

2

Gestione degli errori in .NET

Un errore non è solo una linea di codice scritta male. Il più delle volte per errore intendiamo una situazione non prevista e/o non gestita correttamente. Un buon componente, così come una buona applicazione, non solo è esente da bug (utopia), ma gestisce al meglio anche le situazioni più imprevedibili! Questo non sempre vuol dire risolvere il problema; a volte è sufficiente fornire una diagnostica sufficiente per consentire al programmatore o all'utente di capire o descrivere meglio cosa è successo.

Analizziamo quindi le tecniche di gestione degli errori in .NET. Ovviamente cominciamo col descrivere il ruolo e l'utilizzo delle eccezioni nel CLR, facendo poi alcune considerazioni in merito alle prestazioni: sono diffusi alcuni luoghi comuni che, pur essendo in parte fondati, portano a comportamenti errati. Cronometro alla mano, valutiamo pro e contro di varie scelte.

Vedremo poi alcuni scenari d'uso reale in funzione della libreria usata o del tipo di applicazione o componente sviluppato: componenti condivisi, ADO.NET, ASP.NET, Windows Forms, Servizi, Web Service.

Eccezioni nel CLR

Un'eccezione è una situazione anomala individuata durante l'esecuzione di un programma. L'eccezione è un evento inaspettato, altrimenti che eccezione sarebbe?

Al verificarsi di un'anomalia durante l'esecuzione del programma, l'eccezione interrompe il normale flusso di esecuzione, ricercando un punto nel codice in grado di gestire la condizione rilevata. Tale codice, se esiste, assume il controllo dell'esecuzione ricevendo un oggetto (l'eccezione) che contiene tutte le informazioni relative alla causa scatenante dell'anomalia.

Con il termine *eccezione*, dunque, si identificano sia l'oggetto che trasferisce le informazioni, sia la stessa condizione di anomalia.

Il CLR supporta le eccezioni con particolari costrutti definiti nel codice IL. Ciascun linguaggio .NET espone tali concetti con delle keyword e una sintassi parti-

Capitolo 2

colare. Gli esempi che seguono sono prevalentemente in C#, ma tutti i linguaggi .NET offrono capacità analoghe, eventualmente con alcune differenze sintattiche.

try e catch

Per intercettare un'eccezione occorre definire un blocco di codice particolare, identificato in C# dalla keyword `catch`. Un blocco `catch` distingue il tipo di eccezione in base al tipo di oggetto che contiene le informazioni sull'eccezione stessa. Un blocco `catch` intercetta tutte le eccezioni che si verificano nel blocco `try` immediatamente precedente: per scrivere uno o più blocchi `catch`, è necessario comunque definire un blocco `try`.

La sintassi è la seguente.

DIR: EXCEPTION **FILE: EXCEPT_01.CS**

```
try {
    a();
    b();
    c();
}
catch (Exception ex) {
    Console.WriteLine( ex.Message );
}
```

Il blocco `catch` contiene delle istruzioni (in questo caso la visualizzazione di una scritta) che saranno eseguite solo se all'interno delle istruzioni contenute nel blocco `try` si verificherà un'eccezione di tipo `Exception`. L'eccezione può verificarsi in una qualsiasi delle funzioni chiamate (a, b e c), a un qualsiasi livello di chiamata (anche se a chiama d che a sua volta chiama f, l'eccezione può verificarsi anche in f). In pratica, tutto il codice che viene eseguito all'interno del blocco `try`, a qualsiasi livello di chiamata, è sottoposto al "monitoraggio" da parte del `catch`.

L'eccezione può verificarsi in due modi: implicitamente, grazie al CLR, o esplicitamente, usando la keyword C# `throw`.

Esempi classici di eccezioni implicite sono la divisione per zero (`DivideByZeroException`) o l'accesso a un'istanza non valida (`NullReferenceException`).

DIR: EXCEPTION **FILE: DIVIDEBYZERO.CS**

```
int a, b;
a = 0;
try {
    Console.WriteLine( "Step 1" );
    b = 5 / a;
    Console.WriteLine( "Step 2" );
}
catch (DivideByZeroException ex) {
    Console.WriteLine( "Step 3" );
    Console.WriteLine( ex.Message );
}
Console.WriteLine( "Step 4" );
```

La linea in grassetto genera un'eccezione `DivideByZeroException`: la linea successiva nel blocco `try` non viene eseguita e il controllo passa al codice nel blocco `catch` che intercetta l'eccezione corrispondente. Nell'oggetto ricevuto è anche disponibile un messaggio descrittivo dell'errore, come si vede dall'output prodotto.

```
Step 1
Step 3
Attempted to divide by zero.
Step 4
```

Un'eccezione esplicita è generata volontariamente in presenza di condizioni anormali: un esempio tipico è la validazione dei parametri di una funzione. Nel codice che segue, l'istruzione `throw` genera la condizione di eccezione descritta dall'oggetto passato come parametro; esso è generato al volo, creando una nuova istanza della classe `ArgumentOutOfRangeException`.

DIR: EXCEPTION FILE: DIVIDEBYZERO.CS

```
static int GetDigit( char c ) {
    if ((c < '0') || (c > '9')) {
        throw new ArgumentOutOfRangeException( "c" ,
            "Argument must be between 0 and 9" );
    }
    return c - '0';
}
static void Main() {
    int digit;
    try {
        digit = GetDigit( '9' );
        Console.WriteLine( digit );
        digit = GetDigit( 'A' );
        Console.WriteLine( digit );
    }
    catch (ArgumentOutOfRangeException ex) {
        Console.WriteLine( ex.Message );
    }
}
```

Questa volta l'oggetto trasferito al blocco `catch` contiene informazioni specifiche del parametro non valido, che sono visualizzate nell'output prodotto.

```
9
Argument must be between 0 and 9
Parameter name: c
```

Come si vede, esistono diversi tipi di eccezioni, ciascuno dei quali non è altro che una classe del framework con particolari caratteristiche che analizzeremo tra poco. Un blocco `try` può essere seguito da più blocchi `catch`, ciascuno associato a un tipo di eccezione diversa.

Capitolo 2

DIR: EXCEPTION FILE: EXCEPT_02.CS

```

try {
    a();
    b();
    c();
}
catch (ArgumentException ex) {
    Console.WriteLine( "ArgumentException : argument {0}",
        ex.ParamName );
}
catch (DivideByZeroException ex) {
    Console.WriteLine( "DivideByZeroException from {0}",
        ex.TargetSite );
}
catch (Exception ex) {
    Console.WriteLine( ex.Message );
}

```

L'ordine dei blocchi `catch` non è casuale: essi sono valutati sequenzialmente e se uno corrisponde al tipo di eccezione che si è verificato, i blocchi successivi non sono più considerati. Bisogna ricordare che in .NET qualsiasi istanza di un tipo `B` derivato da `A` può essere assegnata e usata in luogo di un'istanza di `A`. Le eccezioni sono classi strutturate in una particolare gerarchia, dove tutte le classi derivano da `Exception`: la `catch` associata a `Exception` è quindi in grado di catturare qualsiasi eccezione¹, per questo motivo sarà quasi sempre l'ultima in un elenco di `catch`.

Per individuare l'ordine corretto delle `catch` è indispensabile conoscere la gerarchia delle classi di eccezione usate: i tipi di eccezione più derivati andranno specificati prima dei tipi di eccezione più generici. Il compilatore `C#` effettua questo controllo e impedisce la compilazione di codice che non rispetta questo vincolo. Il compilatore `VB.NET`, invece, non fornisce questa funzionalità: la possibilità di specificare clausole di filtro (`When`) sulla `catch`, prevista dal CLR ma non disponibile in `C#`, fa sì che in alcune circostanze abbia senso contravvenire alla regola appena citata.

finally

Con `finally` si può definire un blocco di codice la cui esecuzione è garantita rispetto all'uscita del blocco `try` o `try/catch` che lo precede, per qualsiasi motivo ciò avvenga. In altre parole, il blocco `finally` è eseguito sia quando si esce dal `try` normalmente (anche in anticipo con l'istruzione `return!`), sia quando si esce anticipatamente dal `try` a seguito del verificarsi di un'eccezione.

Tipicamente libri e documentazione sull'argomento riportano questa sintassi.

¹ In realtà il CLR prevede la possibilità di usare come eccezioni anche oggetti che non derivano da `Exception`. Oltre a non essere conforme alle specifiche CLS, questo comportamento è ottenibile in `C++` ma non in `C#` o in `VB.NET`. È consigliabile attenersi alle specifiche CLS e usare solo eccezioni derivate da `Exception`.

DIR: EXCEPTION **FILE: TRYCATCHFINALLY.CS**

```
int a, b;
a = 0;
try {
    Console.WriteLine( "try - step 1" );
    b = 5 / a;
    Console.WriteLine( "try - step 2" );
}
catch (DivideByZeroException ex) {
    Console.WriteLine( "catch DivideByZeroException" );
    Console.WriteLine( ex.Message );
}
catch (Exception ex) {
    Console.WriteLine( "catch Exception" );
    Console.WriteLine( ex.Message );
}
finally {
    Console.WriteLine( "finally - step 3" );
}
Console.WriteLine( "step 4" );
```

È alquanto improbabile che un costrutto simile sia realmente utile o efficace. L'esecuzione del codice appena visto genera questo risultato.

```
try - step 1
catch DivideByZeroException
Attempted to divide by zero.
finally - step 3
step 4
```

Nel mondo reale il ruolo del blocco `finally` è di assicurare il rilascio di lock e/o risorse acquisite per il codice del blocco `try`. Tale rilascio deve presumibilmente avvenire il prima possibile, ma l'esecuzione del blocco `catch` potrebbe ritardare sensibilmente quest'azione, in particolare se nel blocco `catch` si effettuassero operazioni di I/O o se addirittura si interagisse con l'utente.

Il più delle volte il codice che si vuole realizzare è implementato meglio da un blocco `try/finally` nidificato in un blocco `try/catch`.

DIR: EXCEPTION **FILE: TRYFINALLYCATCH.CS**

```
int a, b;
a = 0;
try {
    try {
        Console.WriteLine( "try - step 1" );
        b = 5 / a;
        Console.WriteLine( "try - step 2" );
    }
    finally {
        Console.WriteLine( "finally - step 3" );
    }
}
```

Capitolo 2

```

}
catch (DivideByZeroException ex) {
    Console.WriteLine( "catch DivideByZeroException" );
    Console.WriteLine( ex.Message );
}
catch (Exception ex) {
    Console.WriteLine( "catch Exception" );
    Console.WriteLine( ex.Message );
}
Console.WriteLine( "step 4" );

```

In questo modo l'esecuzione di `finally` anticipa l'esecuzione del blocco `catch` in caso di eccezione, mentre in caso di uscita regolare dal blocco `try` il comportamento è identico al codice `try/catch/finally` visto prima. Vediamo il risultato prodotto in caso di eccezione.

```

try - step 1
finally - step 3
catch DivideByZeroException
Attempted to divide by zero.
step 4

```

Questa disquisizione può sembrare un po' accademica, ma serve a chiarire un concetto fondamentale: l'esistenza di uno strumento non ne giustifica un uso indiscriminato. Il più delle volte una funzione ha solo uno o più blocchi `try/finally` o un blocco `try/catch`. Normalmente le funzioni di livello più alto gestiscono gli errori (con `try/catch`), mentre ai livelli più bassi il codice non ha modo di sapere come recuperare una condizione di errore, ma sicuramente deve far sì che, a prescindere da ciò che farà il chiamante, anche in caso di errore non vi siano risorse bloccate per più tempo del necessario, usando `try/finally` per garantirne il rilascio in ogni condizione. Non usare mai nei propri programmi la completa sintassi `try/catch/finally` non costituisce quindi nota di demerito, anzi!

Per comprendere meglio l'uso di `finally`, vedere anche la sezione relativa a `using` più avanti in questo capitolo, e la sezione relativa al rilascio deterministico delle risorse nel capitolo 1.

Creare nuove eccezioni

Un'eccezione è rappresentata dall'istanza di una particolare classe, che individua il tipo di eccezione. L'unico requisito che una classe deve avere perché sia usata come eccezione è che derivi più o meno direttamente da `System.Exception` (per brevità ometto nel testo il namespace `System`). Per creare un nuovo tipo di eccezione è quindi sufficiente definire una classe derivata direttamente o indirettamente da `Exception`, il cui nome termina convenzionalmente con il suffisso "Exception".

