`. Furthermore, exceptions thrown by the CLR would be derived from `SystemException`, and all application-thrown exceptions would be derived from `ApplicationException`. This way, developers could write a `catch` block that catches all CLR-thrown exceptions or all application-thrown exceptions.

However, as you can see, this rule was not followed very well; some exception types are immediately derived from `Exception` (`IsolatedStorageException`), some CLR-thrown exceptions are derived from `ApplicationException` (`TargetInvocationException`), and some application-thrown exceptions are derived from `SystemException` (`FormatException`). So it is all a big mess, and the result is that the `SystemException` and `ApplicationException` types have no special meaning at all. At this point, Microsoft would like to remove them from the exception class hierarchy, but they can't because it would break any code that already references these two types.

Throwing an Exception

When implementing your own methods, you should throw an exception when the method cannot complete its task as indicated by its name. When you want to throw an exception, there are two issues that you really need to think about and consider.

The first issue is about which `Exception`-derived type you are going to throw. You really want to select a type that is meaningful here. Consider the code that is higher up the call stack and how that code might want to determine that a method failed in order to execute some graceful recovery code. You can use a type that is already defined in the FCL, but there may not be one in the FCL that matches your exact semantics. So you'll probably need to define your own type, ultimately derived from `System.Exception`.

If you want to define an exception type hierarchy, it is highly recommended that the hierarchy be shallow and wide in order to create as few base classes as possible. The reason is that base classes act as a way of treating lots of errors as one error, and this is usually dangerous. Along these lines, you should never throw a `System.Exception` object, and you should use extreme caution if you throw any other base class exception type.[3]

Important There are versioning ramifications here, too. If you define a new exception type derived from an existing exception type, then all code that catches the existing base type will now catch your new type as well. In some scenarios, this may be desired and in some scenarios, it may not be desired. The problem is that it really depends on how code that catches the base class responds to the exception type and types derived from it. Code that never anticipated the new exception may now behave unpredictably and open security holes. The person defining the new exception type can't know about all the places where the base exception is caught and how it is handled. And so, in practice, it is impossible to make a good intelligent decision here.

The second issue is about deciding what string message you are going to pass to the exception type's constructor. When you throw an exception, you should include a string message with detailed information indicating why the method couldn't complete its task. If the exception is caught and handled, this string message is not seen. However, if the exception becomes an unhandled exception, this message is usually logged. An unhandled exception indicates a true bug in the application, and a developer must get involved to fix the bug. An end user will not have the source code or the ability

[3] In fact, the `System.Exception` class should have been marked as `abstract`, which would forbid code that tried to throw it from even compiling.

to fix the code and recompile it. In fact, this string message should not be shown to an end user. So these string messages can be very technically detailed and as geeky as is necessary to help developers fix their code.

Furthermore, because all developers have to speak English (at least to some degree, because programming languages and the FCL classes and methods are in English), there is usually no need to localize exception string messages. However, you may want to localize the strings if you are building a class library that will be used by developers who speak different languages. Microsoft localizes the exception messages thrown by the FCL, because developers all over the world will be using this class library.

Defining Your Own Exception Class

Unfortunately, designing your own exception is tedious and error prone. The main reason for this is because all `Exception`-derived types should be serializable so that they can cross an AppDomain boundary or be written to a log or database. There are many issues related to serialization and they are discussed in Chapter 24, "Runtime Serialization." So, in an effort to simplify things, I made my own generic `Exception<TExceptionArgs>` class, which is defined as follows.

```
[Serializable]
public sealed class Exception<TExceptionArgs> : Exception, ISerializable
   where TExceptionArgs : ExceptionArgs {

   private const String c_args = "Args";  // For (de)serialization
   private readonly TExceptionArgs m_args;

   public  TExceptionArgs Args { get { return m_args; } }

   public Exception(String message = null, Exception innerException = null)
      : this(null, message, innerException) { }

   public Exception(TExceptionArgs args, String message = null,
      Exception innerException = null): base(message, innerException) { m_args = args; }

   // This constructor is for deserialization; since the class is sealed, the constructor is
   // private. If this class were not sealed, this constructor should be protected
   [SecurityPermission(SecurityAction.LinkDemand,
      Flags=SecurityPermissionFlag.SerializationFormatter)]
   private Exception(SerializationInfo info, StreamingContext context)
      : base(info, context) {
      m_args = (TExceptionArgs)info.GetValue(c_args, typeof(TExceptionArgs));
   }

   // This method is for serialization; it's public because of the ISerializable interface
   [SecurityPermission(SecurityAction.LinkDemand,
      Flags=SecurityPermissionFlag.SerializationFormatter)]
   public override void GetObjectData(SerializationInfo info, StreamingContext context) {
      info.AddValue(c_args, m_args);
      base.GetObjectData(info, context);
   }
   public override String Message {
      get {
         String baseMsg = base.Message;
         return (m_args == null) ? baseMsg : baseMsg + " (" + m_args.Message + ")";
      }
   }

   public override Boolean Equals(Object obj) {
      Exception<TExceptionArgs> other = obj as Exception<TExceptionArgs>;
      if (other == null) return false;
      return Object.Equals(m_args, other.m_args) && base.Equals(obj);
   }
   public override int GetHashCode() { return base.GetHashCode(); }
}
```

And the `ExceptionArgs` base class that `TExceptionArgs` is constrained to is very simple and looks like this.

```
[Serializable]
public abstract class ExceptionArgs {
   public virtual String Message { get { return String.Empty; } }
}
```

Now, with these two classes defined, I can trivially define more exception classes when I need to. To define an exception type indicating the disk is full, I simply do the following.

```
[Serializable]
public sealed class DiskFullExceptionArgs : ExceptionArgs {
   private readonly String m_diskpath; // private field set at construction time

   public DiskFullExceptionArgs(String diskpath) { m_diskpath = diskpath; }

   // Public read-only property that returns the field
   public String DiskPath { get { return m_diskpath; } }

   // Override the Message property to include our field (if set)
   public override String Message {
      get {
         return (m_diskpath == null) ? base.Message : "DiskPath=" + m_diskpath;
      }
   }
}
```

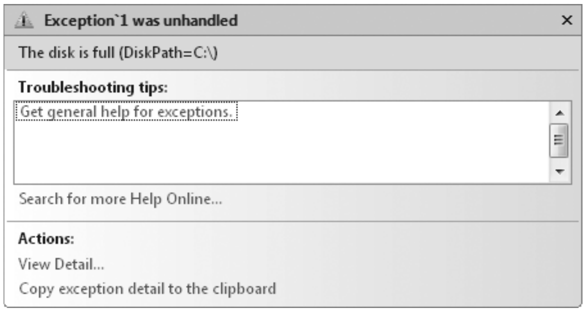
And, if I have no additional data that I want to put inside the class, it gets as simple as the following.

```
[Serializable]
public sealed class DiskFullExceptionArgs : ExceptionArgs { }
```

And now I can write code like this, which throws and catches one of these.

```
public static void TestException() {
   try {
      throw new Exception<DiskFullExceptionArgs>(
         new DiskFullExceptionArgs(@"C:\"), "The disk is full");
   }
   catch (Exception<DiskFullExceptionArgs> e) {
      Console.WriteLine(e.Message);
   }
}
```

Note There are two issues to note about my `Exception<TExceptionArgs>` class. The first issue is that any exception type you define with it is always derived from `System.Exception`. In most scenarios, this is not a problem at all and, in fact, having a shallow and wide exception type hierarchy is preferred. The second issue is that Visual Studio's unhandled exception dialog box doesn't display `Exception<T>` type's generic type parameter, as you can see in the following graphic.


Exception`1 was unhandled

The disk is full (DiskPath=C:\)

Troubleshooting tips:
Get general help for exceptions.

Search for more Help Online...

Actions:
View Detail...
Copy exception detail to the clipboard

Trading Reliability for Productivity

I started writing software in 1975. I did a fair amount of BASIC programming, and as I got more interested in hardware, I switched to assembly language. Over time, I switched to the C programming language because it allowed me access to hardware with a much higher level of abstraction, making my programming easier. My background is in writing operating systems' code and platform/library code, so I always work hard to make my code as small and as fast as possible, because applications can only be as good as the operating system and libraries they consume.

In addition to creating small and fast code, I always focused on error recovery. When allocating memory (by using C++'s `new` operator or by calling `malloc`, `HeapAlloc`, `VirtualAlloc`, etc.), I would always check the return value to ensure that the memory I requested was actually given to me. And, if the memory request failed, I always had an alternate code path ensuring that the rest of the program's state was unaffected and would let any of my callers know that I failed so that the calling code can take corrective measures too.

For some reason that I can't quite explain, this attention to detail is not done when writing code for the .NET Framework. Getting an out-of-memory situation is always possible and yet I almost never see any code containing a `catch` block to recover from an `OutOfMemoryException`. In fact, I've even had some developers tell me that the CLR doesn't let a program catch an `OutOfMemory-Exception`. For the record, this is absolutely not true; you can catch this exception. In fact, there are many errors that are possible when executing managed code and I hardly ever see developers write code that attempts to recover from these potential failures. In this section, I'd like to point out some of the potential failures and why it has become culturally acceptable to ignore them. I'd also like to point out some of the significant problems that can occur when ignoring these failures and suggest some ways to help mitigate these problems.

Object-oriented programming allows developers to be very productive. A big part of this is composability which makes it easy to write, read and maintain code. Take this line of code, for example.

```
Boolean f = "Jeff".Substring(1, 1).ToUpper().EndsWith("E");
```

There is a big assumption being made with the preceding code: no errors occur. But, of course, errors are always possible, and so we need a way to handle those errors. This is what the exception handling constructs and mechanisms are all about and why we need them as opposed to having methods that return `true/false` or an HRESULT to indicate success/failure the way that Win32 and COM functions do.

In addition to code composability, we are productive due to all kinds of great features provided by our compilers. For example, the compiler implicitly:

- Inserts optional parameters when calling a method.
- Boxes value type instances.
- Constructs/initializes parameter arrays.
- Binds to members of `dynamic` variables and expressions.
- Binds to extension methods.
- Binds/invokes overloaded operators.
- Constructs delegate objects.
- Infers types when calling generic methods, declaring a local variable, and using a lambda expression.
- Defines/constructs closure classes for lambda expressions and iterators.
- Defines/constructs/initializes anonymous types and instances of them.
- Rewrites code to support Language Integrated Queries (LINQs; query expressions and expression trees).

And, the CLR itself does all kinds of great things for developers to make our lives even easier. For example, the CLR implicitly:

- Invokes virtual methods and interface methods.
- Loads assemblies and JIT-compiles methods that can potentially throw `FileLoadException`, `BadImageFormatException`, `InvalidProgramException`, `FieldAccessException`, `MethodAccessException`, `MissingFieldException`, `MissingMethodException`, and `VerificationException`.
- Transitions across AppDomain boundaries when accessing an object of a `MarshalByRefObject`-derived type which can potentially throw `AppDomainUnloadedException`.
- Serializes and deserializes objects when crossing an AppDomain boundary.
- Causes thread(s) to throw a `ThreadAbortException` when `Thread.Abort` or `AppDomain.Unload` is called.
- Invokes `Finalize` methods after a garbage collection before objects have their memory reclaimed.
- Creates type objects in the loader heap when using generic types.
- Invokes a type's static constructor potential throwing of `TypeInitializationException`.
- Throws various exceptions, including `OutOfMemoryException`, `DivideByZeroException`, `NullReferenceException`, `RuntimeWrappedException`, `TargetInvocationException`, `OverflowException`, `NotFiniteNumberException`, `ArrayTypeMismatchException`, `DataMisalignedException`, `IndexOutOfRangeException`, `InvalidCastException`, `RankException`, `SecurityException`, and more.

And, of course, the .NET Framework ships with a massive class library that contains tens of thousands of types, each type encapsulating common, reusable functionality. There are types for building web form applications, web services, rich GUI applications, working with security, manipulation of images, speech recognition, and the list goes on and on. Any of this code could throw an exception, indicating failure. And future versions could introduce new exception types derived from existing exception types and now your `catch` blocks catch exception types that never existed before.

All of this stuff—object-oriented programming, compiler features, CLR features, and the enormous class library—is what makes the .NET Framework such a compelling software development platform.[4] My point is that all of this stuff introduces points of failure into your code, which you have little control over. As long as everything is working great, all is well: we write code easily, the code is easy to read and maintain. But, when something goes wrong, it is nearly impossible to fully understand what went wrong and why.

[4] I should also add that Microsoft Visual Studio's editor, IntelliSense support, code snippet support, templates, extensibility system, debugging system, and various other tools also contribute to making the platform compelling for developers. However, I leave this out of the main discussion because it has no impact on the behavior of the code at run time.

Here is an example that should really help get my point across.

```
private static Object OneStatement(Stream stream, Char charToFind) {
   return (charToFind + ": " + stream.GetType() + String.Empty + (stream.Position + 512M))
      .Where(c=>c == charToFind).ToArray();
}
```

This slightly contrived method contains just one C# statement in it, but this statement does an awful lot of work. In fact, here is the Intermediate Language (IL) the C# compiler produced for this method. (I've put some lines in boldface italics that are potential points of failure due to implicit operations that are occurring.)

```
.method private hidebysig static object OneStatement(
   class [mscorlib]System.IO.Stream stream, char charToFind) cil managed {
   .maxstack 4
   .locals init (
      [0] class Program/<>c__DisplayClass1 V_0,
      [1] object[] V_1)
   IL_0000: newobj instance void Program/<>c__DisplayClass1::.ctor()
   IL_0005: stloc.0
   IL_0006: ldloc.0
   IL_0007: ldarg.1
   IL_0008: stfld char Program/<>c__DisplayClass1::charToFind
   IL_000d: ldc.i4.5
   IL_000e: newarr [mscorlib]System.Object
   IL_0013: stloc.1
   IL_0014: ldloc.1
   IL_0015: ldc.i4.0
   IL_0016: ldloc.0
   IL_0017: ldfld char Program/<>c__DisplayClass1::charToFind
   IL_001c: box [mscorlib]System.Char
   IL_0021: stelem.ref
   IL_0022: ldloc.1
   IL_0023: ldc.i4.1
   IL_0024: ldstr ": "
   IL_0029: stelem.ref
   IL_002a: ldloc.1
   IL_002b: ldc.i4.2
   IL_002c: ldarg.0
   IL_002d: callvirt instance class [mscorlib]System.Type [mscorlib]System.Object::GetType()
   IL_0032: stelem.ref
   IL_0033: ldloc.1
   IL_0034: ldc.i4.3
   IL_0035: ldsfld string [mscorlib]System.String::Empty
   IL_003a: stelem.ref
   IL_003b: ldloc.1
   IL_003c: ldc.i4.4
   IL_003d: ldarg.0
   IL_003e: callvirt   instance int64 [mscorlib]System.IO.Stream::get_Position()
   IL_0043: call       valuetype [mscorlib]System.Decimal
                       [mscorlib]System.Decimal::op_Implicit(int64)
   IL_0048: ldc.i4     0x200
   IL_004d: newobj     instance void [mscorlib]System.Decimal::.ctor(int32)
```

```
    IL_0052: call valuetype [mscorlib]System.Decimal [mscorlib]System.Decimal::op_Addition
                (valuetype [mscorlib]System.Decimal, valuetype [mscorlib]System.Decimal)
    IL_0057: box [mscorlib]System.Decimal
    IL_005c: stelem.ref
    IL_005d: ldloc.1
    IL_005e: call string [mscorlib]System.String::Concat(object[])
    IL_0063: ldloc.0
    IL_0064: ldftn instance bool Program/<>c__DisplayClass1::<OneStatement>b__0(char)
    IL_006a: newobj instance
          void [mscorlib]System.Func`2<char, bool>::.ctor(object, native int)
    IL_006f: call class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>
                [System.Core]System.Linq.Enumerable::Where<char>(
                class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>,
                class [mscorlib]System.Func`2<!!0, bool>)
    IL_0074: call !!0[] [System.Core]System.Linq.Enumerable::ToArray<char>
              (class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>)
    IL_0079: ret
}
```

As you can see, an `OutOfMemoryException` is possible when constructing the `<>c__DisplayClass1` class (a compiler-generated type), the `Object[]` array, the `Func` delegate, and boxing the `char` and `Decimal`. Memory is also allocated internally when `Concat`, `Where`, and `ToArray` are called. Constructing the `Decimal` instance could cause its type constructor to be invoked, resulting in a `TypeInitializationException`.[5] And then, there are the implicit calls to `Decimal`'s `op_Implicit` operator and its `op_Addition` operator methods, which could do anything, including throwing an `OverflowException`.

Querying `Stream`'s `Position` property is interesting. First, it is a virtual property and so my `OneStatement` method has no idea what code will actually execute which could throw any exception at all. Second, `Stream` is derived from `MarshalByRefObject`, so the stream argument could actually refer to a proxy object which itself refers to an object in another AppDomain. The other AppDomain could be unloaded, so an `AppDomainUnloadedException` could also be thrown here.

Of course, all the methods that are being called are methods that I personally have no control over because they are produced by Microsoft. And it's entirely possible that Microsoft might change how these methods are implemented in the future, so they could throw new exception types that I could not possibly know about on the day I wrote the `OneStatement` method. How can I possibly write my `OneStatement` method to be completely robust against all possible failures? By the way, the opposite is also a problem: a `catch` block could catch an exception type derived from the specified exception type and now I'm executing recovery code for a different kind of failure.

So now that you have a sense of all the possible failures, you can probably see why it has become culturally acceptable to not write truly robust and reliable code: it is simply impractical. Moreover, one could argue that it is actually impossible. The fact that errors do not occur frequently is another reason why it has become culturally acceptable. Because errors (like `OutOfMemoryException`) occur very infrequently, the community has decided to trade truly reliable code for programmer productivity.

[5] By the way, `System.Char`, `System.String`, `System.Type`, and `System.IO.Stream` all define class constructors, which could all potentially cause a `TypeInitializationException` to be thrown at some point in this application.

One of the nice things about exceptions is that an unhandled one causes your application to terminate. This is nice because during testing, you will discover problems quickly and the information you get with an unhandled exception (error message and stack trace) are usually enough to allow you to fix your code. Of course, a lot of companies don't want their application to just terminate after it has been tested and deployed, so a lot of developers insert code to catch `System.Exception`, the base class of all exception types. However, the problem with catching `System.Exception` and allowing the application to continue running is that state may be corrupted.

Earlier in this chapter, I showed an `Account` class that defines a `Transfer` method whose job is to transfer money from one account to another account. What if, when this `Transfer` method is called, it successfully subtracts money from the `from` account and then throws an exception before it adds money to the `to` account? If calling code catches `System.Exception` and continues running, then the state of the application is corrupted: both the `from` and `to` accounts have less money in them then they should. Because we are talking about money here, this state corruption wouldn't just be considered a simple bug, it would definitely be considered a security bug. If the application continues running, it will attempt to perform more transfers to and from various accounts and now state corruption is running rampant within the application.

One could say that the `Transfer` method itself should catch `System.Exception` and restore money back into the `from` account. And this might actually work out OK if the `Transfer` method is simple enough. But if the `Transfer` method produces an audit record of the withdrawn money or if other threads are manipulating the same account at the same time, then attempting to undo the operation could fail as well, producing yet another thrown exception. And now, state corruption is getting worse, not better.

Note One could argue that knowing where something went wrong is more useful than knowing *what* error occurred. For example, it might be more useful to know that transferring money out of an account failed instead of knowing that `Transfer` failed due to a `SecurityException` or `OutOfMemoryException`, etc. In fact, the Win32 error model works this way: methods return `true/false` to indicate success/failure so you know which method failed. Then, if your program cares about *why* it failed, it calls the Win32 `GetLastError` method. `System.Exception` does have a `Source` property that tells you the method that failed. But this property is a `String` that you'd have to parse, and if two methods internally call the same method, you can't tell from the `Source` property alone which method your code called that failed. Instead, you'd have to parse the `String` returned from `Exception's StackTrace` property to get this information. Because this is so difficult, I've never seen anyone actually write code to do it.

There are several things you can do to *help* mitigate state corruption:

- The CLR doesn't allow a thread to be aborted when executing code inside a `catch` or `finally` block. So, we could make the `Transfer` method more robust simply by doing the following.

  ```
  public static void Transfer(Account from, Account to, Decimal amount) {
     try { /* do nothing in here */ }
     finally {
        from -= amount;
        // Now, a thread abort (due to Thread.Abort/AppDomain.Unload) can't happen here
        to += amount;
     }
  }
  ```

  However, it is absolutely not recommended that you write all your code in `finally` blocks! You should only use this technique for modifying extremely sensitive state.

- You can use the `System.Diagnostics.Contracts.Contract` class to apply code contracts to your methods. Code contracts give you a way to validate arguments and other variables before you attempt to modify state by using these arguments/variables. If the arguments/variables meet the contract, then the *chance* of corrupted state is minimized (not completely eliminated). If a contract fails, then an exception is thrown before any state has been modified. I will talk about code contracts later in this chapter.

- You can use constrained execution regions (CERs), which give you a way to take some CLR uncertainty out of the picture. For example, before entering a `try` block, you can have the CLR load any assemblies needed by code in any associated `catch` and `finally` blocks. In addition, the CLR will compile all the code in the `catch` and `finally` blocks including all the methods called from within those blocks. This will eliminate a bunch of potential exceptions (including `FileLoadException`, `BadImageFormatException`, `InvalidProgramException`, `FieldAccessException`, `MethodAccessException`, `MissingFieldException`, and `MissingMethodException`) from occurring when trying to execute error recovery code (in `catch` blocks) or cleanup code (in the `finally` block). It will also reduce the potential for `OutOfMemoryException` and some other exceptions as well. I talk about CERs later in this chapter.

- Depending on where the state lives, you can use transactions which ensure that all state is modified or no state is modified. If the data is in a database, for example, transactions work well. Windows also now supports transacted registry and file operations (on an NTFS volume only), so you might be able to use this; however, the .NET Framework doesn't expose this functionality directly today. You will have to P/Invoke to native code to leverage it. See the `System.Transactions.TransactionScope` class for more details about this.

- You can design your methods to be more explicit. For example, the `Monitor` class is typically used for taking/releasing a thread synchronization lock as follows.

```
public static class SomeType {
   private static Object s_myLockObject = new Object();

   public static void SomeMethod () {
      Monitor.Enter(s_myLockObject);  // If this throws, did the lock get taken or
                                      // not? If it did, then it won't get released!
      try {
         // Do thread-safe operation here...
      }
      finally {
         Monitor.Exit(s_myLockObject);
      }
   }
   // ...
}
```

  Due to the problem just shown, the overload of the preceding `Monitor`'s `Enter` method used is now discouraged, and it is recommended that you rewrite the preceding code as follows.

```
public static class SomeType {
   private static Object s_myLockObject = new Object();

   public static void SomeMethod () {
      Boolean lockTaken = false;  // Assume the lock was not taken
      try {
         // This works whether an exception is thrown or not!
         Monitor.Enter(s_myLockObject, ref lockTaken);

         // Do thread-safe operation here...
      }
      finally {
         // If the lock was taken, release it
         if (lockTaken) Monitor.Exit(s_myLockObject);
      }
   }
   // ...
}
```

Although the explicitness in this code is an improvement, in the case of thread synchronization locks, the recommendation now is to not use them with exception handling at all. See Chapter 30, "Hybrid Thread Synchronization Constructs," for more details about this.