DIR: EXCEPTION

FILE: CUSTOMEXCEPTION.CS

```

public class AlreadyExistsException : Exception, ISerializable {
    private string _itemName;

```

```

public string ItemName {
    get { return _itemName; }
}

public AlreadyExistsException( string itemName )
    : this( itemName, null ) {}

public AlreadyExistsException( string itemName, Exception inner )
    : base( "Item already exists", inner ) {
    _itemName = itemName;
}

public override string Message {
    get {
        return String.Format( "Item {0} already exists", ItemName );
    }
}

protected AlreadyExistsException( SerializationInfo info,
    StreamingContext context )
    : base( info, context ) {
    _itemName = info.GetString( "ItemName" );
}

void ISerializable.GetObjectData( SerializationInfo info,
    StreamingContext context ) {
    info.AddValue( "ItemName", ItemName );
    base.GetObjectData( info, context );
}
}

```

La classe `AlreadyExistsException` trasporta il nome dell'utente duplicato: lo scopo di questa eccezione è di segnalare che l'inserimento di un elemento in un elenco non può avvenire perché l'elemento è già presente.

Notare che esistono diversi costruttori: in particolare, ve ne sono due che dovrebbero essere sempre presenti in un'eccezione. Uno è quello che prevede tra i parametri un riferimento a un'eccezione (`inner`), utile quando si trasforma un'eccezione in un tipo diverso (vedremo meglio qualche esempio più avanti, nella sezione "Componenti condivisi"). L'altro è quello usato nella serializzazione della classe per ricreare l'istanza in memoria (riceve i parametri di tipo `SerializationInfo` e `StreamingContext`): senza questo costruttore e l'implementazione di `ISerializable`, l'eccezione non può essere trasferita in una chiamata remota attraverso .NET Remoting.

Oltre a definire una diversa tipologia di errore, una classe che deriva da `Exception` può fornire informazioni specifiche dell'eccezione (in questo caso la proprietà `ItemName`) o specializzare comportamenti standard della classe `Exception` (in questo caso l'accessor alla proprietà `Message`).

Il CLR fornisce una gerarchia di classi predefinite, la cui radice è `Exception`. In tale gerarchia le classi sono organizzate in più livelli.

```

System.Exception
    System.SystemException

```

Capitolo 2

```

System.ArgumentException
    System.ArgumentNullException
    System.ArgumentOutOfRangeException
System.ArithmeticException
    System.DivideByZeroException
    System.NotFiniteNumberException
    System.OverflowException
...
System.ApplicationException
    System.Reflection.TargetException
    System.Reflection.TargetInvocationException
...
System.IO.IsolatedStorage.IsolatedStorageException
...

```

Il ruolo della gerarchia è anche quello di semplificare le istruzioni `catch`: intercettando `ArgumentException` si intercettano tutte le eccezioni derivate da tale classe (come `ArgumentNullException` e `ArgumentOutOfRangeException`). Nel codice che segue, il `catch` di `f1` intercetta tutte le eccezioni che `f2` potrebbe generare.

DIR: EXCEPTION **FILE: CUSTOMEXCEPTION.CS**

```

static void f1( int a, string s ) {
    try {
        Console.WriteLine( "f1( {0}, {1} )", a, s );
        f2( a, s );
    }
    catch (ArgumentException ex) {
        Console.WriteLine( "Argument error on {0}", ex.ParamName );
        Console.WriteLine( "Exception type: {0}", ex.GetType().Name );
    }
}
static void f2( int a, string s ) {
    if ((a < 0) || (a > 9)) {
        throw new ArgumentOutOfRangeException( "a" );
    }
    if (s == null) {
        throw new ArgumentNullException( "s" );
    }
    Console.WriteLine( "f2( {0}, {1} )", a, s );
}

```

Molta documentazione fornisce informazioni fuorvianti rispetto a questa gerarchia: `SystemException` e `ApplicationException` sono presentate come le classi base da cui derivano, rispettivamente, le eccezioni definite dal runtime e le eccezioni definite dalle applicazioni. Come evidenziato da Jeffrey Richter², Microsoft non segue completamente queste linee guida (come si vede sopra, esistono delle classi definite nel

² Applied Microsoft .NET Framework Programming, Microsoft Press, edizione in italiano di Mondadori Informatica, "Microsoft .NET Programmazione avanzata per il Framework", ISBN 88-8331-368-2

framework che, anziché derivare da `SystemException`, derivano direttamente da `Exception` o da `ApplicationException`) e la linea guida in sé non è molto sensata.

Difficilmente può avere senso intercettare tutte le eccezioni derivate da `SystemException` (mettendo sullo stesso piano `DivideByZeroException` e `StackOverflowException`) e un discorso analogo si può fare per `ApplicationException`, qualora tutte le eccezioni definite in un'applicazione derivino da tale classe. Personalmente aggiungo che il concetto stesso di `ApplicationException` è a volte difficilmente applicabile. Le eccezioni generate da un componente, magari condiviso da più applicazioni, dove andrebbero a collocarsi? Forse un terzo ramo gerarchico, basato su `LibraryException`, potrebbe darci la soluzione? Assolutamente no. Semplicemente, è sensato avere delle classi base comuni a gruppi di eccezioni omogenee. Alcune classi di eccezioni specifiche per un componente potrebbero anche derivare da una classe base comune, ma valuterei molto bene dove mettere un'eccezione che specializza un errore sugli argomenti: sicuramente dovrebbe derivare più da `ArgumentException` piuttosto che da un'eventuale `MyLibraryException`.

Dunque non bisogna sentirsi troppo in colpa se si deriva direttamente da `Exception`. Piuttosto, è molto sensato individuare, nelle classi già definite, una valida classe base, in particolare quando si sviluppano dei componenti (ma a parte il `Main`, una qualsiasi applicazione non è forse un insieme di componenti?). Soprattutto, prima di creare una nuova classe di eccezioni è opportuno verificare che non ne esista già una con le caratteristiche desiderate (al solito, le classi derivate da `ArgumentException` sono un ottimo esempio). Generare eccezioni istanziando una classe definita dal Framework non è "peccato".

Funzionalità comuni

La classe `Exception` definisce alcune proprietà che, grazie all'ereditarietà, troviamo in qualsiasi eccezione. Una di queste l'abbiamo già vista: la proprietà `Message`, che è virtuale e di sola lettura, fornisce una descrizione dell'errore presentabile all'utente (quindi andrebbe anche localizzata). Un'altra proprietà pensata per l'utente è `HelpLink`, anch'essa virtuale, che definisce un URL per la documentazione relativa all'errore.

Le altre proprietà disponibili servono invece a fornire informazioni diagnostiche destinate a un tecnico e sono già valorizzate senza bisogno di interventi da parte del programmatore: `Source` identifica il nome dell'assembly che ha generato l'eccezione, `TargetSite` identifica il metodo e `StackTrace` descrive lo stack di chiamata che ha portato a generare l'eccezione. Visualizzare il contenuto di `StackTrace` all'utente può non essere molto sensato per del codice in produzione, mentre è consigliabile salvare tale informazione in modo che possa essere analizzata a posteriori, per capire i motivi di un comportamento anomalo.

DIR: EXCEPTION **FILE: DUMPEXCEPTION.CS**

```
static void Main() {  
    try {  
        a();  
        b();  
        c();  
    }  
}
```

Capitolo 2

```

    }
    catch (Exception ex) {
        Console.WriteLine( ex.Message );
        LogException( ex );
    }
}
static void LogException( Exception ex ) {
    StreamWriter writer;
    writer = new StreamWriter( @"c:\temp\exception.log", true );
    try {
        writer.WriteLine( DateTime.Now );
        writer.WriteLine( "Exception: {0}", ex.GetType().Name );
        writer.WriteLine( "Assembly : {0}", ex.Source );
        writer.WriteLine( ex.StackTrace );
    }
    finally {
        writer.Close();
    }
}

```

Il codice dell'esempio completo produce questo risultato nel file di log:

```

27/12/2002 11.00.48
Exception: DivideByZeroException
Assembly : DumpException
  at Demo.f() in e:\dev\dotnet\console_02\console_02.cs:line 26
  at Demo.d() in e:\dev\dotnet\console_02\console_02.cs:line 18
  at Demo.a() in e:\dev\dotnet\console_02\console_02.cs:line 7
  at Demo.Main() in e:\dev\dotnet\console_02\console_02.cs:line 30

```

Scrivere del codice generico per la gestione degli errori “imprevisti” a fini diagnostici è una buona idea. In tal caso ha senso intercettare la classe base `Exception`, mentre in caso contrario questo è un comportamento da evitare: approfondiremo meglio questo aspetto parlando di eccezioni non gestite.

Sintassi di using

L'uso di `finally` va considerato obbligatorio tutte le volte che si deve rilasciare una risorsa in modo deterministico. Il Framework definisce un'interfaccia, `IDisposable`, che va implementata da tutte le classi che devono rilasciare una risorsa il più velocemente possibile rispetto a quando l'uso della risorsa effettivamente termina. Ciò può avvenire molto prima rispetto alla distruzione dell'oggetto in memoria per opera del garbage collector.

In altri termini, tutte le volte che si usa una classe che implementa `IDisposable`, bisognerebbe scrivere qualcosa di simile.

DIR: EXCEPTION FILE: FINALLY_01.CS

```

static void WriteLog( string s ) {
    StreamWriter writer;

```

```

writer = new StreamWriter( @"c:\temp\test.log", true );
try {
    writer.WriteLine( "{0} : {1}", DateTime.Now, s );
}
finally {
    ((IDisposable)writer).Dispose();
}
}

```

L'uso di `finally` garantisce il rilascio della risorsa anche in caso di eccezione; in questo caso l'uso di `Dispose` corrisponde alla chiamata al metodo `Close`, che è indispensabile per far sì che una successiva chiamata a `WriteLog` non trovi il file ancora aperto. Se nella stessa funzione si usano più oggetti che implementano `IDisposable`, è bene usare un blocco `finally` per ciascuna chiamata a `Dispose`.

DIR: EXCEPTION **FILE: FINALLY_02.CS**

```

static void WriteLog( string s ) {
    StreamWriter writer1, writer2;
    writer1 = new StreamWriter( @"c:\temp\test1.log", true );
    try {
        writer2 = new StreamWriter( @"c:\temp\test2.log", true );
        try {
            writer1.WriteLine( "{0} : {1}", DateTime.Now, s );
            writer2.WriteLine( "-> {0}", s );
        }
        finally {
            ((IDisposable)writer2).Dispose();
        }
    }
    finally {
        ((IDisposable)writer1).Dispose();
    }
}

```

C# offre una sintassi particolare, basata sulla keyword `using`, che semplifica il codice precedente.

DIR: EXCEPTION **FILE: FINALLY_03.CS**

```

static void WriteLog( string s ) {
    StreamWriter writer1, writer2;
    using (writer1 = new StreamWriter( @"c:\temp\test1.log", true )) {
        using (writer2 = new StreamWriter( @"c:\temp\test2.log", true )) {
            writer1.WriteLine( "{0} : {1}", DateTime.Now, s );
            writer2.WriteLine( "-> {0}", s );
        }
    }
}

```

Capitolo 2

Se, come in questo caso, i due oggetti su cui si chiama `Dispose` sono dello stesso tipo, si può scrivere una sola `using`; in questo caso anche la dichiarazione delle variabili avviene all'interno della `using`.

DIR: EXCEPTION FILE: FINALLY_04.CS

```
static void WriteLog( string s ) {
    using (StreamWriter
        writer1 = new StreamWriter( @"c:\temp\test1.log", true ),
        writer2 = new StreamWriter( @"c:\temp\test2.log", true ) ) {
        writer1.WriteLine( "{0} : {1}", DateTime.Now, s );
        writer2.WriteLine( "-> {0}", s );
    }
}
```

Il codice generato a livello di IL è identico a quello prodotto usando `try/finally` nell'esempio che segue. Come si può notare, nel `finally` è presente anche il test su `null`, per evitare eccezioni nel caso in cui la variabile sia stata modificata all'interno del `try`.

DIR: EXCEPTION FILE: FINALLY_05.CS

```
static void WriteLog( string s ) {
    StreamWriter writer1 = new StreamWriter( @"c:\temp\test1.log",
        true );
    try {
        StreamWriter writer2 = new StreamWriter( @"c:\temp\test2.log",
            true );
        try {
            writer1.WriteLine( "{0} : {1}", DateTime.Now, s );
            writer2.WriteLine( "-> {0}", s );
        }
        finally {
            if (writer2 != null) ((IDisposable)writer2).Dispose();
        }
    }
    finally {
        if (writer1 != null) ((IDisposable)writer1).Dispose();
    }
}
```

Condizioni di filtro (When)

Abbiamo già accennato al fatto che VB.NET supporta una caratteristica del CLR che consente di definire delle espressioni di filtro sulle eccezioni. Ogni istruzione `catch` di VB.NET può avere un'espressione booleana associata, che deve essere vera perché il blocco `catch` intercetti l'eccezione. In pratica, la condizione booleana scritta dal programmatore è valutata se l'eccezione è compatibile col tipo specificato nella `catch`: se l'espressione è falsa, la ricerca di un blocco `catch` continua come se il tipo dell'eccezione non corrispondesse.

```
Public Sub ElabData()
    Try
        '...
    Catch e As Exception When a < 0
        Console.WriteLine("Exception for a < 0")
    Catch e As Exception When b < 0
        Console.WriteLine("Exception for b < 0")
    End Try
End Sub
```

Apparentemente questo codice consente la scrittura di codice come il seguente, che a prima vista sembrerebbe irrealizzabile in C#.