If, in your code, you have determined that state has already been corrupted beyond repair, then you should destroy any corrupted state so that it can cause no additional harm. Then, restart your application so your state initializes itself to a good condition and hopefully, the state corruption will not happen again. Because managed state cannot leak outside of an AppDomain, you can destroy any corrupted state that lives within an AppDomain by unloading the entire AppDomain by calling `AppDomain`'s `Unload` method (see Chapter 22 for details).

And, if you feel that your state is so bad that the whole process should be terminated, then you can call `Environment`'s static `FailFast` method.

```
public static void FailFast(String message);
public static void FailFast(String message, Exception exception);
```

This method terminates the process without running any active `try/finally` blocks or `Finalize` methods. This is good because executing more code while state is corrupted could easily make matters worse. However, `FailFast` will allow any `CriticalFinalizerObject`-derived objects, discussed in Chapter 21, "The Managed Heap and Garbage Collection," a chance to clean up. This is usually OK because they tend to just close native resources, and Windows state is probably fine even if the CLR's state or your application's state is corrupted. The `FailFast` method writes the message string and optional exception (usually the exception captured in a `catch` block) to the Windows Application event log, produces a Windows error report, creates a memory dump of your application, and then terminates the current process.

Important Most of the FCL code does not ensure that state remains good in the case of an unexpected exception. If your code catches an exception that passes through FCL code and then continues to use FCL objects, there is a chance that these objects will behave unpredictably. It's a shame that more FCL objects don't maintain their state better in the face of unexpected exceptions or call `FailFast` if their state cannot be restored.

The point of this discussion is to make you aware of the potential problems related to using the CLR's exception-handling mechanism. Most applications cannot tolerate running with a corrupted state because it leads to incorrect data and possible security holes. If you are writing an application that cannot tolerate terminating (like an operating system or database engine), then managed code is not a good technology to use. And although Microsoft Exchange Server is largely written in managed code, it uses a native database to store email messages. The native database is called the Extensible Storage Engine; it ships with Windows, and can usually be found at C:\Windows\System32\EseNT.dll. Your applications can also use this engine if you'd like; search for "Extensible Storage Engine" on the Microsoft MSDN website.

Managed code is a good choice for applications that can tolerate an application terminating when state corruption has possibly occurred. There are many applications that fall into this category. Also, it takes significantly more resources and skills to write a robust native class library or application; for many applications, managed code is the better choice because it greatly enhances programmer productivity.

Guidelines and Best Practices

Understanding the exception mechanism is certainly important. It is equally important to understand how to use exceptions wisely. All too often, I see library developers catching all kinds of exceptions, preventing the application developer from knowing that a problem occurred. In this section, I offer some guidelines for developers to be aware of when using exceptions.

> **Important** If you're a *class library developer* developing types that will be used by other developers, take these guidelines very seriously. You have a huge responsibility: you're trying to design the type in your class library so that it makes sense for a wide variety of applications. Remember that you don't have intimate knowledge of the code that you're calling back (via delegates, virtual methods, or interface methods). And you don't know which code is calling you. It's not feasible to anticipate every situation in which your type will be used, so don't make any policy decisions. Your code must not decide what conditions constitute an error; let the caller make that decision.
>
> In addition, watch state very closely and try not to corrupt it. Verify arguments passed to your method by using code contracts (discussed later in this chapter). Try not to modify state at all. If you do modify state, be ready for a failure and then try to restore state. If you follow the guidelines in this chapter, application developers will not have a difficult time using the types in your class library.
>
> If you're an *application developer*, define whatever policy you think is appropriate. Following the design guidelines in this chapter will help you discover problems in your code before it is released, allowing you to fix them and make your application more robust. However, feel free to diverge from these guidelines after careful consideration. You get to set the policy. For example, application code can get more aggressive about catching exceptions than class library code.

Use `finally` Blocks Liberally

I think `finally` blocks are awesome! They allow you to specify a block of code that's guaranteed to execute no matter what kind of exception the thread throws. You should use `finally` blocks to clean up from any operation that successfully started before returning to your caller or allowing code following the `finally` block to execute. You also frequently use `finally` blocks to explicitly dispose of any objects to avoid resource leaking. Here's an example that has all cleanup code (closing the file) in a `finally` block.

```
using System;
using System.IO;

public sealed class SomeType {
   private void SomeMethod() {
      FileStream fs = new FileStream(@"C:\Data.bin ", FileMode.Open);
      try {
         // Display 100 divided by the first byte in the file.
         Console.WriteLine(100 / fs.ReadByte());
      }
      finally {
         // Put cleanup code in a finally block to ensure that the file gets closed regardless
         // of whether or not an exception occurs (for example, the first byte was 0).
         if (fs != null) fs.Dispose();
      }
   }
}
```

Ensuring that cleanup code always executes is so important that many programming languages offer constructs that make writing cleanup code easier. For example, the C# language automatically emits `try`/`finally` blocks whenever you use the `lock`, `using`, and `foreach` statements. The C# compiler also emits `try`/`finally` blocks whenever you override a class's destructor (the `Finalize` method). When using these constructs, the compiler puts the code you've written inside the `try` block and automatically puts the cleanup code inside the `finally` block. Specifically:

- When you use the `lock` statement, the lock is released inside a `finally` block.
- When you use the `using` statement, the object has its `Dispose` method called inside a `finally` block.
- When you use the `foreach` statement, the `IEnumerator` object has its `Dispose` method called inside a `finally` block.
- When you define a destructor method, the base class's `Finalize` method is called inside a `finally` block.

For example, the following C# code takes advantage of the `using` statement. This code is shorter than the code shown in the previous example, but the code that the compiler generates is identical to the code generated in the previous example.

```
using System;
using System.IO;

internal sealed class SomeType {
   private void SomeMethod() {
      using (FileStream fs = new FileStream(@"C:\Data.bin", FileMode.Open)) {
         // Display 100 divided by the first byte in the file.
         Console.WriteLine(100 / fs.ReadByte());
      }
   }
}
```

For more about the `using` statement, see Chapter 21; and for more about the `lock` statement, see Chapter 30.

Don't `Catch` Everything

A ubiquitous mistake made by developers who have not been properly trained on the proper use of exceptions is to use `catch` blocks too often and improperly. When you catch an exception, you're stating that you expected this exception, you understand why it occurred, and you know how to deal with it. In other words, you're defining a policy for the application. This all goes back to the "Trading Reliability for Productivity" section earlier in this chapter.

All too often, I see code like this.

```
try {
   // try to execute code that the programmer knows might fail...
}
catch (Exception) {
   ...
}
```

This code indicates that it was expecting any and all exceptions and knows how to recover from *any* and *all* situations. How can this possibly be? A type that's part of a class library should *never*, *ever*, *under any circumstance* catch and swallow all exceptions because there is no way for the type to know exactly how the application intends to respond to an exception. In addition, the type will frequently call out to application code via a delegate, virtual method, or interface method. If the application code throws an exception, another part of the application is probably expecting to catch this exception. The exception should be allowed to filter its way up the call stack and let the application code handle the exception as it sees fit.

If the exception is unhandled, the CLR terminates the process. I'll discuss unhandled exceptions later in this chapter. Most unhandled exceptions will be discovered during testing of your code. To fix these unhandled exceptions, you will either modify the code to look for a specific exception, or you will rewrite the code to eliminate the conditions that cause the exception to be thrown. The final version of the code that will be running in a production environment should see very few unhandled exceptions and will be extremely robust.

Note In some cases, a method that can't complete its task will detect that some object's state has been corrupted and cannot be restored. Allowing the application to continue running might result in unpredictable behavior or security vulnerabilities. When this situation is detected, that method should not throw an exception; instead, it should force the process to terminate immediately by calling `System.Environment`'s `FailFast` method.

By the way, it *is* OK to catch `System.Exception` and execute some code inside the `catch` block's braces as long as you re-throw the exception at the bottom of that code. Catching `System.Exception` and swallowing the exception (not re-throwing it) should never be done because it hides failures that allow the application to run with unpredictable results and potential security vulnerabilities. Visual Studio's code analysis tool (FxCopCmd.exe) will flag any code that contains a `catch (Exception)` block unless there is a `throw` statement included in the block's code. The "Backing Out of a Partially

Completed Operation When an Unrecoverable Exception Occurs—Maintaining State" section, coming shortly in this chapter, will discuss this pattern.

Finally, it is OK to catch an exception occurring in one thread and re-throw the exception in another thread. The Asynchronous Programming Model (discussed in Chapter 28, "I/O-Bound Asynchronous Operations") supports this. For example, if a thread pool thread executes code that throws an exception, the CLR catches and swallows the exception and allows the thread to return to the thread pool. Later, a thread should call an `EndXxx` method to determine the result of the asynchronous operation. The `EndXxx` method will throw the same exception object that was thrown by the thread pool thread that did the actual work. In this scenario, the exception is being swallowed by the first thread; however, the exception is being re-thrown by the thread that called the `EndXxx` method, so it is not being hidden from the application.

Recovering Gracefully from an Exception

Sometimes you call a method knowing in advance some of the exceptions that the method might throw. Because you expect these exceptions, you might want to have some code that allows your application to recover gracefully from the situation and continue running. Here's an example in pseudocode.

```
public String CalculateSpreadsheetCell(Int32 row, Int32 column) {
   String result;
   try {
      result = /* Code to calculate value of a spreadsheet's cell */
   }
   catch (DivideByZeroException) {
      result = "Can't show value: Divide by zero";
   }
   catch (OverflowException) {
      result = "Can't show value: Too big";
   }
   return result;
}
```

This pseudocode calculates the contents of a cell in a spreadsheet and returns a string representing the value to the caller so that the caller can display the string in the application's window. However, a cell's contents might be the result of dividing one cell by another cell. If the cell containing the denominator contains 0, the CLR will throw a `DivideByZeroException` object. In this case, the method catches this specific exception and returns a special string that will be displayed to the user. Similarly, a cell's contents might be the result of multiplying one cell by another. If the multiplied value doesn't fit in the number of bits allowed, the CLR will throw an `OverflowException` object, and again, a special string will be displayed to the user.

When you catch specific exceptions, fully understand the circumstances that cause the exception to be thrown, and know what exception types are derived from the exception type you're catching. Don't catch and handle `System.Exception` (without re-throwing) because it's not feasible for you to know all of the possible exceptions that could be thrown within your `try` block (especially if you consider the `OutOfMemoryException` or the `StackOverflowException`, to name two).

Backing Out of a Partially Completed Operation When an Unrecoverable Exception Occurs—Maintaining State

Usually, methods call several other methods to perform a single abstract operation. Some of the individual methods might complete successfully, and some might not. For example, let's say that you're serializing a set of objects to a disk file. After serializing 10 objects, an exception is thrown. (Perhaps the disk is full or the next object to be serialized isn't marked with the `Serializable` custom attribute.) At this point, the exception should filter up to the caller, but what about the state of the disk file? The file is now corrupted because it contains a partially serialized object graph. It would be great if the application could back out of the partially completed operation so that the file would be in the state it was in before any objects were serialized into it. The following code demonstrates the correct way to implement this.

```
public void SerializeObjectGraph(FileStream fs, IFormatter formatter, Object rootObj) {

   // Save the current position of the file.
   Int64 beforeSerialization = fs.Position;

   try {
      // Attempt to serialize the object graph to the file.
      formatter.Serialize(fs, rootObj);
   }
   catch {  // Catch any and all exceptions.
      // If ANYTHING goes wrong, reset the file back to a good state.
      fs.Position = beforeSerialization;

      // Truncate the file.
      fs.SetLength(fs.Position);

      // NOTE: The preceding code isn't in a finally block because
      // the stream should be reset only when serialization fails.

      // Let the caller(s) know what happened by re-throwing the SAME exception.
      throw;
   }
}
```

To properly back out of the partially completed operation, write code that catches all exceptions. Yes, catch *all* exceptions here because you don't care what kind of error occurred; you need to put your data structures back into a consistent state. After you've caught and handled the exception, don't swallow it—let the caller know that the exception occurred. You do this by re-throwing the same exception. In fact, C# and many other languages make this easy. Just use C#'s `throw` keyword without specifying anything after `throw`, as shown in the previous code.