DIR: EXCEPTION **FILE: CATCHWHEN.VB**

```
Sub GetData()
    Dim retry As Integer
    Dim looping As Boolean
    looping = True
    retry = 0
    While looping
        looping = False
        Try
            Console.WriteLine("Retry n. {0}", retry)
            ReadData()
        Catch ex As DatabaseBusyException When retry < 5
            retry += 1
            looping = True
        End Try
    End While
End Sub
```

In realtà, magari con qualche linea di codice in più, la medesima funzionalità è ottenibile anche in C#. La differenza è che l'eccezione va sempre catturata e, se la condizione `retry < 5` non è soddisfatta, va poi rilanciata con l'istruzione `throw` senza parametri. Costa qualcosa in più in termini di prestazioni, ma solo quando si verifica l'eccezione, quindi possiamo considerare la cosa poco rilevante.

DIR: EXCEPTION **FILE: CATCHWHEN.CS**

```
static void GetData() {
    int retry = 0;
    bool looping = true;
    while (looping) {
        looping = false;
        try {
            Console.WriteLine( "Retry n. {0}", retry );
            ReadData();
        }
        catch (DatabaseBusyException) {
            if (retry < 5) {
```

Capitolo 2

```

        retry += 1;
        looping = true;
    }
    else {
        throw;
    }
}
}
}

```

L'uso di `When` non è così indispensabile, ma l'aspetto interessante è che in questo caso `C#` è meno espressivo di `VB.NET` nei confronti di una funzionalità disponibile nel Framework. In alcuni casi più complessi, la possibilità di esaminare più `Catch` in funzione della condizione `When` consente di scrivere un solo blocco `Try` seguito da molte `Catch`; usando l'approccio alternativo visto con `C#`, per ottenere la stessa funzionalità è necessario nidificare più `try/catch` tra loro, perché una volta entrato in un blocco `catch`, il controllo passa a un blocco `try` esterno se l'eccezione è rilanciata con `throw` (vedremo meglio questa sintassi più avanti, nella sezione "Componenti condivisi"). Come al solito, la presenza di una funzionalità non ne giustifica automaticamente l'utilizzo; la sintassi di `When` può talvolta essere comoda, ma sicuramente non è mai indispensabile, dunque non andrebbe considerata una discriminante per la scelta del linguaggio.

Eccezioni non gestite

Scrivendo un metodo è facile decidere quando generare un'eccezione: se viene rilevata una condizione che non consente la normale prosecuzione dell'operazione richiesta, si usa `throw` per lanciare un'eccezione, che deve essere la più specifica per l'anomalia riscontrata e col maggior numero possibile di informazioni su essa.

Diverso è il discorso per la gestione delle eccezioni. Chi deve intercettare le eccezioni? Quali eccezioni vanno intercettate? Che cosa si deve fare dopo?

La considerazione più ovvia è che un metodo dovrebbe intercettare delle eccezioni solo se è in grado di intraprendere delle azioni volte a correggere l'errore o annullare la richiesta che ha provocato la condizione di errore. Questo significa che un metodo può e deve intercettare solo i tipi di eccezione che è in grado di gestire; dunque, tutte le volte che in un'applicazione reale vediamo una `catch` che cattura qualsiasi eccezione, è bene essere sospettosi.

```

static bool a() {
    try {
        b();
        return true;
    }
    catch (Exception) {
        return false;
    }
}

```

Si potrebbe pensare che la funzione `a` appena vista non sia altro che una chiamata a `b` che elimina qualsiasi eccezione, restituendo `true` se la chiamata è andata bene e

false se ci sono stati errori. Il problema è che l'eccezione verificata si potrebbe essere qualcosa come `OutOfMemoryException`: una situazione simile è probabile che si ripresenti a breve, soprattutto se `b` è una funzione che ha allocato oggetti ancora in vita che saturano la memoria. Nascondere la prima eccezione di questo tipo non consente di individuare la vera origine del problema, perché si cercherà quale sia il consumo anomalo di memoria in un'operazione successiva a quella che ne è la reale causa. Sarebbe stato opportuno limitarsi a intercettare le eccezioni che la chiamata a `b` potrebbe generare: la documentazione di una classe dovrebbe contenere tali informazioni. Ogni metodo delle classi del framework è documentato in questa maniera e i commenti XML in C# prevedono proprio questo caso col tag `<exception>`.

```

/// <summary>
/// Description of f
/// </summary>
/// <param name="s">First parameter</param>
/// <exception cref="System.ArgumentNullException">
/// Thrown when null parameter passed
/// </exception>
static void f( string s ) {
    if ( s == null ) {
        throw new ArgumentNullException( "s" );
    }
    Console.WriteLine( "Parameter = " + s );
}

```

In un'applicazione normalmente esistono delle eccezioni non gestite, ossia dei tipi di eccezione che, se si verificano, non andrebbero intercettati perché non ci sarebbe modo di continuare regolarmente le operazioni. Questo è il caso di eccezioni come `OutOfMemoryException`, `StackOverflowException` e `ExecutionEngineException`. In realtà normalmente in un programma vi sono molte eccezioni che, per distrazione o per veri errori del programma, non sono intercettate.

Consideriamo l'uso delle eccezioni come uno strumento diagnostico. Abbiamo già visto che ci sono molte proprietà nella classe `Exception` che forniscono informazioni diagnostiche. Sembra quindi naturale pensare che qualsiasi programma sia strutturato in questo modo.

```

static int Main( string[] args ) {
    try {
        return CheckedMain( args );
    }
    catch (Exception ex) {
        Console.WriteLine( ex.Message );
        Console.WriteLine( ex.Source );
        Console.WriteLine( ex.StackTrace );
        return 1;
    }
}

```

Questa tecnica può avere un senso per fare dei log parziali (magari all'interno di un componente), ma non è una soluzione universale. Il problema di questo approccio

Capitolo 2

cio è che in .NET qualsiasi programma è multi-thread e il codice appena visto agisce esclusivamente sulle eccezioni generate nel thread principale. Per fare un esempio, il codice contenuto all'interno di un distruttore C# (il metodo `Finalize` nel CLR) è sempre eseguito in un thread separato.

DIR: EXCEPTION FILE: FINALIZE.CS

```
class Demo {
    private InternalData _internalData;
    public Demo() {
        _internalData = null;
    }
    ~Demo() {
        _internalData.Display();
    }
    // ...
}
```

Questo codice è un esempio di pessima programmazione, perché è sbagliato chiamare il metodo di un altro oggetto all'interno di `Finalize` se tale oggetto è referenziato con un membro di istanza: se non ci sono altri riferimenti, l'oggetto `_internalData` potrebbe essere già stato "finalizzato", pur avendo un riferimento apparentemente valido in tale variabile.

L'esempio dell'eccezione in `Finalize` è puramente accademico, però a livello diagnostico è sempre importante catturare qualsiasi eccezione, perché può essere il segnale di un errore più o meno latente. Per fortuna è prevista la possibilità di intercettare, con un apposito evento (`AppDomain.UnhandledException`), tutte le eccezioni non gestite da parte di qualsiasi thread di un application domain. Oltre al thread per la chiamata dei metodi `Finalize`, esistono thread creati automaticamente per le chiamate asincrone (`BeginInvoke`) e per l'accodamento di operazioni tramite `ThreadPool`, oltre ai thread eventualmente creati direttamente istanziando la classe `Thread`.

Vediamo come risolvere il problema del codice precedente attraverso questo evento: nel codice che segue, tutte le eccezioni non gestite sono trasmesse a `LogException`, indipendentemente dal thread da cui provengono; l'associazione all'evento `UnhandledException` avviene all'inizio del `Main`.

DIR: EXCEPTION FILE: UNHANDLED.CS

```
static int Main( string[] args ) {
    AppDomain.CurrentDomain.UnhandledException +=
        new UnhandledExceptionEventHandler(LogException);
    return CheckedMain( args );
}

private static void LogException(
    object sender, UnhandledExceptionEventArgs e) {
    if (e.ExceptionObject is Exception) {
        Exception ex = (Exception) e.ExceptionObject;
        Console.WriteLine( ex.Message );
        Console.WriteLine( ex.Source );
        Console.WriteLine( ex.StackTrace );
    }
}
```

```
}
else {
    Console.WriteLine("Unhandled exception {0}",
        e.ExceptionObject.GetType().Name);
}
}
```

Oltre a risolvere il problema dei thread secondari, questo meccanismo consente anche di definire il metodo da agganciare all'evento in un punto qualsiasi del programma, nonché di agganciare più metodi allo stesso evento. Lo scopo, è evidente, non è quello di gestire gli errori, quanto piuttosto di produrne dei log o delle segnalazioni al mondo esterno.

Grazie a questa tecnica è possibile scrivere dei componenti che registrano automaticamente un metodo che intercetta tutti gli errori dell'applicazione, per inviarli via e-mail all'autore del componente insieme a tutte le informazioni salvate nelle proprietà dell'eccezione intercettata (privacy permettendo, ovviamente!).

Prestazioni

Scrivere un blocco `try/catch` o `try/finally` ha sicuramente dei costi in termini prestazionali. Tale penalizzazione esiste, ma è sicuramente più leggera di quella legata alla gestione delle eccezioni standard del C++: il concetto di eccezione cablato nell'ambiente di esecuzione (il CLR) è indubbiamente un enorme vantaggio da questo punto di vista.

Purtroppo si sono diffusi dei luoghi comuni secondo cui l'uso delle eccezioni è addirittura sconsigliato a causa del rallentamento che provoca. Queste considerazioni si basano in particolare sui costi derivati dall'uso di `try/catch` nei casi in cui si verifica un errore. Non mi stancherò mai di dirlo: un'eccezione è un evento eccezionale, che non si deve verificare il 100% delle volte in cui il codice transita per un `try/catch`! Cerchiamo di fare un po' di chiarezza e di capire quali sono quei casi (pochi) in cui ha senso pensare di rinunciare alle eccezioni³.

I numeri che vedremo vanno considerati unicamente dal punto di vista delle differenze relative. Non c'è nessuna pretesa di fare dei benchmark assoluti in termini di prestazioni, di conseguenza non è significativo citare le caratteristiche dell'hardware su cui sono stati effettuati i test. Su ogni macchina le misure numeriche potranno essere diverse, ma le differenze percentuali si discosteranno di poco, quindi le considerazioni finali restano sempre valide.

Costi di `try/finally`

Prima di tutto una premessa: non ci sono grosse alternative all'uso di `finally`: grazie a questo costrutto il codice è più sicuro e affidabile. Valutarne i costi è un esercizio che non deve avere un impatto nel modo in cui si affronta la programmazione: `finally` va usato tutte le volte che è semanticamente corretto usarlo.

³ Sul sito DevLeap (www.devleap.it) ho pubblicato una serie di articoli sulle eccezioni destinati in particolare agli sviluppatori VB6/VB.NET. Il contenuto di tali articoli è complementare a questo capitolo; in particolare, per quanto riguarda l'analisi delle prestazioni, è raffrontato il costo delle eccezioni VB.NET al costo dell'uso di `On Error Goto` in VB6 e in VB.NET.

Capitolo 2

Per comprendere quanto pesa usare `try/finally`, valutiamo i tempi di esecuzione di una funzione che chiama `Dispose` nel caso in cui non abbia `try/finally` (`TestNoFinallyDispose`) e nel caso in cui l'abbia (`TestFinallyDispose`).

DIR: EXCEPTION FILE: BENCHMARKFINALLY.CS

```
static void TestNoFinallyDispose() {
    Component c = new Component();
    c.ToString();
    c.Dispose();
}

static void TestFinallyDispose() {
    Component c = new Component();
    try {
        c.ToString();
    }
    finally {
        c.Dispose();
    }
}
```

Effettuando 5.000.000 di chiamate per ogni versione, otteniamo i seguenti risultati.

```
try/finally - Dispose
TestFinally   : 7667,772 msec
TestNoFinally : 7476,572 msec
```

La penalizzazione usando `finally` è di circa il 2,5%, considerando questo già come caso peggiore: probabilmente il codice all'interno dei blocchi `try/finally` è più pesante di questo, che agisce su un oggetto (`Component`) che non fa praticamente nulla. Nel caso dell'uso di `finally` per una chiamata a `Dispose` il costo di `finally` è un onere minore (il vero costo è normalmente dato da `Dispose`); quando invece `finally` è usato per garantire il rilascio di un oggetto di sincronizzazione, il peso percentuale è maggiore.