Notice that the `catch` block in the previous example doesn't specify any exception type because I want to catch any and all exceptions. In addition, the code in the `catch` block doesn't need to know exactly what kind of exception was thrown, just that something went wrong. Fortunately, C# lets me do this easily just by not specifying any exception type and by making the `throw` statement re-throw whatever object is caught.

Hiding an Implementation Detail to Maintain a “Contract”

In some situations, you might find it useful to catch one exception and re-throw a different exception. The only reason to do this is to maintain the meaning of a method’s contract. Also, the new exception type that you throw should be a specific exception (an exception that’s not used as the base type of any other exception type). Imagine a PhoneBook type that defines a method that looks up a phone number from a name, as shown in the following pseudocode.

```
internal sealed class PhoneBook {
   private String m_pathname;  // path name of file containing the address book

   // Other methods go here.

   public String GetPhoneNumber(String name) {
      String phone;
      FileStream fs = null;
      try {
         fs = new FileStream(m_pathname, FileMode.Open);
         // Code to read from fs until name is found goes here
         phone = /* the phone # found */
      }
      catch (FileNotFoundException e) {
         // Throw a different exception containing the name, and
         // set the originating exception as the inner exception.
         throw new NameNotFoundException(name, e);
      }
      catch (IOException e) {
         // Throw a different exception containing the name, and
         // set the originating exception as the inner exception.
         throw new NameNotFoundException(name, e);
      }
      finally {
         if (fs != null) fs.Close();
      }
      return phone;
   }
}
```

The phone book data is obtained from a file (versus a network connection or database). However, the user of the PhoneBook type doesn’t know this because this is an implementation detail that could change in the future. So if the file isn’t found or can’t be read for any reason, the caller would see a `FileNotFoundException` or `IOException`, which wouldn’t be anticipated. In other words, the file’s existence and ability to be read is not part of the method’s implied contract: there is no way the caller could have guessed this. So the `GetPhoneNumber` method catches these two exception types and throws a new `NameNotFoundException`.

When using this technique, you should catch specific exceptions that you fully understand the circumstances that cause the exception to be thrown. And, you should also know what exception types are derived from the exception type you’re catching.

Throwing an exception still lets the caller know that the method cannot complete its task, and the NameNotFoundException type gives the caller an abstracted view as to why. Setting the inner exception to FileNotFoundException or IOException is important so that the real cause of the exception isn't lost. Besides, knowing what caused the exception could be useful to the developer of the PhoneBook type and possibly to a developer using the PhoneBook type.

Important When you use this technique, you are lying to callers about two things. First, you are lying about what actually went wrong. In my example, a file was not found but I'm reporting that a name was not found. Second, you are lying about where the failure occurred. If the FileNotFoundException were allowed to propagate up the call stack, its StackTrace property would reflect that the error occurred inside FileStream's constructor. But when I swallow this exception and throw a new NameNotFoundException, the stack trace will indicate that the error occurred inside the catch block, several lines away from where the real exception was thrown. This can make debugging very difficult, so this technique should be used with great care.

Now let's say that the PhoneBook type was implemented a little differently. Assume that the type offers a public PhoneBookPathname property that allows the user to set or get the path name of the file in which to look up a phone number. Because the user is aware of the fact that the phone book data comes from a file, I would modify the GetPhoneNumber method so that it doesn't catch any exceptions; instead, I let whatever exception is thrown propagate out of the method. Note that I'm not changing any parameters of the GetPhoneNumber method, but I am changing how it's abstracted to users of the PhoneBook type. Users now expect a path to be part of the PhoneBook's contract.

Sometimes developers catch one exception and throw a new exception in order to add additional data or context to an exception. However, if this is all you want to do, you should just catch the exception type you want, add data to the exception object's Data property collection, and then re-throw the same exception object.

```
private static void SomeMethod(String filename) {
   try {
      // Do whatevere here...
   }
   catch (IOException e) {
      // Add the filename to the IOException object
      e.Data.Add("Filename", filename);

      throw;   // re-throw the same exception object that now has additional data in it
   }
}
```

Here is a good use of this technique: when a type constructor throws an exception that is not caught within the type constructor method, the CLR internally catches that exception and throws a new TypeInitializationException instead. This is useful because the CLR emits code within your methods to implicitly call type constructors.[6] If the type constructor threw a DivideByZeroException,

[6] For more information about this, see the "Type Constructors" section in Chapter 8, "Methods."

your code might try to catch it and recover from it but you didn't even know you were invoking the type constructor. So the CLR converts the `DivideByZeroException` into a `TypeInitialization-Exception` so that you know clearly that the exception occurred due to a type constructor failing; the problem wasn't with your code.

On the other hand, here is a bad use of this technique: when you invoke a method via reflection, the CLR internally catches any exception thrown by the method and converts it to a `Target-InvocationException`. This is incredibly annoying because you must now catch the `Target-InvocationException` object and look at its `InnerException` property to discern the real reason for the failure. In fact, when using reflection, it is common to see code that looks like this.

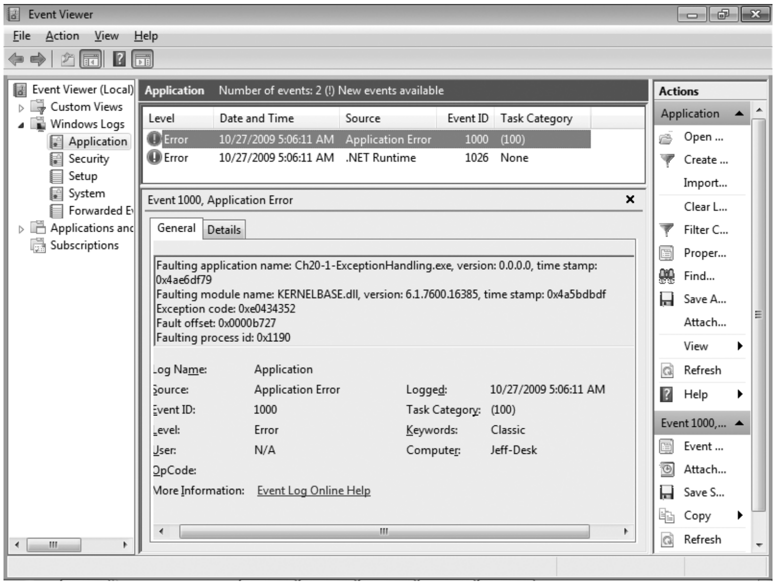
```
private static void Reflection(Object o) {
   try {
      // Invoke a DoSomething method on this object
      var mi = o.GetType().GetMethod("DoSomething");
      mi.Invoke(o, null);  // The DoSomething method might throw an exception
   }
   catch (System.Reflection.TargetInvocationException e) {
      // The CLR converts reflection-produced exceptions to TargetInvocationException
      throw e.InnerException; // Re-throw what was originally thrown
   }
}
```

I have good news though: if you use C#'s `dynamic` primitive type (discussed in Chapter 5, "Primitive, Reference, and Value Types") to invoke a member, the compiler-generated code does not catch any and all exceptions and throw a `TargetInvocationException` object; the originally thrown exception object simply walks up the stack. For many developers, this is a good reason to prefer using C#'s `dynamic` primitive type rather than reflection.

Unhandled Exceptions

When an exception is thrown, the CLR climbs up the call stack looking for `catch` blocks that match the type of the exception object being thrown. If no `catch` block matches the thrown exception type, an *unhandled exception* occurs. When the CLR detects that any thread in the process has had an unhandled exception, the CLR terminates the process. An unhandled exception identifies a situation that the application didn't anticipate and is considered to be a true bug in the application. At this point, the bug should be reported back to the company that publishes the application. Hopefully, the publisher will fix the bug and distribute a new version of the application.

Class library developers should not even think about unhandled exceptions. Only application developers need to concern themselves with unhandled exceptions, and the application should have a policy in place for dealing with unhandled exceptions. Microsoft actually recommends that application developers just accept the CLR's default policy. That is, when an application gets an unhandled exception, Windows writes an entry to the system's event log. You can see this entry by opening the Event Viewer application and then looking under the Windows Logs ➔ Application node in the tree, as shown in Figure 20-1.



Application Number of events: 2 (!) New events available

Level	Date and Time	Source	Event ID	Task Category
Error	10/27/2009 5:06:11 AM	Application Error	1000	(100)
Error	10/27/2009 5:06:11 AM	.NET Runtime	1026	None

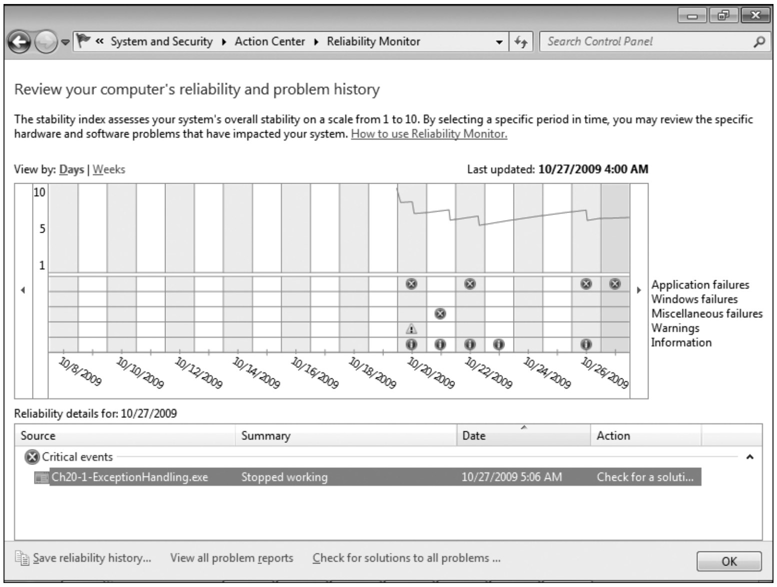
Event 1000, Application Error

```
Faulting application name: Ch20-1-ExceptionHandling.exe, version: 0.0.0.0, time stamp:
0x4ae6df79
Faulting module name: KERNELBASE.dll, version: 6.1.7600.16385, time stamp: 0x4a5bdbdf
Exception code: 0xe0434352
Fault offset: 0x0000b727
Faulting process id: 0x1190
```

Log Name:	Application		
Source:	Application Error	Logged:	10/27/2009 5:06:11 AM
Event ID:	1000	Task Category:	(100)
Level:	Error	Keywords:	Classic
User:	N/A	Computer:	Jeff-Desk
OpCode:			
More Information:	Event Log Online Help		

FIGURE 20-1 Windows Event log showing an application that terminated due to an unhandled exception.

However, you can get more interesting details about the problem by using the Windows Reliability Monitor applet. To start Reliability Monitor, open the Windows Control Panel and search for "reliability history". From here, you can see the applications that have terminated due to an unhandled exception in the bottom pane, as shown in Figure 20-2.



Review your computer's reliability and problem history

The stability index assesses your system's overall stability on a scale from 1 to 10. By selecting a specific period in time, you may review the specific hardware and software problems that have impacted your system. How to use Reliability Monitor.