DIR: EXCEPTION FILE: BENCHMARKFINALLY.CS

```
static void TestNoFinallyMonitor() {
    Monitor.Enter( demo );
    demo.counter++;
    Monitor.Exit( demo );
}

static void TestFinallyMonitor() {
    Monitor.Enter( demo );
    try {
        demo.counter++;
    }
}
```

```

    finally {
        Monitor.Exit( demo );
    }
}

```

Usando nuovamente 5.000.000 di chiamate per ogni versione, vediamo risultati diversi: le funzioni chiamate sono meno “pesanti”, quindi l’incidenza percentuale di try/finally aumenta.

```

try/finally - Monitor
TestFinally   : 803,0018 msec
TestNoFinally : 722,8665 msec

```

Questa volta la differenza è intorno a un 11%. Il problema però non va visto tanto sull’uso di `finally`, quanto sull’uso di un meccanismo di sincronizzazione: l’accesso concorrente a una risorsa più veloce è quello totalmente privo di lock, ma siccome vogliamo preservare l’integrità dei dati a discapito delle prestazioni, usiamo delle tecniche di semaforizzazione. Di conseguenza, anche l’uso di `finally` va considerato obbligatorio, visto che si dà più importanza all’affidabilità.

Costi di try/catch

I costi legati all’uso di `try/catch` vanno considerati in due casi: quando ci sono eccezioni e quando non ve ne sono. Nel primo caso la `catch` viene valutata (per verificare il tipo) ed eventualmente eseguita, nel secondo caso no. Il codice che analizziamo è un esempio didattico un po’ estremo: pochissimo codice con un peso percentuale di `try/catch` che si configura come ipotesi peggiore (nel mondo reale dovrebbe esserci molto più codice da eseguire nel blocco `try`).

DIR: EXCEPTION **FILE: BENCHMARKCATCH.CS**

```

static int BenchmarkCatch( int a, int b ) {
    try {
        return Div( a, b );
    }
    catch (Exception) {
        return 0;
    }
}

static int Div( int a, int b ) {
    return a / b;
}

```

Confrontiamo le prestazioni di `BenchmarkCatch` con una funzione equivalente (`BenchmarkNoCatch`) ma costruita trasferendo l’eventuale errore con un valore di ritorno anziché con un’eccezione.

Capitolo 2

DIR: EXCEPTION FILE: BENCHMARKCATCH.CS

```

static int BenchmarkNoCatch( int a, int b ) {
    int c;
    if (!Div( a, b, out c )) c = 0;
    return c;
}

static bool Div( int a, int b, out int c ) {
    if (b == 0) {
        c = 0;
        return false;
    }
    c = a / b;
    return true;
}

```

Partiamo dallo scenario senza errori: chiamiamo 500.000 volte la funzione `BenchmarkXXX` passando dei valori che non fanno fallire la divisione.

```

Benchmark a = 2, b = 1 - count = 500000
BenchmarkNoCatch   : 40,12418 msec
BenchmarkCatch     : 39,9774 msec

```

La versione basata sulle eccezioni risulta addirittura più veloce di quella senza eccezioni, anche se di pochissimo⁴: il motivo è che `Div` ha un parametro in più per segnalare eventuali errori. Spesso si sottovaluta il costo del passaggio di un parametro sullo stack, ma per fare un confronto serio bisogna immaginare di avere una situazione funzionalmente equivalente. Le differenze sono minime e in parte trascurabili, visto che un parametro in più o in meno può già cambiare i risultati.

Vediamo ora lo scenario in presenza di errori. Assumiamo di avere un errore per ogni chiamata, passando 0 per il valore usato come denominatore nella divisione: non è verosimile, ma è il caso teoricamente peggiore. La chiamata viene ripetuta sempre 500.000 volte per ogni versione.

```

Benchmark a = 2, b = 0 - count = 500000
BenchmarkNoCatch   : 24,03108 msec
BenchmarkCatch     : 16821,3 msec

```

Il costo delle eccezioni diventa rilevante, soprattutto se paragonato alla situazione che otteniamo usando una tecnica di gestione degli errori basata sul valore restituito da una funzione anziché sulle eccezioni. Un dato eclatante è che questa volta il codice senza `catch` chiamato in condizioni di errore è addirittura più veloce di un'esecuzione regolare (quasi il doppio): individuando preventivamente l'errore si eseguono alcune operazioni in meno, quindi la cosa non è poi così strana.

⁴ La differenza è più evidente compilando in Debug, mentre compilando in Release (come nelle prove descritte) i valori sono quasi identici; ripetendo la prova capita che qualche volta sia più veloce la versione standard, ma sempre con differenze poco rilevanti.

Veniamo invece al codice che usa il blocco `catch`: in presenza di un'eccezione, il costo di gestione della stessa è rilevante. Attenzione, però: questo prezzo lo si paga solo quando si verifica un'eccezione, e per l'ennesima volta ripeterò che un'eccezione non dovrebbe essere la regola!

Facendo un paragone sui tempi rilevati, possiamo dire che la gestione di un'eccezione (nel caso che abbiamo visto) è 420 volte più lenta del caso in cui non ci sono errori⁵. In termini assoluti, però, bisogna ricordare che ogni eccezione ha richiesto 33 microsecondi, il che significa che in un secondo potremmo gestire quasi 30.000 eccezioni. Un po' tante per chiamarsi così!

Tornando a ragionare in termini relativi e basandoci sui numeri raccolti finora, vediamo nella tabella 2.1 l'incidenza percentuale della gestione delle eccezioni nel codice che abbiamo usato, ipotizzando vari scenari di frequenza dell'errore.

Tabella 2.1 – Incidenza percentuale della gestione delle eccezioni

	Penalizzazione %
1 errore su 1 (100%)	42.000,00%
1 errore su 10 (10%)	4.200,00%
1 errore su 100 (1%)	420,00%
1 errore su 1.000 (0,1%)	42,00%
1 errore su 10.000 (0,01%)	4,20%

Già considerando questo scenario irrealistico, è evidente che più un'eccezione è veramente tale, minore è il costo di elaborazione necessario a sostenerla. Semplicemente aggiungendo una minima elaborazione che aumenti i tempi della chiamata a `Div`, si riduce il fattore iniziale (ricordo che siamo partiti da 420) e proporzionalmente si riducono le percentuali di penalizzazione viste nella tabella 2.1.

In uno scenario reale un'eccezione non è un evento che si verifica il 100% delle volte. Nonostante la tabella precedente, è difficile stabilire una percentuale entro cui l'eccezione può essere considerata tale: in alcuni casi è una volta su dieci, in altri una volta su un milione. Inoltre, l'eccezione semplifica enormemente il codice e trasferisce informazioni più dettagliate sul tipo di errore che si è verificato.

Perciò, quando rinunciare alle eccezioni per gestire gli errori? Possiamo rispondere solo caso per caso, senza generalizzare troppo. Se il punto in cui si può individuare la condizione di errore è contestuale (o molto vicino) al punto in cui siamo in grado di gestire e risolvere la condizione di errore e se tale evento non è in realtà una condizione rarissima, ma qualcosa che si verifica piuttosto di frequente, magari all'interno di un loop, allora il costo implementativo per rinunciare alle eccezioni è qualcosa di accettabile (talvolta si scrive addirittura meno codice!).

Se, viceversa, la gestione dell'errore è molto distante dal punto in cui l'errore è rilevato, dobbiamo valutare il costo di trasferimento delle informazioni sull'errore da un livello di chiamata all'altro. Questo richiede linee di codice, disciplina da parte del programmatore e, se il numero di livelli di chiamata è alto, anche un certo overhead

⁵ È utile ricordare che questo è uno dei casi peggiori; nel mondo reale, se il codice eseguito nel try è più corposo, l'incidenza percentuale sarà sicuramente minore, anche in modo significativo.

Capitolo 2

nel passaggio di tali valori da una funzione all'altra (passaggio che avviene sempre, anche quando l'errore non si verifica). In questi casi quasi sicuramente le eccezioni sono la strada migliore da seguire.

Componenti condivisi

L'uso delle eccezioni assume prospettive diverse a seconda del tipo di codice che si sta scrivendo. Chi scrive lo strato finale di un'applicazione, in genere l'interfaccia utente, si preoccupa di intercettare le eccezioni provenienti dai componenti sottostanti per decidere (magari chiedendolo all'utente) come procedere; le informazioni relative a eccezioni non conosciute sono eventualmente archiviate per un'analisi successiva.

L'approccio di chi scrive il codice di un componente è invece diverso: l'obiettivo, in genere, non è quello di intercettare e risolvere le eccezioni, quanto di generare delle eccezioni contenenti la maggior quantità di informazioni possibile sull'errore individuato. Questa sezione si intitola "Componenti condivisi" perché la maggiore responsabilità ricade su chi crea componenti che saranno usati in applicazioni diverse da sviluppatori diversi. Tali componenti dovranno avere un comportamento prevedibile, fornire eccezioni specifiche per gli errori e allo stesso tempo non dovranno creare ulteriori problemi (oggetti non chiusi tempestivamente, lock "dimenticati" e così via) in caso di eccezioni che, gestite ai livelli superiori, non provochino l'uscita dall'intera applicazione.

Cose da evitare

Partiamo con gli avvertimenti su ciò che bisogna evitare di fare: prima di sapere quali sono le cose giuste, è meglio conoscere quelle sbagliate.

Evitare richieste all'utente

Un componente, normalmente, non interagisce con l'utente. Attenzione, non vuol dire che un componente non debba avere interfaccia utente: si può tranquillamente definire un componente per la visualizzazione di un orologio, ma questo non significa che il componente debba interagire con l'utente per modificare l'ora corrente. Tutte le attività andranno coordinate dall'applicazione che usa il componente, che dovrà limitarsi a mettere a disposizione tutti gli strumenti necessari, sotto forma di proprietà, metodi ed eventi. Il componente che deve chiedere qualcosa all'utente lo fa con un evento: il programmatore che lo usa decide come e se rispondere a tale richiesta.

Che impatto ha questo sulla gestione delle eccezioni? Semplicemente, è impensabile che, all'interno di una `catch`, si visualizzi un messaggio all'utente chiedendo come vuole procedere. In generale, dentro la `catch` in un componente si dovrebbero fare poche cose, come effettuare un'operazione compensativa per annullare un'operazione non completata correttamente a causa di un'eccezione (come il classico `rollback` di una transazione). C'è anche un altro aspetto da considerare: se nella `catch` si verifica una nuova eccezione (se si richiama un evento esterno non si sa a priori che codice sarà eseguito), questa interrompe l'operazione in corso e "nasconde" l'eccezione che si stava gestendo⁶.

⁶ Per questo è importante il `finally`: anche in un caso del genere l'esecuzione del blocco `finally` sarebbe garantita.

Evitare di nascondere gli errori

Un componente non deve mai catturare un'eccezione e "nasconderla" se non sa esattamente di cosa si tratta. In altre parole, è spiacevole scoprire in un componente del codice come questo.

```
static int BenchmarkCatch( int a, int b ) {
    try {
        return Div( a, b );
    }
    catch (Exception) {
        return 0;
    }
}
```

Sì, lo so, è esattamente la stessa funzione che ho scritto qualche pagina prima. Non volevo correre il rischio che qualcuno pensasse che si trattasse di una buona idea⁷. Il motivo credo di averlo già illustrato, ma non fa male ribadirlo: intercettare indiscriminatamente tutte le eccezioni significa intercettare anche ciò che non si è in grado di recuperare. In questo senso si "nasconde" l'eccezione al chiamante: è una pessima idea, perché in talune circostanze farà perdere ore e ore al povero programmatore che userà il nostro componente.

Evitare errori generici

Come vedremo tra poco, un componente dovrebbe preoccuparsi di generare delle eccezioni più che di gestirle. Chiaro, non lo deve fare a sproposito ma solo quando c'è un valido motivo: bene, il motivo dell'eccezione è proprio la prima informazione che bisogna trasferire.

Siccome il tipo dell'eccezione è l'informazione più significativa che viene trasmessa, è importante scegliere una classe che esponga al meglio il tipo di errore verificatosi. Se non esiste una classe sufficientemente descrittiva, sarà bene crearne una.

In pratica, il codice che segue è veramente deprecabile.

```
static int Div( int a, int b ) {
    if ( b == 0 ) {
        throw new Exception( "b can't be zero" );
    }
    return a / b;
}
```

In un caso simile c'è un'eccezione del CLR che fa al caso nostro.

⁷ Prima che qualcuno mi scriva una mail: sì, era più furbo scrivere `DivisionByZeroException`, ma d'altra parte si trattava di un benchmark (e non di un componente) e l'obiettivo dell'esempio era un altro. Altrimenti dovremmo anche discutere del fatto che qualcuno debba scrivere sei linee di codice al posto di `return (b == 0) ? 0 : a / b;`

Capitolo 2

```
static int Div( int a, int b ) {  
    if ( b == 0 ) {  
        throw new ArgumentOutOfRangeException( "b", "b can't be zero" );  
    }  
    return a / b;  
}
```

Con un'eccezione più specifica il chiamante di `Div` può intercettare l'errore in maniera più efficace, altrimenti sarebbe costretto a intercettare `Exception`, ricadendo nel comportamento da evitare che abbiamo visto poco prima.

Un piccolo chiarimento ai lettori: se in alcune parti di questo capitolo o di questo libro troverete `throw new Exception() O catch (Exception)`, sappiate che esiste una speciale licenza, rilasciata unicamente a chi scrive libri e articoli, che consente di usare tali pericolosi costrutti. Questa licenza è limitata a essere utilizzata in codice esemplificativo che non verrà mai messo in produzione. Gli stessi detentori della licenza non possono farne uso in circostanze diverse, pena l'immediata revoca della licenza stessa più eventuali pene accessorie. Spero di essere stato abbastanza chiaro. Anche se c'è scritto nel libro, non significa che possiate farlo.

Cose da fare

Ci sono cose da evitare, ma questo non significa che non fare nulla sia meglio. Un buon componente deve generare le eccezioni giuste, manipolarne alcune di quelle che riceve e soprattutto deve usare abbondantemente il costrutto `try/finally`.

Creare eccezioni specifiche

La controparte dell'evitare la generazione di errori generici è la creazione di eccezioni specifiche dell'evento rilevato. A costo di ripetermi, questo non significa creare una classe di eccezioni per ogni nuovo errore. Una buona idea è quella di studiare la gerarchie di eccezioni già definite nel Framework: alcune di queste sono facilmente riutilizzabili nel proprio codice (tra tutte citiamo `ArgumentException` e classi derivate). In ogni caso questo studio fa comprendere quali caratteristiche debba avere una classe di eccezione: abbastanza specifica per un certo problema ma anche abbastanza generica da poter essere utilizzata (con opportuni parametri/proprietà) in situazioni diverse.

Un buon esempio è, come al solito, `ArgumentNullException`: il problema è abbastanza chiaro dal nome della classe (c'è un parametro di un metodo che vale `null` e non dovrebbe), la proprietà `ParamName` consente di sapere qual è il nome di tale argomento. Sufficientemente specifica, ma anche sufficientemente generica. Non tutte le classi di eccezioni possono soddisfare questi requisiti, ma rifletterci un attimo prima di creare una classe è sempre una buona idea.

Un'altra buona mossa è quella di pensare a come impostare una gerarchia di classi. Quando più eccezioni sono simili ma ciascuna possiede caratteristiche specifiche diverse, ha senso definire una classe base: può servire a condividere proprietà e metodi in comune, ma serve anche a scrivere più agevolmente il codice di gestione delle eccezioni. Anche in questo caso, l'esempio di `ArgumentException` è illuminante: la classe generalizza tutti i problemi relativi a errori nei parametri di un metodo. Quando è possibile, l'errore si materializza in un'eccezione particolare che deriva da questa classe (come `ArgumentNullException` o `ArgumentOutOfRangeException`), ma il codice che intercetta l'errore può anche affidarsi unicamente alla classe base, più generica.

Attenzione, però, a non sforzarsi di pensare in termini di gerarchia mastodontica e monolitica, dove tutte le classi devono necessariamente derivare da una sola classe base (come potrebbe essere `ApplicationException` o un'ipotetica `LibraryException`). Ho già spiegato quale sia l'errore pratico di questo approccio; il punto è che, come al solito, bisogna evitare gli estremi. Voler avere a tutti i costi una classe base unica non serve (c'è già `Exception`); allo stesso tempo, è probabilmente sbagliato frammentare le eccezioni in modo che derivino tutte da `Exception` senza un minimo di strutturazione. Bisogna trovare una giusta via di mezzo. Il Framework (considerando le eccezioni derivate da `SystemException`) offre un discreto esempio in questo senso.

Un ultimo aspetto riguarda il deployment. È ragionevole pensare che tutte le eccezioni generate dai componenti di un assembly, e definite appositamente per tale componente, siano definite all'interno dello stesso assembly. Se si usano eccezioni definite altrove, è accettabile che siano definite in un qualsiasi assembly del Framework, oppure negli assembly che contengono classi referenziate come membri o classi base delle proprie. In altre parole, è bene che le classi di eccezioni usate facciano parte di un assembly già referenziato per altri motivi. Suscita invece perplessità definire una dipendenza rispetto a un assembly solo per usare un'eccezione definita all'interno. È una cosa che può funzionare perfettamente, ma potrebbe essere indice di una progettazione poco accurata della gerarchia di classi o di un utilizzo un po' "forzato" di una classe di eccezione già esistente.

Attenzione ai costruttori

Una classe derivata da `Exception` dovrebbe sempre avere due costruttori particolari: uno usato per la (de)serializzazione dell'eccezione, l'altro per impostare il valore di `InnerException`. La proprietà `InnerException` è usata per definire una lista di eccezioni, qualora un'eccezione sia trasformata in un'eccezione diversa da un componente (vedremo meglio questa tecnica nella sezione successiva, "Trasformare le eccezioni"). Riprendiamo il codice di un esempio precedente (la classe riportata in questo caso non è completa).

DIR: EXCEPTION **FILE: CUSTOMEXCEPTION.CS**

```
public class AlreadyExistsException : Exception, ISerializable {

    protected AlreadyExistsException( SerializationInfo info,
                                     StreamingContext context )
    {
        : base( info, context ) {
        _itemName = info.GetString( "ItemName" );
    }

    public AlreadyExistsException( string itemName, Exception inner )
    {
        : base( "Item already exists", inner ) {
        _itemName = itemName;
    }
}
```

Il costruttore necessario alla serializzazione è quello che riceve due parametri, rispettivamente di tipo `SerializationInfo` e `StreamingContext`. Tale costruttore viene usato dal runtime per deserializzare una classe ed è indispensabile se l'eccezione implementa `ISerializable`: sarebbe opportuno implementare sempre tale interfaccia

Capitolo 2

perché, come abbiamo già visto in “Creare nuove eccezioni”, rende l’eccezione trasferibile anche su chiamate remote.

Il costruttore per l’impostazione di `InnerException` può in realtà non essere uno solo: l’importante è che ci sia almeno un costruttore, magari quello con più parametri, che accetti un riferimento a un oggetto `Exception` che verrà salvato e reso disponibile con la proprietà `InnerException`. Essa è definita in `Exception` come proprietà a sola lettura e non è virtuale, quindi senza un costruttore con questo parametro non è possibile valorizzare tale proprietà.

Trasformare le eccezioni

Un componente deve essere in grado di rilevare le condizioni anomale trasformandole in eccezioni, ma come si raggiunge questo scopo? Essenzialmente in due modi: da una parte agendo preventivamente (per esempio controllando il valore di una variabile prima di usarla), dall’altra intercettando alcune eccezioni. Sì, in alcuni casi un componente intercetta alcune eccezioni, ma ho detto alcune e non tutte.

DIR: EXCEPTION

FILE: TRANSFORM.CS

```
static int Calc( int a, int b, int c ) {
    try {
        return checked(a * b / c);
    }
    catch (OverflowException ex) {
        throw new ArgumentException( "Value too large", "a or b", ex );
    }
    catch (DivideByZeroException) {
        throw new ArgumentOutOfRangeException( "c can't be zero", "c" );
    }
}
```

In questo esempio è ragionevole pensare che le eccezioni intercettate (`OverflowException` e `DivideByZeroException`) siano sempre causate da problemi nei parametri: dunque le `catch` intercettano tali eccezioni, trasformandole in una più significativa `ArgumentException` o derivata.

Nel caso di `OverflowException`, l’eccezione `ArgumentException` creata riceve un riferimento all’eccezione originaria: tale riferimento è successivamente leggibile tramite la proprietà `InnerException`. Chi riceve la nuova eccezione può quindi scorrere una lista concatenata di eccezioni, usando `InnerException` come link all’elemento successivo: la lista termina in corrispondenza della prima eccezione che ha dato origine alla catena. Notare che nel caso di `ArgumentOutOfRangeException` non è disponibile un costruttore che riceva il riferimento da usare per `InnerException`. Questo viola una delle linee guida appena citate (costruttori indispensabili), ed è un problema specifico di questa classe che non è stato corretto nella versione 1.1 del Framework: anche il Framework non è perfetto.

Usare finally

Un componente dovrebbe avere molti blocchi `try/finally`. Esistono due condizioni tipiche in cui va usato `finally`, tanto che C# offre due keyword per semplificarne l’utilizzo.

- Quando un oggetto è allocato e usato all'interno di una funzione ed è importante il rilascio deterministico delle risorse, chiamando un metodo apposito dell'oggetto. Le classi che richiedono questo comportamento da parte di chi le utilizza dovrebbero implementare l'interfaccia `IDisposable`. C# ha l'istruzione `using` per semplificare la chiamata di `IDisposable.Dispose`.
- Quando si "blocca" una risorsa per evitare un accesso concorrente. Il concetto è molto generico e copre le tecniche di sincronizzazione per la programmazione multi-thread così come l'accesso a una risorsa logica o fisica condivisa anche da più processi (come un file o un record in un database). C# mette a disposizione l'istruzione `lock` per semplificare l'uso della classe `Monitor`, usata per sincronizzare l'accesso a una risorsa condivisa all'interno di un application domain. Per l'uso di altre classi (`Mutex`, `ReaderWriterLock` e così via) è necessario specificare manualmente `try/finally` anche in C#.

È importante che un componente usi `finally`, perché chi usa il componente potrebbe intercettare un'eccezione e proseguire. In tal caso, il fatto di ritardare il rilascio di un lock o di una risorsa limitata può pregiudicare la stabilità dell'intera applicazione che fa uso del componente. Questo tipo di errore è molto difficile da individuare. Una buona disciplina nell'uso di `try/finally` è la migliore prevenzione.

Ripristinare uno stato consistente in caso di errore

A volte l'uso di `finally` non è adeguato per garantire una situazione "stabile" all'uscita da una funzione. Il comportamento nell'eventualità di un'eccezione potrebbe essere diverso da quello da seguire in condizioni normali. In questo caso ha senso intercettare l'eccezione per effettuare l'operazione di ripristino, rigenerando, immediatamente dopo, la stessa eccezione.

Immaginiamo di creare un file in cui salvare dei dati: se qualcosa durante il salvataggio va storto, il file deve essere cancellato. Nell'esempio che segue usiamo la sintassi basata sull'istruzione `throw` senza parametri, che è utilizzabile solo all'interno di un blocco `catch` e "rilancia" l'eccezione intercettata senza crearne una nuova e senza modificare il contenuto di `StackTrace`.

DIR: EXCEPTION FILE: CATCHALL.CS

```
public void SaveData() {
    try {
        using( StreamWriter writer = new StreamWriter( Filename, true ) ) {
            WriteInfo( writer );
        }
    }
    catch {
        if (File.Exists( Filename )) {
            File.Delete( Filename );
        }
        throw;
    }
}
```

Capitolo 2

Eseguendo l'esempio completo, la funzione `WriteInfo` genera un'eccezione a metà del file. La `catch`, quando viene richiamata, non deve preoccuparsi di chiudere il file; la `using` dentro il blocco `try/catch` corrisponde a un blocco `try/finally`: come abbiamo già detto, l'uso di `try/finally` dentro un `try/catch` è un pattern di utilizzo più comune rispetto a `try/catch/finally`.

L'unica preoccupazione è quella di verificare che il file esista già, per evitare un'eccezione da parte di `File.Delete`: se il nome in `Filename` non fosse valido, infatti, la chiamata al costruttore di `StreamWriter` genererebbe un'eccezione senza creare il file. Generare un'eccezione dentro il blocco `catch` sarebbe dannoso, perché nasconderebbe involontariamente l'eccezione intercettata al chiamante.