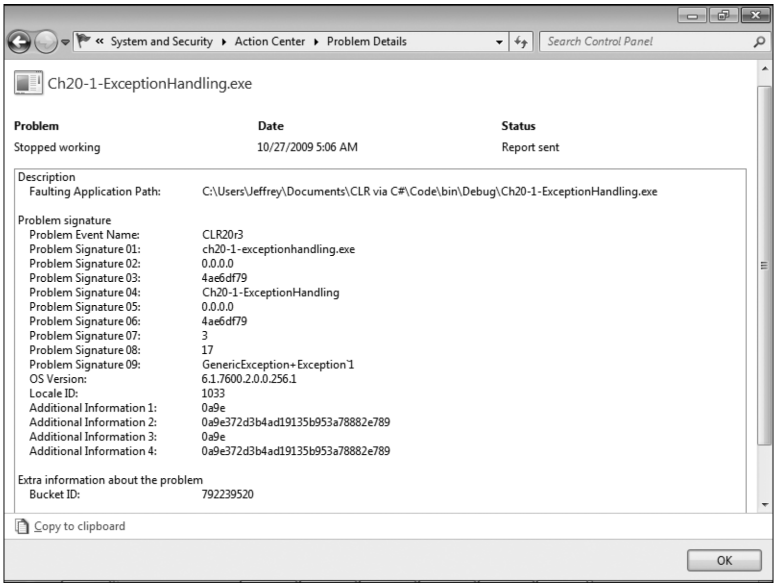
Last updated: **10/27/2009 4:00 AM**

Reliability details for: 10/27/2009

Source	Summary	Date	Action
Critical events			
Ch20-1-ExceptionHandling.exe	Stopped working	10/27/2009 5:06 AM	Check for a soluti...

FIGURE 20-2 Reliability Monitor showing an application that terminated due to an unhandled exception.

To see more details about the terminated application, double-click a terminated application in Reliability Monitor. The details will look something like Figure 20-3 and the meaning of the problem signatures are described in Table 20-2. All unhandled exceptions produced by managed applications are placed in the CLR20r3 bucket.



Ch20-1-ExceptionHandling.exe

Problem	Date	Status
Stopped working	10/27/2009 5:06 AM	Report sent

```
Description
   Faulting Application Path:    C:\Users\Jeffrey\Documents\CLR via C#\Code\bin\Debug\Ch20-1-ExceptionHandling.exe

Problem signature
   Problem Event Name:           CLR20r3
   Problem Signature 01:         ch20-1-exceptionhandling.exe
   Problem Signature 02:         0.0.0.0
   Problem Signature 03:         4ae6df79
   Problem Signature 04:         Ch20-1-ExceptionHandling
   Problem Signature 05:         0.0.0.0
   Problem Signature 06:         4ae6df79
   Problem Signature 07:         3
   Problem Signature 08:         17
   Problem Signature 09:         GenericException+Exception`1
   OS Version:                   6.1.7600.2.0.0.256.1
   Locale ID:                    1033
   Additional Information 1:     0a9e
   Additional Information 2:     0a9e372d3b4ad19135b953a78882e789
   Additional Information 3:     0a9e
   Additional Information 4:     0a9e372d3b4ad19135b953a78882e789

Extra information about the problem
   Bucket ID:                    792239520
```

FIGURE 20-3 Reliability Monitor showing more details about the failed application.

TABLE 20-2 Problem Signatures

Problem Signature	Description*
01	EXE file's name (32-character limit)
02	EXE file's assembly version number
03	EXE file's timestamp
04	EXE file's full assembly name (64-character limit)
05	Faulting assembly's version
06	Faulting assembly's timestamp
07	Faulting assembly's type and method. This value is a MethodDef metadata token (after stripping off the 0x06 high byte) identifying the method that threw the exception. From this value, you can use ILDasm.exe to determine the offending type and method.
08	Faulting method's IL instruction. This value is an offset within the faulting method of the IL instruction that threw the exception. From this value, you can use ILDasm.exe to determine the offending instruction.
09	Exception type thrown (32-character limit)

* If a string is beyond the allowed limit, then some intelligent truncations are performed, like removing "Exception" from the exception type or ".dll" from a file name. If the resulting string is still too long, then the CLR will create a value by hashing or base-64–encoding the string.

After recording information about the failing application, Windows displays the message box allowing the end user to send information about the failing application to Microsoft servers.[7] This is called *Windows Error Reporting*, and more information about it can be found at the Windows Quality website (*http://WinQual.Microsoft.com*).

Companies can optionally sign up with Microsoft to view this information about their own applications and components. Signing up is free, but it does require that your assemblies be signed with a VeriSign ID (also called a Software Publisher's Digital ID for Authenticode).

Naturally, you could also develop your own system for getting unhandled exception information back to you so that you can fix bugs in your code. When your application initializes, you can inform the CLR that you have a method that you want to be called whenever any thread in your application experiences an unhandled exception.

Unfortunately, every application model Microsoft produces has its own way of tapping into unhandled exceptions. The members that you want to look up in the FCL documentation are.

- For many applications, look at `System.AppDomain`'s `UnhandledException` event. Windows Store applications and Microsoft Silverlight applications cannot access this event.
- For a Windows Store App, look at `Windows.UI.Xaml.Application`'s `UnhandledException` event.
- For a Windows Forms application, look at `System.Windows.Forms.NativeWindow`'s `OnThreadException` virtual method, `System.Windows.Forms.Application`'s `OnThreadException` virtual method, and `System.Windows.Forms.Application`'s `ThreadException` event.
- For a Windows Presentation Foundation (WPF) application, look at `System.Windows.Application`'s `DispatcherUnhandledException` event and `System.Windows.Threading.Dispatcher`'s `UnhandledException` and `UnhandledExceptionFilter` events.
- For Silverlight, look at `System.Windows.Application`'s `UnhandledException` event.
- For an ASP.NET Web Form application, look at `System.Web.UI.TemplateControl`'s `Error` event. `TemplateControl` is the base class of the `System.Web.UI.Page` and `System.Web.UI.UserControl` classes. Furthermore, you should also look at `System.Web.HttpApplication`'s `Error` event.
- For a Windows Communication Foundation application, look at `System.ServiceModel.Dispatcher.ChannelDispatcher`'s `ErrorHandlers` property.

Before I leave this section, I'd like to say a few words about unhandled exceptions that could occur in a distributed application such as a website or web service. In an ideal world, a server application that experiences an unhandled exception should log it, send some kind of notification back to the

[7] You can actually disable this message box by using P/Invoke to call Win32's `SetErrorMode` function, passing in `SEM_NOGPFAULTERRORBOX`.

client indicating that the requested operation could not complete, and then the server should terminate. Unfortunately, we don't live in an ideal world, and therefore, it may not be possible to send a failure notification back to the client. For some stateful servers (such as Microsoft SQL Server), it may not be practical to terminate the server and start a brand new instance.

For a server application, information about the unhandled exception should not be returned to the client because there is little a client could do about it, especially if the client is implemented by a different company. Furthermore, the server should divulge as little information about itself as possible to its clients to reduce that potential of the server being hacked.

Note The CLR considers some exceptions thrown by native code as *corrupted state exceptions (CSEs)* because they are usually the result of a bug in the CLR itself or in some native code for which the managed developer has no control over. By default, the CLR will not let managed code catch these exceptions and `finally` blocks will not execute. Here is the list of native Win32 exceptions that are considered CSEs.

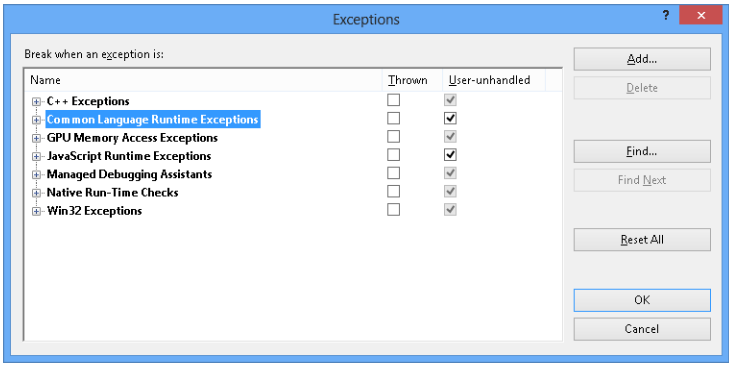
```
EXCEPTION_ACCESS_VIOLATION          EXCEPTION_STACK_OVERFLOW
EXCEPTION_ILLEGAL_INSTRUCTION       EXCEPTION_IN_PAGE_ERROR
EXCEPTION_INVALID_DISPOSITION       EXCEPTION_NONCONTINUABLE_EXCEPTION
EXCEPTION_PRIV_INSTRUCTION          STATUS_UNWIND_CONSOLIDATE.
```

Individual managed methods can override the default and catch these exceptions by applying the `System.Runtime.ExceptionServices.HandleProcessCorruptedStateExceptionsAttribute` to the method. In addition, the method must have the `System.Security.SecurityCriticalAttribute` applied to it. You can also override the default for an entire process by setting the `legacyCorruptedStateExceptionPolicy` element in the application's Extensible Markup Language (XML) configuration file to `true`. The CLR converts most of these to a `System.Runtime.InteropServices.SEHException` object except for `EXCEPTION_ACCESS_VIOLATION`, which is converted to a `System.AccessViolationException` object, and `EXCEPTION_STACK_OVERFLOW`, which is converted to a `System.StackOverflowException` object.

Note Just before invoking a method, you could check for ample stack space by calling the `RuntimeHelper` class's `EnsureSufficientExecutionStack` method. This method checks whether the calling thread has enough stack space available to execute the average method (which is not well defined). If there is insufficient stack space, the method throws an `InsufficientExecutionStackException` which you can catch. The `EnsureSufficientExecutionStack` method takes no arguments and returns `void`. This method is typically used by recursive methods.

Debugging Exceptions

The Visual Studio debugger offers special support for exceptions. With a solution open, choose Exceptions from the Debug menu, and you'll see the dialog box shown in Figure 20-4.

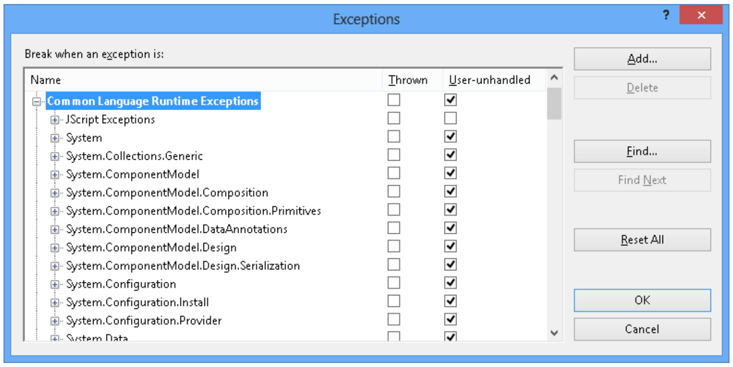

Exceptions

Break when an exception is:

Name	Thrown	User-unhandled
C++ Exceptions	☐	✓
Common Language Runtime Exceptions	☐	✓
GPU Memory Access Exceptions	☐	✓
JavaScript Runtime Exceptions	☐	✓
Managed Debugging Assistants	☐	✓
Native Run-Time Checks	☐	✓
Win32 Exceptions	☐	✓

FIGURE 20-4 The Exceptions dialog box, showing the different kinds of exceptions.

This dialog box shows the different kinds of exceptions that Visual Studio is aware of. For Common Language Runtime Exceptions, expanding the corresponding branch in the dialog box, as in Figure 20-5, shows the set of namespaces that the Visual Studio debugger is aware of.


Exceptions

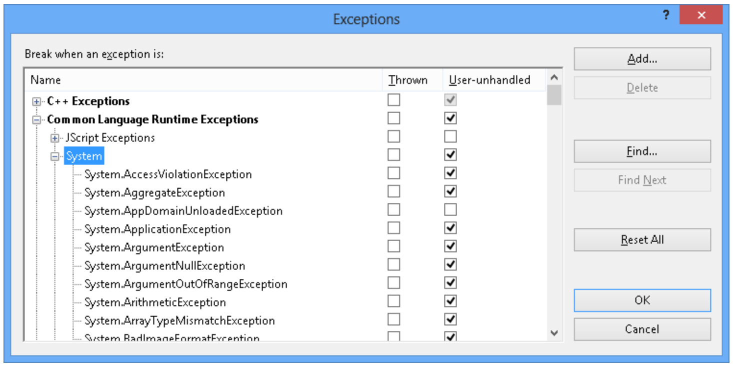
Break when an exception is:

Name	Thrown	User-unhandled
Common Language Runtime Exceptions	☐	✓
JScript Exceptions	☐	☐
System	☐	✓
System.Collections.Generic	☐	✓
System.ComponentModel	☐	✓
System.ComponentModel.Composition	☐	✓
System.ComponentModel.Composition.Primitives	☐	✓
System.ComponentModel.DataAnnotations	☐	✓
System.ComponentModel.Design	☐	✓
System.ComponentModel.Design.Serialization	☐	✓
System.Configuration	☐	✓
System.Configuration.Install	☐	✓
System.Configuration.Provider	☐	✓
System.Data	☐	✓

FIGURE 20-5 The Exceptions dialog box, showing CLR exceptions by namespace.

If you expand a namespace, you'll see all of the `System.Exception`-derived types defined within that namespace. For example, Figure 20-6 shows what you'll see if you open the `System` namespace.

For any exception type, if its Thrown check box is selected, the debugger will break as soon as that exception is thrown. At this point, the CLR has not tried to find any matching `catch` blocks. This is useful if you want to debug your code that catches and handles an exception. It is also useful when you suspect that a component or library may be swallowing or re-throwing exceptions, and you are uncertain where exactly to set a break point to catch it in the act.



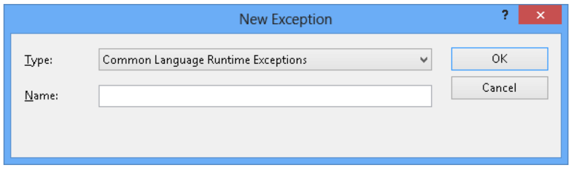
Break when an exception is:

Name	Thrown	User-unhandled
C++ Exceptions	☐	✓
Common Language Runtime Exceptions	☐	✓
JScript Exceptions	☐	☐
System	☐	✓
System.AccessViolationException	☐	✓
System.AggregateException	☐	✓
System.AppDomainUnloadedException	☐	☐
System.ApplicationException	☐	✓
System.ArgumentException	☐	✓
System.ArgumentNullException	☐	✓
System.ArgumentOutOfRangeException	☐	✓
System.ArithmeticException	☐	✓
System.ArrayTypeMismatchException	☐	✓
System.BadImageFormatException	☐	✓

FIGURE 20-6 The Exceptions dialog box, showing CLR exceptions defined in the `System` namespace.

If an exception type's Thrown check box is not selected, the debugger will also break where the exception was thrown, but only if the exception type was not handled. Developers usually leave the Thrown check box cleared because a handled exception indicates that the application anticipated the situation and dealt with it; the application continues running normally.

If you define your own exception types, you can add them to this dialog box by clicking Add. This causes the dialog box in Figure 20-7 to appear.



Type: Common Language Runtime Exceptions

Name:

FIGURE 20-7 Making Visual Studio aware of your own exception type: the New Exception dialog box.

In this dialog box, you first select the type of exception to be Common Language Runtime Exceptions, and then you can enter the fully qualified name of your own exception type. Note that the type you enter doesn't have to be a type derived from `System.Exception`; non–CLS-compliant types are fully supported. If you have two or more types with the same name but in different assemblies, there is no way to distinguish the types from one another. Fortunately, this situation rarely happens.

If your assembly defines several exception types, you must add them one at a time. In the future, I'd like to see this dialog box allow me to browse for an assembly and automatically import all `Exception`-derived types into Visual Studio's debugger. Each type could then be identified by assembly as well, which would fix the problem of having two types with the same name in different assemblies.

Exception-Handling Performance Considerations

The developer community actively debates the performance of exception handling. Some people claim that exception handling performance is so bad that they refuse to even use exception handling. However, I contend that in an object-oriented platform, exception handling is not an option; it is mandatory. And besides, if you didn't use it, what would you use instead? Would you have your methods return `true/false` to indicate success/failure or perhaps some error code `enum` type? Well, if you did this, then you have the worst of both worlds: the CLR and the class library code will throw exceptions and your code will return error codes. You'd have to now deal with both of these in your code.

It's difficult to compare performance between exception handling and the more conventional means of reporting exceptions (such as `HRESULT`s, special return codes, and so forth). If you write code to check the return value of every method call and filter the return value up to your own callers, your application's performance will be seriously affected. But performance aside, the amount of additional coding you must do and the potential for mistakes is incredibly high when you write code to check the return value of every method. Exception handling is a much better alternative.

However, exception handling has a price: unmanaged C++ compilers must generate code to track which objects have been constructed successfully. The compiler must also generate code that, when an exception is caught, calls the destructor of each successfully constructed object. It's great that the compiler takes on this burden, but it generates a lot of bookkeeping code in your application, adversely affecting code size and execution time.

On the other hand, managed compilers have it much easier because managed objects are allocated in the managed heap, which is monitored by the garbage collector. If an object is successfully constructed and an exception is thrown, the garbage collector will eventually free the object's memory. Compilers don't need to emit any bookkeeping code to track which objects are constructed successfully and don't need to ensure that a destructor has been called. Compared to unmanaged C++, this means that less code is generated by the compiler, and less code has to execute at run time, resulting in better performance for your application.

Over the years, I've used exception handling in different programming languages, different operating systems, and different CPU architectures. In each case, exception handling is implemented differently with each implementation having its pros and cons with respect to performance. Some implementations compile exception handling constructs directly into a method, whereas other implementations store information related to exception handling in a data table associated with the method—this table is accessed only if an exception is thrown. Some compilers can't inline methods that contain exception handlers, and some compilers won't enregister variables if the method contains exception handlers.

The point is that you can't determine how much additional overhead is added to an application when using exception handling. In the managed world, it's even more difficult to tell because your assembly's code can run on any platform that supports the .NET Framework. So the code produced by the JIT compiler to manage exception handling when your assembly is running on an x86 machine will be very different from the code produced by the JIT compiler when your code is running on an x64 or ARM processor. Also, JIT compilers associated with other CLR implementations (such as Microsoft's .NET Compact Framework or the open-source Mono project) are likely to produce different code.

Actually, I've been able to test some of my own code with a few different JIT compilers that Microsoft has internally, and the difference in performance that I've observed has been quite dramatic and surprising. The point is that you must test your code on the various platforms that you expect your users to run on, and make changes accordingly. Again, I wouldn't worry about the performance of using exception handling; the benefits typically far outweigh any negative performance impact.

If you're interested in seeing how exception handling impacts the performance of your code, you can use the Performance Monitor tool that comes with Windows. The screen in Figure 20-8 shows the exception-related counters that are installed along with the .NET Framework.

Occasionally, you come across a method that you call frequently that has a high failure rate. In this situation, the performance hit of having exceptions thrown can be intolerable. For example, Microsoft heard back from several customers who were calling `Int32`'s `Parse` method, frequently passing in data entered from an end user that could not be parsed. Because `Parse` was called frequently, the performance hit of throwing and catching the exceptions was taking a large toll on the application's overall performance.

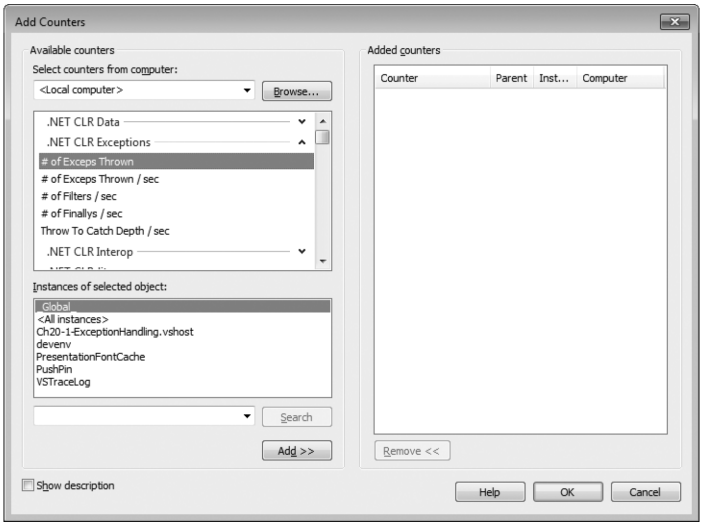


FIGURE 20-8 Performance Monitor showing the .NET CLR Exceptions counters.

To address customers' concerns and to satisfy all the guidelines described in this chapter, Microsoft added a new method to the `Int32` class. This new method is called `TryParse`, and it has two overloads that look like the following.

```
public static Boolean TryParse(String s, out Int32 result);
public static Boolean TryParse(String s, NumberStyles styles,
   IFormatProvider, provider, out Int32 result);
```

You'll notice that these methods return a `Boolean` that indicates whether the `String` passed in contains characters that can be parsed into an `Int32`. These methods also return an output parameter named `result`. If the methods return `true`, `result` will contain the result of parsing the string into a 32-bit integer. If the methods return `false`, `result` will contain 0, but you really shouldn't execute any code that looks at it anyway.

One thing I want to make absolutely clear: A `TryXxx` method's `Boolean` return value returns `false` to indicate one and only one type of failure. The method should still throw exceptions for any other type of failure. For example, `Int32`'s `TryParse` throws an `ArgumentException` if the style's argument is not valid, and it is certainly still possible to have an `OutOfMemoryException` thrown when calling `TryParse`.

I also want to make it clear that object-oriented programming allows programmers to be productive. One way that it does this is by not exposing error codes in a type's members. In other words, constructors, methods, properties, etc. are all defined with the idea that calling them won't fail. And, if defined correctly, for most uses of a member, it will not fail, and there will be no performance hit because an exception will not be thrown.

When defining types and their members, you should define the members so that it is unlikely that they will fail for the common scenarios in which you expect your types to be used. If you later hear from users that they are dissatisfied with the performance due to exceptions being thrown, then and only then should you consider adding `TryXxx` methods. In other words, you should produce the best object model first and then, if users push back, add some `TryXxx` methods to your type so that the users who experience performance trouble can benefit. Users who are not experiencing performance trouble should continue to use the non-`TryXxx` versions of the methods because this is the better object model.

Constrained Execution Regions (CERs)

Many applications don't need to be robust and recover from any and all kinds of failures. This is true of many client applications like Notepad.exe and Calc.exe. And, of course, many of us have seen Microsoft Office applications like WinWord.exe, Excel.exe, and Outlook.exe terminate due to unhandled exceptions. Also, many server-side applications, like web servers, are stateless and are automatically restarted if they fail due to an unhandled exception. Of course some servers, like SQL Server, are all about state management and having data lost due to an unhandled exception is potentially much more disastrous.

In the CLR, we have AppDomains (discussed in Chapter 22), which contain state. When an AppDomain is unloaded, all its state is unloaded. And so, if a thread in an AppDomain experiences an unhandled exception, it is OK to unload the AppDomain (which destroys all its state) without terminating the whole process.[8]

[8] This is definitely true if the thread lives its whole life inside a single AppDomain (like in the ASP.NET and managed SQL Server stored procedure scenarios). But you might have to terminate the whole process if a thread crosses AppDomain boundaries during its lifetime.