Un aspetto interessante è la sintassi `catch` usata. È naturale aspettarsi di vedere qualcosa di simile al codice che segue.

```
catch (Exception) {
    if (File.Exists( Filename )) {
        File.Delete( Filename );
    }
    throw;
}
```

Si potrebbe pensare che la sintassi usata prima (dove non compare il tipo dell'eccezione da catturare) sia equivalente, ma non è così. La differenza è che in questo caso si intercettano tutte le eccezioni corrispondenti a `Exception` o derivate. La sintassi dell'esempio precedente, con `catch` senza un tipo di eccezione specificato, intercetta invece un'eccezione di qualsiasi tipo. Come abbiamo detto, con `C#` (e in generale per aderire a CLS) tutte le eccezioni devono derivare da `Exception`. Ma se `WriteFile` avesse fatto uso di un componente scritto in Managed C++? Che magari usa eccezioni non CLS⁸? Per stare tranquilli, se lo scopo non è intercettare l'eccezione e risolverla ma semplicemente intercettarla per rilanciarla dopo aver eseguito un'operazione compensativa, allora meglio usare `catch {` invece di `catch (Exception) {`.

Un ultimo dettaglio. Proprio perché usando `catch {` non si ha accesso all'oggetto che rappresenta l'eccezione, si è più sicuri di evitare la scrittura involontaria di qualcosa di simile.

```
catch (Exception ex) {
    if (File.Exists( Filename )) {
        File.Delete( Filename );
    }
    throw ex;
}
```

A prima vista sembra tutto uguale alla variante precedente, ma questa volta abbiamo usato `throw ex;` invece di `throw;`: il codice chiamante troverà come origine dell'eccezione in `StackTrace` questo punto del programma e non quello originale! In pratica, in questo modo si perde tutta l'informazione di `StackTrace` per risalire alla linea

⁸ Non sto consigliando di creare componenti C++ per aggirare i limiti di `C#` e contravvenire alle Common Language Specification (CLS). Però, se usate dei componenti che non avete scritto voi non potete essere sicuri di ciò che fanno, quindi è meglio essere previdenti.

effettiva in cui è avvenuta l'eccezione. Notare che, applicando questa variante all'esempio completo, il chiamante si ritrova un'eccezione `DivisionByZeroException` in una linea di codice dove non c'è alcuna divisione. Ormai dovrebbe essere chiaro perché sto stressando tanto su questo concetto: non mi piace passare ore a cercare un errore nel posto sbagliato.

Internazionalizzazione

Le classi derivate da `Exception` ereditano la proprietà `Message`, che contiene tipicamente un messaggio informativo sull'eccezione, che può essere visualizzato all'utente. Se l'applicazione o il componente che si sviluppa va localizzato (cioè tradotto in lingue diverse), è una buona idea fare due cose.

1. Definire una funzione statica per istanziare l'oggetto derivato da `Exception`, centralizzando l'assegnazione del parametro corrispondente a `Message`.
2. Usare una risorsa per il testo del messaggio, semplificando la fase di localizzazione.

In pratica, anziché avere

```
throw new ArgumentOutOfRangeException( "month",
    "Argument must be between 1 and 12" );
```

si userà il codice che segue.

```
throw new ExceptionBuilder.ArgumentOutOfRange( "month", "1", "12" );
```

Nell'esempio usiamo una classe che racchiude la generazione di tutte le eccezioni del componente/applicazione. Il metodo usato sarà simile a quello che segue: il caricamento della stringa come risorsa (che qui è invece costante) potrebbe avvenire con la classe `ResourceManager`⁹.

```
public class ExceptionBuilder {
    public ExceptionBuilder() {}

    public static ArgumentOutOfRangeException( string paramName,
        string minValue,
        string maxValue ) {
        return new ArgumentOutOfRangeException(
            paramName,
            String.Format( "Argument must be between {0} and {1}" ),
            minValue, maxValue );
    }

    // ...
}
```

⁹ La spiegazione del funzionamento di `ResourceManager` è al di fuori degli scopi di questo capitolo; consultare in merito la documentazione di MSDN.

ASP.NET

Per gestire al meglio gli errori in un'applicazione ASP.NET bisogna essere pienamente consci di cosa significa scrivere un'applicazione ASP.NET. Una normale applicazione eseguibile è un'applicazione dove il punto di ingresso del programma, il `Main`, è scritto dal programmatore, che ha il pieno controllo della situazione. Se egli decide di usare un componente, può istanziarlo e chiamarne dei metodi. Se vuole intercettare le eccezioni, scriverà appositamente del codice.

Scrivere un'applicazione ASP.NET è invece come scrivere (o se preferite estendere) dei componenti, che sono poi usati da un'applicazione, già scritta, che svolge autonomamente tutta una serie di attività. Tra queste c'è la gestione delle eccezioni. Una semplice pagina ASPX con un pulsante che richiama una funzione che fa una divisione per zero produce come risultato una pagina HTML con tutte le informazioni sull'eccezione che si è verificata. Tale pagina è personalizzabile con la proprietà `ErrorMessage` della classe base `System.Web.UI.Page` o sull'intera applicazione ASP.NET attraverso il tag `<customErrors>` in `web.config` (come nell'esempio che segue).

```
<configuration>
  <system.web>
    <customErrors defaultRedirect="ErrorPage.htm" mode="RemoteOnly" />
  </system.web>
</configuration>
```

Tralasciamo la personalizzazione della visualizzazione di un'eccezione intercettata da ASP.NET, che è argomento per libri più orientati al Web. Come possiamo eseguire del codice particolare al verificarsi di un'eccezione? Considerata l'architettura di ASP.NET, quasi tutte le considerazioni che abbiamo fatto per i componenti condivisi restano valide anche per questo tipo di applicazione. Una differenza significativa è che alcune eccezioni, in particolare quelle che possiamo aspettarci dalla chiamata di un componente, possono essere intercettate e risolte dal codice di una pagina ASP.NET, specialmente quando il problema è prevedibile e produce semplicemente un output HTML diverso. Quindi, qualche blocco `try/catch` risolutivo¹⁰ in più rispetto a un componente puro è perfettamente lecito, ma sempre evitando di usare sistematicamente `catch { e catch (Exception) }`.

Poiché abbiamo un controllo maggiore sull'applicazione rispetto a un componente puro, possiamo anche pensare a dei meccanismi generici per intercettare le eccezioni (in genere di tipo diagnostico e non risolutivo). Una pagina ASP.NET è una classe che deriva da `System.Web.UI.Page`, dove è disponibile un evento `Error` a cui si può agganciare un delegate in seguito invocato per tutte le eccezioni non gestite, che comunque vanno sempre a richiamare anche la pagina di errore standard o personalizzata. Il metodo agganciato a tale evento non riceve direttamente l'eccezione, ma deve andarla a recuperare da `Server.GetLastError: Server` è una proprietà di tipo `HttpServerUtility`, che recupera l'errore da `HttpApplication` o da `HttpContext` (secondo il punto in cui si è generato l'errore), dando la priorità a quest'ultimo. Il codice che segue evidenzia l'associazione all'evento e l'acquisizione dell'oggetto `Exception` intercettato.

¹⁰ Intercettare un'eccezione in modo risolutivo significa risolvere la situazione, proseguendo l'esecuzione senza richiamare nuovamente `throw` nel blocco `try/catch`.

DIR: EXCEPTION\ASP.NET\EXCEPTIONS FILE: WEBFORM1.ASPX.CS

```
private void InitializeComponent() {
    this.btnDivisionByZero.Click +=
        new System.EventHandler(this.btnDivisionByZero_Click);
    this.btnExceptionFinally.Click +=
        new System.EventHandler(this.btnExceptionFinally_Click);
    this.Error += new System.EventHandler(this.WebForm1_Error);
    this.Load += new System.EventHandler(this.Page_Load);
}

private void WebForm1_Error(object sender, System.EventArgs e) {
    Exception ex = Server.GetLastError();
    Debug.WriteLine( "Error : " + ex.GetType().Name );
    Debug.WriteLine( ex.Message );
    Debug.WriteLine( ex.StackTrace );
}
```

Per agganciare del codice alle eccezioni gestite di tutte le pagine di un sito, e non a una pagina specifica, bisogna agire sulla classe derivata da `HttpApplication` accessibile attraverso il file `global.asax.cs`: è possibile scrivere il codice all'interno del metodo `Application_Error` o definire un proprio delegate associato all'evento `Error`¹¹. Un esempio è l'invio di una mail per ogni errore intercettato.

DIR: EXCEPTION\ASP.NET\EXCEPTIONS FILE: GLOBAL.ASAX.CS

```
protected void Application_Error(Object sender, EventArgs e) {
    StringBuilder errorText = new StringBuilder();
    int innerLevel = 0;
    Exception ex = Server.GetLastError();
    while (ex != null) {
        errorText.AppendFormat( "Application Error (inner={0}): {1}",
            innerLevel, ex.GetType().Name );
        errorText.Append( ex.Message );
        errorText.Append( ex.StackTrace );
        ex = ex.InnerException;
        innerLevel++;
    }
    MailMessage mailError = new MailMessage();
    mailError.To = "marco@devleap.com";
    mailError.From = "ExceptionDemo@devleap.com";
    mailError.Subject = "Unhandled Exception in Application";
    mailError.Body = errorText.ToString();
    mailError.BodyFormat = MailFormat.Text;
    try {
        SmtpMail.SmtpServer = "relay.devleap.com";
        SmtpMail.Send(mailError);
    }
}
```

¹¹ Il metodo `Application_Error`, se definito, è agganciato automaticamente all'evento `HttpApplication.Error`. I due sistemi sono quindi equivalenti.

Capitolo 2

```

catch (Exception ex2) {
    Debug.WriteLine( "Exception sending Smtplib mail: " + ex2.GetType().Name );
    Debug.WriteLine( ex2.Message );
    Debug.WriteLine( ex2.StackTrace );
}
}

```

Il motivo per cui c'è un loop per elencare la lista di eccezioni seguendo `InnerException` è che l'eccezione tipicamente ricevuta a livello di `HttpApplication` è `HttpUnhandledException`, che contiene in `InnerException` un riferimento all'eccezione originale. Personalmente ritengo che questo comportamento non sia molto coerente: la documentazione di `HttpUnhandledException` complica ancora le cose sostenendo che essa è a uso interno e non dice altro, rendendo la cosa ancora più criptica. Sapendo come funziona, comunque, ci si può convivere. Per ottenere l'eccezione originale, senza scrivere il loop dell'esempio, si può usare l'espressione `Server.GetLastError().GetBaseException()`, mentre sarebbe sbagliato affidarsi a `Server.GetLastError().InnerException`: se l'eccezione provenisse dall'evento `Error` di una pagina (potrebbe capitare!) `InnerException` non sarebbe valorizzato e l'eccezione originale sarebbe proprio quella restituita da `Server.GetLastError`.

Nell'esempio si può anche notare il `try/catch` in cui è inserito l'invio della mail. Un'eventuale eccezione su `SmtplibMail.Send` non sarebbe altrimenti gestita in alcun modo, come capiremo tra poco. A parte questo, sarebbe comunque opportuno mettere `SmtplibMail` in una chiamata asincrona a cui passare il messaggio composto in `mailError`: la chiamata sincrona produce infatti un inutile ritardo nella produzione della pagina HTML all'utente per la gestione dell'errore (personalizzata o meno). Questa ottimizzazione è lasciata come utile esercizio per il lettore.

Giochi pericolosi

ASP.NET prende il controllo di parecchi aspetti relativi alla gestione delle eccezioni, creando un ambiente che è apparentemente più semplice e forse più vicino alla modalità di programmazione di VB6 e ASP. Dietro questa semplificazione si celano però dei rischi.

Una prima attività "rischiosa", tipicamente involontaria, è quella di generare un'eccezione all'interno di un evento `Error`. Quando ciò avviene a livello di pagina, come abbiamo appena visto, l'eccezione può essere catturata dall'evento `Error` di `HttpApplication` (in `global.asax.cs`). Se invece l'eccezione è generata proprio all'interno di `Application_Error` o nell'evento `Error` equivalente, allora l'eccezione viene persa. Sì, ASP.NET pensa bene di intercettare ed eliminare tutte le eccezioni che si verificano da quelle parti. Anche con `AppDomain.UnhandledException`, che vedremo tra poco, non si può intercettare nulla. È come se la chiamata di `Application.Error` avvenisse con un codice simile a quello che segue (notare che è un codice ipotetico, anche perché la proprietà `LastError` usata non esiste).

```

try {
    // ...
}
catch (Exception ex) {

```

```

try {
    Application.LastError = ex;
    Application.Error( this, EventArgs.Empty );
}
catch {
}
}

```

La parte importante è quella evidenziata: l'uso di `catch {}` porta alle conseguenze che abbiamo già avuto modo di descrivere e deprecare.