By definition, a CER is a block of code that must be resilient to failure. Because AppDomains can be unloaded, destroying their state, CERs are typically used to manipulate any state that is shared by multiple AppDomains or processes. CERs are useful when trying to maintain state in the face of exceptions that get thrown unexpectedly. Sometimes we refer to these kinds of exceptions as *asynchronous exceptions*. For example, when calling a method, the CLR has to load an assembly, create a type object in the AppDomain's loader heap, call the type's static constructor, JIT IL into native code, and so on. Any of these operations could fail, and the CLR reports the failure by throwing an exception.

If any of these operations fail within a `catch` or `finally` block, then your error recovery or cleanup code won't execute in its entirety. Here is an example of code that exhibits the potential problem.

```
private static void Demo1() {
   try {
      Console.WriteLine("In try");
   }
   finally {
      // Type1's static constructor is implicitly called in here
      Type1.M();
   }
}

private sealed class Type1 {
   static Type1() {
      // if this throws an exception, M won't get called
      Console.WriteLine("Type1's static ctor called");
   }

   public static void M() { }
}
```

When I run the preceding code, I get the following output.

```
In try
Type1's static ctor called
```

What we want is to not even start executing the code in the preceding `try` block unless we know that the code in the associated `catch` and `finally` blocks is guaranteed (or as close as we can get to guaranteed) to execute. We can accomplish this by modifying the code as follows.

```
private static void Demo2() {
   // Force the code in the finally to be eagerly prepared
   RuntimeHelpers.PrepareConstrainedRegions();  // System.Runtime.CompilerServices namespace
   try {
      Console.WriteLine("In try");
   }
   finally {
      // Type2's static constructor is implicitly called in here
      Type2.M();
   }
}

public class Type2 {
   static Type2() {
      Console.WriteLine("Type2's static ctor called");
   }

   // Use this attribute defined in the System.Runtime.ConstrainedExecution namespace
   [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
   public static void M() { }
}
```

Now, when I run this version of the code, I get the following output.

```
Type2's static ctor called
In try
```

The `PrepareConstrainedRegions` method is a very special method. When the JIT compiler sees this method being called immediately before a `try` block, it will eagerly compile the code in the `try`'s `catch` and `finally` blocks. The JIT compiler will load any assemblies, create any type objects, invoke any static constructors, and JIT any methods. If any of these operations result in an exception, then the exception occurs *before* the thread enters the `try` block.

When the JIT compiler eagerly prepares methods, it also walks the entire call graph eagerly preparing called methods. However, the JIT compiler only prepares methods that have the `ReliabilityContractAttribute` applied to them with either `Consistency.WillNotCorruptState` or `Consistency.MayCorruptInstance` because the CLR can't make any guarantees about methods that might corrupt AppDomain or process state. Inside a `catch` or `finally` block that you are protecting with a call to `PrepareConstrainedRegions`, you want to make sure that you only call methods with the `ReliabillityContractAttribute` set as I've just described.

The `ReliabilityContractAttribute` looks like this.

```
public sealed class ReliabilityContractAttribute : Attribute {
   public ReliabilityContractAttribute(Consistency consistencyGuarantee, Cer cer);
   public Cer Cer { get; }
   public Consistency ConsistencyGuarantee { get; }
}
```

This attribute lets a developer document the reliability contract of a particular method to the method's potential callers. Both the `Cer` and `Consistency` types are enumerated types defined as follows.[9]

```
enum Consistency {
   MayCorruptProcess, MayCorruptAppDomain, MayCorruptInstance, WillNotCorruptState
}

enum Cer { None, MayFail, Success }
```

If the method you are writing promises not to corrupt any state, use `Consistency.WillNotCorruptState`. Otherwise, document what your method does by using one of the other three possible values that match whatever state your method might corrupt. If the method that you are writing

[9] You can also apply this attribute to an interface, a constructor, a structure, a class, or an assembly to affect the members inside it.

promises not to fail, use `Cer.Success`. Otherwise, use `Cer.MayFail`. Any method that does not have the `ReliabiiltyContractAttribute` applied to it is equivalent to being marked like this.

```
[ReliabilityContract(Consistency.MayCorruptProcess, Cer.None)]
```

The `Cer.None` value indicates that the method makes no CER guarantees. In other words, it wasn't written with CERs in mind; therefore, it may fail and it may or may not report that it failed. Remember that most of these settings are giving a method a way to document what it offers to potential callers so that they know what to expect. The CLR and JIT compiler do not use this information.

When you want to write a reliable method, make it small and constrain what it does. Make sure that it doesn't allocate any objects (no boxing, for example), don't call any virtual methods or interface methods, use any delegates, or use reflection because the JIT compiler can't tell what method will actually be called. However, you can manually prepare these methods by calling one of these methods defined by the `RuntimeHelpers`'s class.

```
public static void PrepareMethod(RuntimeMethodHandle method)
public static void PrepareMethod(RuntimeMethodHandle method,
   RuntimeTypeHandle[] instantiation)
public static void PrepareDelegate(Delegate d);
public static void PrepareContractedDelegate(Delegate d);
```

Note that the compiler and the CLR do nothing to verify that you've written your method to actually live up to the guarantees you document via the `ReliabiltyContractAttribute`. If you do something wrong, then state corruption is possible.

Note Even if all the methods are eagerly prepared, a method call could still result in a `StackOverflowException`. When the CLR is not being hosted, a `StackOverflowException` causes the process to terminate immediately by the CLR internally calling `Environment.FailFast`. When hosted, the `PreparedConstrainedRegions` method checks the stack to see if there is approximately 48 KB of stack space remaining. If there is limited stack space, the `StackOverflowException` occurs before entering the `try` block.

You should also look at `RuntimeHelper`'s `ExecuteCodeWithGuaranteedCleanup` method, which is another way to execute code with guaranteed cleanup.

```
public static void ExecuteCodeWithGuaranteedCleanup(TryCode code, CleanupCode backoutCode,
   Object userData);
```

When calling this method, you pass the body of the `try` and `finally` block as callback methods whose prototypes match these two delegates respectively.

```
public delegate void TryCode(Object userData);
public delegate void CleanupCode(Object userData, Boolean exceptionThrown);
```

And finally, another way to get guaranteed code execution is to use the `CriticalFinalizerObject` class, which is explained in great detail in Chapter 21.

Code Contracts

Code contracts provide a way for you to declaratively document design decisions that you've made about your code within the code itself. The contracts take the form of the following:

- **Preconditions** Typically used to validate arguments
- **Postconditions** Used to validate state when a method terminates either due to a normal return or due to throwing an exception
- **Object Invariants** Used to ensure an object's fields remain in a good state through an object's entire lifetime

Code contracts facilitate code usage, understanding, evolution, testing, documentation, and early error detection.[10] You can think of preconditions, postconditions, and object invariants as parts of a method's signature. As such, you can loosen a contract with a new version of your code, but you cannot make a contract stricter with a new version without breaking backward compatibility.

At the heart of the code contracts is the static `System.Diagnostics.Contracts.Contract` class.

```
public static class Contract {
   // Precondition methods: [Conditional("CONTRACTS_FULL")]
   public static void Requires(Boolean condition);
   public static void EndContractBlock();

   // Preconditions: Always
   public static void Requires<TException>(Boolean condition) where TException : Exception;

   // Postcondition methods: [Conditional("CONTRACTS_FULL")]
   public static void Ensures(Boolean condition);
   public static void EnsuresOnThrow<TException>(Boolean condition)
      where TException : Exception;

   // Special Postcondition methods: Always
   public static T Result<T>();
   public static T OldValue<T>(T value);
   public static T ValueAtReturn<T>(out T value);

   // Object Invariant methods: [Conditional("CONTRACTS_FULL")]
   public static void Invariant(Boolean condition);

   // Quantifier methods: Always
   public static Boolean Exists<T>(IEnumerable<T> collection, Predicate<T> predicate);
   public static Boolean Exists(Int32 fromInclusive, Int32 toExclusive,
      Predicate<Int32> predicate);
   public static Boolean ForAll<T>(IEnumerable<T> collection, Predicate<T> predicate);
   public static Boolean ForAll(Int32 fromInclusive, Int32 toExclusive,
      Predicate<Int32> predicate);
   // Helper methods: [Conditional("CONTRACTS_FULL")] or [Conditional("DEBUG")]
   public static void Assert(Boolean condition);
```

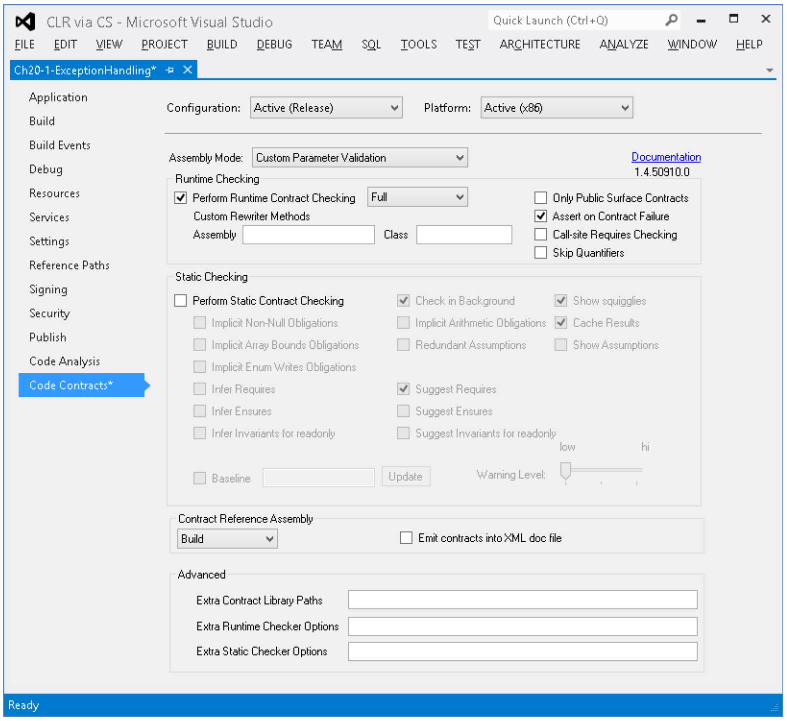
[10] To help with automated testing, see the Pex tool created by Microsoft Research: *http://research.microsoft.com/en-us/projects/pex/*.

```
   public static void Assume(Boolean condition);

   // Infrastructure event: usually your code will not use this event
   public static event EventHandler<ContractFailedEventArgs> ContractFailed;
}
```

As previously indicated, many of these static methods have the `[Conditional("CONTRACTS_FULL")]` attribute applied to them. Some of the helper methods also have the `[Conditional("DEBUG")]` attribute applied to them. This means that the compiler will ignore any code you write that calls these methods unless the appropriate symbol is defined when compiling your code. Any methods marked with "Always" mean that the compiler always emits code to call the method. Also, the `Requires`, `Requires<TException>`, `Ensures`, `EnsuresOnThrow`, `Invariant`, `Assert`, and `Assume` methods have an additional overload (not shown) that takes a `String` message argument so you can explicitly specify a string message that should appear when the contract is violated.

By default, contracts merely serve as documentation because you would not define the `CONTRACTS_FULL` symbol when you build your project. In order to get some additional value out of using contracts, you must download additional tools and a Visual Studio property pane from *http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx*. The reason why all the code contract tools are not included with Visual Studio is because this technology is being improved rapidly. The Microsoft DevLabs website can offer new versions and improvements more quickly than Visual Studio itself. After downloading and installing the additional tools, you will see your projects have a new property pane available to them, as shown in Figure 20-9.



Configuration: Active (Release)   Platform: Active (x86)

Assembly Mode: Custom Parameter Validation   1.4.50910.0

Runtime Checking
[✓] Perform Runtime Contract Checking: Full
Custom Rewriter Methods: Assembly [ ] Class [ ]
[ ] Only Public Surface Contracts
[✓] Assert on Contract Failure
[ ] Call-site Requires Checking
[ ] Skip Quantifiers

Static Checking
[ ] Perform Static Contract Checking
[✓] Check in Background
[✓] Show squiggles
[ ] Implicit Non-Null Obligations
[ ] Implicit Arithmetic Obligations
[✓] Cache Results
[ ] Implicit Array Bounds Obligations
[ ] Redundant Assumptions
[ ] Show Assumptions
[ ] Implicit Enum Writes Obligations
[ ] Infer Requires
[✓] Suggest Requires
[ ] Infer Ensures
[ ] Suggest Ensures
[ ] Infer Invariants for readonly
[ ] Suggest Invariants for readonly
[ ] Baseline
Warning Level: low

Contract Reference Assembly: Build
[ ] Emit contracts into XML doc file

Advanced
Extra Contract Library Paths
Extra Runtime Checker Options
Extra Static Checker Options

FIGURE 20-9 The Code Contracts pane for a Visual Studio project.

To turn on code contract features, select the Perform Runtime Contract Checking check box and select Full from the combo box next to it. This defines the CONTRACTS_FULL symbol when you build your project and invokes the appropriate tools (described shortly) after building your project. Now, at run time, when a contract is violated, Contract's ContractFailed event is raised. Usually, developers do not register any methods with this event, but if you do, then any methods you register will receive a ContractFailedEventArgs object that looks like this.

```
public sealed class ContractFailedEventArgs : EventArgs {
   public ContractFailedEventArgs(ContractFailureKind failureKind,
      String message, String condition, Exception originalException);

   public ContractFailureKind FailureKind       { get; }
   public String              Message           { get; }
   public String              Condition         { get; }
   public Exception           OriginalException { get; }

   public Boolean Handled { get; }   // true if any handler called SetHandled
   public void SetHandled();         // Call to ignore the violation; sets Handled to true

   public Boolean Unwind { get; }    // true if any handler called SetUnwind or threw
   public void SetUnwind();          // Call to force ContractException; set Unwind to true
}
```

Multiple event handler methods can be registered with this event. Each method can process the contract violation any way it chooses. For example, a handler can log the violation, ignore the violation (by calling SetHandled), or terminate the process. If any method calls SetHandled, then the violation will be considered handled and, after all the handler methods return, the application code is allowed to continue running unless any handler calls SetUnwind. If a handler calls SetUnwind, then, after all the handler methods have completed running, a System.Diagnostics.Contracts.ContractException is thrown. Note that this type is internal to MSCorLib.dll and therefore you cannot write a catch block to catch it explicitly. Also note that if any handler method throws an unhandled exception, then the remaining handler methods are invoked and then a ContractException is thrown.

If there are no event handlers or if none of them call SetHandled, SetUnwind, or throw an unhandled exception, then default processing of the contract violation happens next. If the CLR is being hosted, then the host is notified that a contract failed. If the CLR is running an application on a non-interactive window station (which would be the case for a Windows service application), then Environment.FailFast is called to instantly terminate the process. If you compile with the Assert On Contract Failure check box selected, then an assert dialog box will appear allowing you to connect a debugger to your application. If this option is not selected, then a ContractException is thrown.

Let's look at a sample class that is using code contracts.

```
public sealed class Item { /* ... */ }

public sealed class ShoppingCart {
   private List<Item> m_cart      = new List<Item>();
   private Decimal    m_totalCost = 0;
   public ShoppingCart() {
   }

   public void AddItem(Item item) {
      AddItemHelper(m_cart, item, ref m_totalCost);
   }

   private static void AddItemHelper(List<Item> m_cart, Item newItem,
      ref Decimal totalCost) {

      // Preconditions:
      Contract.Requires(newItem != null);
      Contract.Requires(Contract.ForAll(m_cart, s => s != newItem));

      // Postconditions:
      Contract.Ensures(Contract.Exists(m_cart, s => s == newItem));
      Contract.Ensures(totalCost >= Contract.OldValue(totalCost));
      Contract.EnsuresOnThrow<IOException>(totalCost == Contract.OldValue(totalCost));

      // Do some stuff (which could throw an IOException)...
      m_cart.Add(newItem);
      totalCost += 1.00M;
   }

   // Object invariant
   [ContractInvariantMethod]
   private void ObjectInvariant() {
      Contract.Invariant(m_totalCost >= 0);
   }
}
```

The `AddItemHelper` method defines a bunch of code contracts. The preconditions indicate that `newItem` must not be `null` and that the item being added to the cart is not already in the cart. The postconditions indicate that the new item must be in the cart and that the total cost must be at least as much as it was before the item was added to the cart. The postconditions also indicate that if `AddItemHelper` were to throw an `IOException` for some reason, then `totalCost` is unchanged from what it was when the method started to execute. The `ObjectInvariant` method is just a private method that, when called, makes sure that the object's `m_totalCost` field never contains a negative value.

Important All members referenced in a precondition, postcondition, or invariant test must be side-effect free. This is required because testing conditions should not change the state of the object itself. In addition, all members referenced in a precondition test must be at least as accessible as the method defining the precondition. This is required because callers of the method should be able to verify that they have met all the preconditions prior to invoking the method. On the other hand, members referenced in a postcondition or invariant test can have any accessibility as long as the code can compile. The reason why accessibility isn't important here is because postcondition and invariant tests do not affect the callers' ability to invoke the method correctly.

Important In regard to inheritance, a derived type cannot override and change the preconditions of a virtual member defined in a base type. Similarly, a type implementing an interface member cannot change the preconditions defined by that interface member. If a member does not have an explicit contract defined for it, then the member has an implicit contract that logically looks like this.

```
Contract.Requires(true);
```

And because a contract cannot be made stricter with new versions (without breaking compatibility), you should carefully consider preconditions when introducing a new virtual, abstract, or interface member. For postconditions and object invariants, contracts can be added and removed at will as the conditions expressed in the virtual/abstract/interface member and the conditions expressed in the overriding member are just logically AND-ed together.

So now you see how to declare contracts. Let's now talk about how they function at run time. You get to declare all your precondition and postcondition contracts at the top of your methods where they are easy to find. Of course, the precondition contracts will validate their tests when the method is invoked. However, we don't want the postcondition contracts to validate their tests until the method returns. In order to get the desired behavior, the assembly produced by the C# compiler must be processed by the Code Contract Rewriter tool (CCRewrite.exe, found in C:\Program Files (x86)\Microsoft\Contracts\Bin), which produces a modified version of the assembly. After you select the Perform Runtime Contract Checking check box for your project, Visual Studio will invoke this tool for you automatically whenever you build the project. This tool analyzes the IL in all your methods and it rewrites the IL so that any postcondition contracts are executed at the end of each method. If your method has multiple return points inside it, then the CCRewrite.exe tool modifies the method's IL code so that all return points execute the postcondition code prior to the method returning.

The CCRewrite.exe tool looks in the type for any method marked with the `[ContractInvariantMethod]` attribute. The method can have any name but, by convention, people usually name the method `ObjectInvariant` and mark the method as `private` (as I've just done). The method must accept no arguments and have a `void` return type. When the CCRewrite.exe tool sees a method marked with this attribute, it inserts IL code at the end of every `public` instance method to call the `ObjectInvariant` method. This way, the object's state is checked as each method returns to ensure that no method has violated the contract. Note that the CCRewrite.exe tool does not modify a `Finalize` method or an `IDisposable`'s `Dispose` method to call the `ObjectInvariant` method because it is OK for an object's state to be altered if it is considered to be destroyed or disposed. Also note that a single type can define multiple methods with the `[ContractInvariantMethod]` attribute; this is useful when working with partial types. The CCRewrite.exe tool will modify the IL to call all of these methods (in an undefined order) at the end of each public method.

The `Assert` and `Assume` methods are unlike the other methods. First, you should not consider them to be part of the method's signature, and you do not have to put them at the beginning of a method. At run time, these two methods perform identically: they just verify that the condition passed to them is true and throw an exception if it is not. However, there is another tool, the Code Contract Checker (CCCheck.exe), which analyzes the IL produced by the C# compiler in an attempt to statically verify that no code in the method violates a contract. This tool will attempt to prove that any condition passed to `Assert` is `true`, but it will just assume that any condition passed to `Assume` is `true` and the tool will add the expression to its body of facts known to be true. Usually, you will use `Assert` and then change an `Assert` to an `Assume` if the CCCheck.exe tool can't statically prove that the expression is true.

Let's walk through an example. Assume that I have the following type definition.

```
internal sealed class SomeType {
   private static String s_name = "Jeffrey";

   public static void ShowFirstLetter() {
      Console.WriteLine(s_name[0]);   // warning: requires unproven: index < this.Length
   }
}
```

When I build this code with the Perform Static Contract Checking function turned on, the CCCheck.exe tool produces the warning shown as a comment in the preceding code. This warning is notifying me that querying the first letter of `s_name` may fail and throw an exception because it is unproven that `s_name` *always* refers to a string consisting of at least one character.

Therefore, what we'd like to do is add an assertion to the `ShowFirstLetter` method.

```
public static void ShowFirstLetter() {
   Contract.Assert(s_name.Length >= 1);   // warning: assert unproven
   Console.WriteLine(s_name[0]);
}
```

Unfortunately, when the CCCheck.exe tool analyzes this code, it is still unable to validate that `s_name` always refers to a string containing at least one letter, so the tool produces a similar warning. Sometimes the tool is unable to validate assertions due to limitations in the tool; future versions of the tool will be able to perform a more complete analysis.

To override shortcomings in the tool or to claim that something is true that the tool would never be able to prove, we can change `Assert` to `Assume`. If we know for a fact that no other code will modify `s_name`, then we can change `ShowFirstLetter` to this.

```
public static void ShowFirstLetter() {
   Contract.Assume(s_name.Length >= 1);   // No warning at all now!
   Console.WriteLine(s_name[0]);
}
```

With this version of the code, the CCCheck.exe tool just takes our word for it and concludes that `s_name` always refers to a string containing at least one letter. This version of the `ShowFirstLetter` method passes the code contract static checker without any warnings at all.

Now, let's talk about the Code Contract Reference Assembly Generator tool (CCRefGen.exe). Running the CCRewrite.exe tool to enable contract checking helps you find bugs more quickly, but all the code emitted during contract checking makes your assembly bigger and hurts its run-time performance. To improve this situation, you use the CCRefGen.exe tool to create a separate *contract reference assembly*. Visual Studio invokes this tool for you automatically if you set the Contract Reference Assembly combo box to Build. Contract assemblies are usually named *AssemName*.Contracts.dll (for example, MSCorLib.Contracts.dll), and these assemblies contain only metadata and the IL that describes the contracts—nothing else. You can identify a contract reference assembly because it will have the `System.Diagnostics.Contracts.ContractReferenceAssemblyAttribute` applied to the assembly's assembly definition metadata table. The CCRewrite.exe tool and the CCCheck.exe tool can use contract reference assemblies as input when these tools are performing their instrumentation and analysis.

The last tool, the Code Contract Document Generator tool (CCDocGen.exe), adds contract information to the XML documentation files already produced by the C# compiler when you use the compiler's `/doc:file` switch. This XML file, enhanced by the CCDocGen.exe tool, can be processed by Microsoft's Sandcastle tool to produce MSDN-style documentation that will now include contract information.