Che fare? Un `try/catch` dentro `Application_Error` è l'unico modo per intercettare eventuali errori che si verifichino da quelle parti. Non è granché, ma non ci sono alternative migliori se non si vogliono perdere eventuali errori dentro la gestione degli errori!

Il secondo comportamento "a rischio" è quello di annullare la condizione di errore all'interno di un evento di gestione degli errori. Sia in `Page.Error` (gestione a livello di pagina) che in `HttpApplication.Error` (e quindi in `Application_Error`) si può decidere di annullare l'errore: chiamando `Server.ClearError`, l'esecuzione dell'applicazione ASP.NET procede senza arrivare a eseguire la pagina di gestione dell'errore definita in `web.config` (o quella di default). Quindi, un codice come quello che segue provoca la visualizzazione di una pagina HTML costruita dinamicamente con le `Response.Write`, senza chiamare la pagina predefinita di gestione degli errori.

```

private void WebForm1_Error(object sender, System.EventArgs e) {
    Exception ex = Server.GetLastError();
    Server.ClearError();
    Response.Write("<pre>");
    Response.Write( "Page Error : " + ex.Message );
    Response.Write( "<br>");
    Response.Write( ex.StackTrace );
    Response.Write("</pre>");
}

```

Il rischio è insito nella chiamata che sta a monte di tutto ciò, che è simile al codice successivo. Dopo la chiamata all'evento, la proprietà `LastError` potrebbe risultare annullata a fronte della chiamata di `Server.ClearError`: se ciò è avvenuto, l'eccezione si considera gestita e l'applicazione ASP.NET prosegue, altrimenti l'eccezione continua la sua strada.

```

try {
    // ...
}
catch (Exception ex) {
    LastError = ex;
    if (Page.Error != null) Page.Error( this, EventArgs.Empty );
    if (LastError != null) {
        throw;
    }
}
}

```

Capitolo 2

Ovviamente la funzione che ha generato l'eccezione è stata interrotta, così come i suoi chiamanti: probabilmente la pagina o l'operazione richiesta non è stata completata correttamente. Non sto dicendo che ci sia qualcosa di sbagliato, ma solo che ci si va a trovare in una situazione delicata: se non si sono usate tutte le precauzioni dei componenti (`try/finally` prima di tutto), se non si è pensato a cosa fare dopo aver chiamato `Server.ClearError` per visualizzare qualcosa di sensato all'utente, ecco che ci si può venire a trovare, subito dopo, in situazioni di errore difficili da analizzare. Rilanciare un'eccezione con `throw`; non fa perdere nessuna informazione, mentre "digerire" l'eccezione e andare avanti è sempre qualcosa da fare con attenzione: ricordo che la `catch` usata cattura qualsiasi eccezione!

Se avete fatto attenzione al codice precedente, c'è una terza possibile fonte di grattacapi. Le eccezioni intercettate da ASP.NET sono solo quelle conformi a CLS, cioè quelle derivate da `Exception`. Non è probabile che succeda, ma teoricamente un componente scritto in Managed C++ potrebbe generare eccezioni non derivate da `Exception`: in un caso simile, ASP.NET intercetta l'eccezione ma non fornisce alcuna diagnostica, né attraverso ridirezione su una pagina apposita né attraverso gli eventi `Error`, che non sono richiamati. Insomma, un'eccezione simile viene intercettata, digerita e "persa". Ripeto, l'ipotesi che qualcuno decida di scrivere un componente simile è piuttosto remota (se scrivete componenti in C++, non generate eccezioni del genere!), ma al solito di fronte a un comportamento "anomalo" è meglio essere pronti a tutto.

Catturare quello che ASP.NET si perde

Come abbiamo visto, con le tecniche basate sugli eventi `Error` non si intercettano tutte le eccezioni. ASP.NET non intercetta *qualsiasi* eccezione, ma solo quelle che si verificano nel thread che esegue il codice della pagina Web. Tutti i casi di multi-threading che abbiamo già visto in "Eccezioni non gestite" rientrano pienamente nelle situazioni che potremmo avere in un'applicazione ASP.NET, e nessuna delle eccezioni generate in thread secondari è catturabile con questo meccanismo.

Come già sappiamo, usando l'evento `AppDomain.UnhandledException` è possibile intercettare anche tali eccezioni. Però attenzione: le eccezioni generate nel thread principale delle pagine ASP.NET sono intercettabili solo tramite gli eventi `Error` che abbiamo già esaminato, perché di fatto non rientrano nel novero delle eccezioni "non gestite". Il codice che segue mostra come attivare la gestione tramite `AppDomain.UnhandledException` a livello globale.

DIR: EXCEPTION\ASP.NET\EXCEPTIONS FILE: GLOBAL.ASAX.CS

```
protected void Application_Start(Object sender, EventArgs e) {
    AppDomain.CurrentDomain.UnhandledException +=
        new UnhandledExceptionHandler(CurrentDomain_UnhandledException);
}

private void CurrentDomain_UnhandledException( object sender,
                                               UnhandledExceptionHandlerEventArgs e) {
    Debug.Write( "Unhandled exception : " );
    Debug.WriteLine( e.ExceptionObject.GetType().Name );
    Exception ex = e.ExceptionObject as Exception;
```

```
if (ex != null) Debug.WriteLine( ex.StackTrace );
}
```

Per simulare un'eccezione proveniente da un thread diverso usiamo la stessa tecnica usata in precedenza: generiamo un'eccezione di divisione per zero nel distruttore (cioè nel metodo `Finalize`) di una classe istanziata tramite una pagina Web. Notare che le chiamate a `GC.Collect` e `GC.WaitForPendingFinalizers` sono funzionali unicamente a non ritardare la generazione dell'eccezione a scopo dimostrativo e non vanno usate in un'applicazione reale.

DIR: EXCEPTION\ASP.NET\EXCEPTIONS FILE: WEBFORM1.ASPX.CS

```
class Simple {
    -Simple() {
        int b, c;
        b = 0;
        c = 5 / b;
    }
}

private void btnExceptionFinally_Click(object sender, System.EventArgs e) {
    Simple x = new Simple();
    x = null;
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

Per chiudere, un'annotazione sulle eccezioni non derivate da `Exception`: a differenza degli eventi `Error`, con `AppDomain.UnhandledException` si intercettano anche queste eccezioni (ma solo per thread diversi da quello principale!). Notare che nel codice di `global.asax.cs` l'oggetto `ExceptionObject` è di tipo `Object` e viene convertito in `Exception` per accedere alla proprietà `StackTrace`, che è letta solo se tale conversione risulta valida.

Pagine, User Control e Custom Control

ASP.NET predispone un ambiente in cui è già definita una strategia di gestione degli errori ben precisa. Purtroppo l'attuale documentazione non fornisce un quadro molto chiaro: cerchiamo di porvi rimedio.

Una pagina ASP.NET è composta da codice HTML statico e dinamico. Il codice HTML dinamico è generato da codice eseguibile che, durante l'esecuzione, può generare delle eccezioni. Tale codice può risiedere in tre posti:

- il code-behind della pagina;
- il code-behind degli User Control usati nella pagina;
- il codice delle classi dei Custom Control usati nella pagina.

Una prima strategia, comune a tutti questi elementi, è di definire dei blocchi `try/catch` per intercettare le eccezioni risolvibili "localmente". Se è accettabile che una

Capitolo 2

qualsiasi eccezione “blocchi” la costruzione della pagina attuale passando il controllo a una speciale pagina di errore, è possibile agire a livello di evento `Error` sulla pagina o su `HttpApplication`. L'errore potrebbe però essere considerato “non bloccante”: in tal caso la pagina va costruita e visualizzata all'utente, magari con alcune parti incomplete o errate.

La condizione di errore “non bloccante” è quella che ci interessa di più: anche se non vogliamo interrompere la costruzione della pagina, ci può interessare acquisire (magari in un log) informazioni diagnostiche sull'errore. Se un errore simile si genera in uno `User Control` o in un `Custom Control`, il programmatore (del controllo) può decidere di catturare l'errore con una `catch`¹² e “digerirlo”, chiamando `HttpContext.Current.AddError` per aggiungere l'eccezione a un elenco raggiungibile con `HttpContext.Current.AllErrors`.

DIR: EXCEPTION\ASP.NET\EXCEPTIONS2 FILE: GLOBAL.ASAX.CS

```
private void Label1_PreRender(object sender, System.EventArgs e) {
    try {
        Label1.Text = DateTime.Now.ToShortTimeString();
        int b = 0;
        Debug.WriteLine( "*** Division" );
        int c = 5 / b;
    }
    catch( Exception ex ) {
        Debug.WriteLine( String.Format(
            "PreRender Caught {0}", ex.GetType().Name ) );
        Context.AddError( ex );
    }
}

protected override void Render(HtmlTextWriter output) {
    try {
        output.Write("<H2>" + DateTime.Now.ToLongTimeString() + "</H2><br>");
        base.Render(output);
        throw new Exception("Simulation of rendering error");
    }
    catch( Exception ex ) {
        Debug.WriteLine( String.Format(
            "Render Caught {0}", ex.GetType().Name ) );
        Context.AddError( ex );
    }
}
```

¹² Purtroppo non è possibile agire in questo modo intercettando l'evento `Error` a livello di pagina o di `User Control`. Un'eccezione durante la fase di render in un controllo blocca il ciclo di rendering dei controlli della pagina (la gestione dell'evento `Error` è al di fuori di tale ciclo), interrompendo la costruzione della pagina e la chiamata di altri `User Control` o `Custom Control`. Per questo è necessario usare `try/catch` nei metodi a rischio. Per controllare tutta la fase di rendering, si può fare un override del metodo `Render` del controllo, mettendo in un blocco `try/catch` la chiamata al metodo base. In alternativa, si potrebbe specializzare e riscrivere a livello di pagina il metodo virtuale `RenderChildren`.

Nell'esempio sono evidenziati due casi: una funzione che non fa parte del rendering della pagina (`Label1_PreRender`, associata all'evento omonimo) e la specializzazione di `Render`. La struttura è identica, sfruttando `try/catch` e la chiamata di `AddError`. `Context` è una proprietà derivata da `Control` che restituisce il contesto corrente.

L'elenco `AllErrors` può essere poi letto alla fine della costruzione della pagina: scorrendone tutti gli elementi, è possibile generare un log che tenga traccia di tutti i componenti nella pagina che hanno qualche problema, anziché limitarsi al primo errore trovato. Questa tecnica può essere generalizzata per tutte le pagine, mettendo la gestione del log all'interno di `Application_EndRequest` in `global.asax.cs` (o ancora meglio in un componente generico agganciato a questa funzione o all'evento `HttpApplication.Error`).

DIR: EXCEPTION\ASP.NET\EXCEPTIONS2 FILE: GLOBAL.ASAX.CS

```
protected void Application_EndRequest(Object sender, EventArgs e) {
    int i = 0;
    if (Context.AllErrors != null) {
        foreach( object o in Context.AllErrors) {
            Exception ex = o as Exception;
            Debug.WriteLine( String.Format( "Exception {0}", ++i ) );
            Debug.WriteLine( String.Format( "Exception type : {0}",
                o.GetType().Name ) );

            if (o != null) {
                Console.WriteLine( ex.Message );
                Console.WriteLine( ex.StackTrace );
            }
        }
        Server.ClearError();
    }
}
```

Notare che le proprietà di cui stiamo parlando sono definite in `HttpContext`: un'eccezione non gestita, in una pagina o in uno dei controlli della pagina stessa, viene automaticamente aggiunta tramite `AddError` all'elenco di eccezioni avvenute nel contesto corrente, anche se in tal caso la generazione della pagina è interrotta. La proprietà `HttpContext.Error` restituisce la prima eccezione verificatasi, ma chiamando `ClearError` si elimina solo la condizione di "errore" del contesto corrente, mentre l'elenco di eccezioni accessibile con `AllErrors` resta inalterato. La chiamata di `ClearError` è necessaria solo se, all'interno dell'evento `Page.Error` o a seguito della lettura di `AllErrors` in `Application_EndRequest`, si vuole evitare di passare alla pagina di errore.

L'esempio completo, di cui abbiamo visto alcuni estratti, contiene due istanze di un componente che genera degli errori, rendendo potenzialmente la pagina "incompleta" ma comunque parzialmente visibile. L'errore (come in questo caso) potrebbe anche non essere visibile all'utente: un menu dinamico potrebbe non contenere delle voci, senza evidenziare all'utente tale mancanza.

ADO.NET

ADO.NET non è un framework applicativo come ASP.NET o Windows Forms: per questo motivo non c'è un'infrastruttura già definita per la gestione degli errori (o delle

Capitolo 2

eccezioni non gestite). In un mondo semplice tutte le eccezioni legate in qualche modo all'accesso ai dati deriverebbero da una specifica classe base. Nel mondo reale (che è un po' più complesso, ma peraltro è l'unico che ci è dato di vivere) le eccezioni sono molte, dipendono da tanti fattori, non è semplice elencarle tutte e ovviamente sono organizzate gerarchicamente in un modo che non rende semplice scrivere un solo blocco `catch` generico per l'accesso ai dati!

Con queste premesse si potrebbe già passare alla sezione successiva, ma cerchiamo comunque di avere un'idea dello scenario che ci si trova di fronte usando ADO.NET. Questa, come qualsiasi libreria, ha le sue eccezioni, che derivano prevalentemente da `System.Data.DataException`, ma ciò non toglie che un'eccezione di divisione per zero sia sempre `DivisionByZeroException`, anche se avviene in un contesto di accesso ai dati. Normalmente, poi, ADO.NET è usata in congiunzione con un'altra libreria per la connessione a una fonte dati¹³, che può avere delle sue eccezioni come `OleDbException`, `SqlException` e così via, che non sono derivate da `DataException`.

Di seguito vediamo una parte della gerarchia di eccezioni definite nel Framework: quelle evidenziate sono sicuramente generabili da ADO.NET (anche questo elenco non è esaustivo).

```
System.Exception
  System.SystemException
    System.InvalidOperationException
    System.Data.SqlClient.SqlException
    System.Runtime.InteropServices.ExternalException
      System.Data.OleDb.OleDbException
    System.Data.DataException
      System.Data.ConstraintException
      System.Data.DeletedRowInaccessibleException
      System.Data.DuplicateNameException
      System.Data.InRowChangingEventException
      System.Data.InvalidConstraintException
      System.Data.InvalidExpressionException
      System.Data.MissingPrimaryKeyException
      System.Data.NotNullAllowedException
      System.Data.ReadOnlyException
      System.Data.RowNotInTableException
      System.Data.StrongTypingException
      System.Data.TypedDataSetGeneratorException
      System.Data.VersionNotFoundException
```

Come si vede, non sono tutte eccezioni definite da ADO.NET. Per esempio, se la proprietà `DataAdapter.MissingSchemaAction` vale `Error` e la colonna specificata nel mapping non esiste, ADO.NET genera un'eccezione `InvalidOperationException`, che non deriva da `DataException` ma è ben documentata su MSDN.

Come districarsi? Bisogna leggere la documentazione, pensare a quello che si sta facendo, capire cosa si vuole intercettare e perché, quindi agire di conseguenza. Per gestire le eccezioni in modo risolutivo e non semplicemente diagnostico non ci sono

¹³ Da un punto di vista pratico, stiamo parlando di altri assembly, come `System.Data.OleDb`, `System.Data.SqlClient` e simili, mentre il cuore di ADO.NET è l'assembly `System.Data`.

scorciatoie. Intercettare `SystemException` non è una soluzione perché è troppo generica, praticamente quanto `Exception`.

In ADO.NET, di fatto, non si usano gli eventi per gestire gli errori come meccanismo complementare o alternativo a `try/catch`. Esistono degli eventi che definiscono aspetti funzionali ma che non generano eccezioni, ed esistono eccezioni specifiche per problemi di accesso ai dati. Un caso particolare è l'evento `DataSet.MergeFailed`, che consente di specificare un delegate richiamato in caso di fallimento della funzione `DataSet.Merge`: se e solo se non ci sono delegate associati a questo evento, ADO.NET genera un'eccezione `DataException`.

Una delle cause per la generazione di eccezioni derivate da `DataException` è il tentativo di violare regole di integrità referenziale della struttura dati. La generazione di un'eccezione interrompe l'operazione in corso e non fornisce un dettaglio così approfondito sulla causa dell'errore (in casi simili serve conoscere con precisione il record e la colonna che hanno generato l'errore). Ancora più importante, a volte (specie negli aggiornamenti che coinvolgono più record) è utile non fermare l'intera operazione, cercando di portare a termine ciò che non presenta errori e segnalare alla fine tutte le attività che non sono andate a buon fine, corredandole di spiegazioni esaustive. Per fare questo è bene sfruttare i metodi `SetColumnError` e `GetColumnError` di `DataRow`, che consentono di controllare gli errori dal punto di vista della logica applicativa¹⁴ prima che dell'integrità referenziale e di trasmettere questi errori agli oggetti `DataTable` e `DataSet` sovrastanti, evitando la generazione di eccezioni.

Uno dei problemi che si possono incontrare in ADO.NET è che talvolta il codice di questa libreria trasforma un'eccezione generica (con `catch (Exception) {}`) in una più specifica, perdendo l'eccezione originale. È opportuno saperlo per evitare di dare per scontato che ADO.NET sia una libreria perfetta da questo punto di vista: purtroppo non lo è.

Per il resto, su ADO.NET valgono tutte le considerazioni fatte finora: se si usa ADO.NET all'interno di un componente, preoccuparsi di trasformare le eccezioni piuttosto che di gestire l'interfaccia utente; viceversa, se si usa ADO.NET all'interno di un'applicazione, il codice dovrà preoccuparsi di gestire in modo opportuno eventuali eccezioni specifiche.

Windows Forms

Se avete saltato la sezione su ASP.NET e siete passati direttamente a questa perché è quella che vi interessa di più, vi invito a tornare un attimo indietro. Molte delle considerazioni fatte per ASP.NET, infatti, valgono anche per Windows Forms.

Le applicazioni Windows Forms, come quelle ASP.NET, usano un'infrastruttura che definisce servizi e regole di comportamento per i componenti che ne fanno parte. Il programmatore, in effetti, definisce dei componenti (Form e controlli) che si inseriscono in un'architettura già definita. Analogamente ad ASP.NET, esistono degli eventi in grado di intercettare le eccezioni a livello applicativo.

L'evento `ThreadException` della classe `Application` è invocato per tutte le eccezioni non gestite all'interno dei metodi di un form o di un componente invocati a fronte di un evento dell'interfaccia utente. Questa definizione, per Windows Forms, corrispon-

¹⁴ Per farlo è necessario scrivere righe di codice che controllano la validità di un dato prima di tentarne l'inserimento o la modifica.

Capitolo 2

de alla quasi totalità dei metodi di un'applicazione, escludendo solo quei metodi eseguiti in thread diversi da quelli chiamati a gestire l'interfaccia utente. Poiché normalmente un'applicazione Windows ha un solo thread per gestire l'interfaccia utente, questo significa che qualsiasi operazione che avviene in una chiamata asincrona¹⁵, in un thread pool o in un thread definito dall'utente, non trasferisce le sue eccezioni all'evento `Application.ThreadException`.

In altre parole, l'evento `System.Windows.Forms.Application.ThreadException` è assimilabile all'evento ASP.NET `System.Web.Application.Error`: intercetta solo le eccezioni derivate dalla classe `Exception` che avvengono all'interno del thread "principale" dell'applicazione. Vediamo un esempio di utilizzo.

DIR: EXCEPTION

FILE: WINFORM.CS

```
static void Main()
{
    Application.ThreadException +=
        new System.Threading.ThreadExceptionHandler(
            Application_ThreadException);
    Application.Run(new Form1());
}

private void button1_Click(object sender, System.EventArgs e) {
    Test(null);
}

void Test(string name) {
    if (name == null) throw new ArgumentNullException( "name" );
    // ...
}

private static void Application_ThreadException(
    object sender, System.Threading.ThreadExceptionHandlerEventArgs e) {
    MessageBox.Show(
        String.Format( "{0}\r\nSave your work and restart application",
            e.Exception.Message ),
        "Application error" );
}
```

L'evento `Application.ThreadException` è chiamato per qualsiasi eccezione e può continuare l'esecuzione dell'applicazione. Questa *non* è una buona idea. In un blocco `try/catch` è possibile avere un'idea di come ripristinare uno stato consistente per continuare l'esecuzione, mentre in un metodo così generale è quasi impossibile¹⁶. Un'applicazione Windows Forms è un'applicazione stateful, mentre un'applicazione ASP.NET spesso è stateless. A meno di non aver individuato un tipo molto specifico di

¹⁵ Ricordiamo che, per la chiamata asincrona, l'eccezione è rilanciata nel thread che chiama `EndInvoke`.

¹⁶ Si potrebbe pensare a una somiglianza con `AppDomain.UnhandledException`, ma non è così, perché quest'ultimo è chiamato in extremis prima dell'uscita dall'applicazione, che resta un processo irreversibile.

eccezione, una buona idea è chiedere all'utente di salvare il lavoro e uscire al più presto. Un'eccezione catturata in questo evento è pur sempre un'eccezione non gestita correttamente da un'altra parte del programma!

La proprietà `ThreadException` ha poi due peculiarità piuttosto inusuali. Per prima cosa, non consente di associare più delegate: pur presentandosi come un normale evento, usa solo l'ultimo delegate associato, che va a ricoprire quelli precedenti. Per questa ragione, non è una buona idea assegnare questa proprietà in un punto diverso dalla funzione che chiama `Application.Run` (tipicamente il `Main`). Nel codice che segue, solo `Exception3` è associato all'evento `ThreadException`, mentre `Exception1` e `Exception2` sono eliminati dall'ultima assegnazione a `ThreadException`, nonostante l'uso dell'operatore `+=`.

```
static void Main()
{
    Application.ThreadException +=
        new System.Threading.ThreadExceptionHandler(Exception1);
    Application.ThreadException +=
        new System.Threading.ThreadExceptionHandler(Exception2);
    Application.ThreadException +=
        new System.Threading.ThreadExceptionHandler(Exception3);
    Application.Run(new Form1());
}
```

La seconda particolarità è che `ThreadException`, pur essendo un evento statico, è definito in modo diverso per ogni thread supportato da Windows Forms. Se si assegna `Application.ThreadException` da un thread secondario, non si modifica il valore eventualmente assegnato facendo la stessa operazione dal thread principale. Su questo punto non ci dilunghiamo oltre: lo sviluppo multi-thread in Windows Forms deve tenere conto del fatto che tale libreria non è thread-safe, il che non significa che non si possano realizzare applicazioni multi-thread, ma che vi sono molte precauzioni da prendere.

Per le applicazioni Windows Forms, l'uso di `AppDomain.UnhandledException` resta confinato alle eccezioni generate in thread diversi da quelli associati all'interfaccia utente (di norma uno solo, quello del `Main`) e alle eccezioni non derivate da `Exception`. In questo, la somiglianza con ASP.NET è mantenuta.

Servizi

Un servizio è un tipo di applicazione particolare: non ha interfaccia utente ed è eseguito in un contesto di sicurezza indipendente da quello dell'utente che sta eventualmente usando il sistema. La mancanza di interfaccia utente è l'aspetto più importante: qualsiasi errore va diagnosticato sotto forma di log e la logica di uscita dalla condizione di errore non può contare sull'interazione con l'utente.

In generale, la gestione delle eccezioni è quella che si ha in un'applicazione .NET multi-thread¹⁷. La segnalazione di errori deve avvenire sfruttando una tecnica di log:

¹⁷ Per sua natura, anche se è "solo", un servizio in .NET creato derivando la classe `ServiceBase` opera con più thread.

Capitolo 2

la strada migliore è usare la proprietà `EventLog` della classe derivata da `ServiceBase`, in quanto attraverso tale classe è possibile inserire informazioni nel log centralizzato di Windows, che può essere a sua volta monitorato con opportuni strumenti amministrativi.

La classe `ServiceBase` intercetta già le eccezioni generate all'interno di eventi e metodi virtuali (come `OnStart` e `OnStop`), che avvengono in un thread controllato dalla classe stessa: le informazioni su tali eccezioni sono scritte nel log di Windows usando `EventLog`. Un pattern comune nello scrivere un servizio consiste però nel creare uno o più nuovi thread nella specializzazione di `OnStart`. Le eccezioni generate in tali thread non sono intercettate da nessuno, ed è compito del programmatore provvedere in tal senso, anche per quanto riguarda il log. Un'eccezione non intercettata in un thread secondario provoca la distruzione del thread (e la chiamata all'evento `AppDomain.UnhandledException`, se definito), ma non comporta automaticamente la fermata del servizio (anche se non fa niente, Windows lo segnala ancora come servizio in esecuzione).

Una trattazione completa di come si debba scrivere un servizio in .NET è al di fuori degli obiettivi di questo capitolo. Per quanto riguarda la gestione degli errori, è opportuno ricordare che la massima attenzione va posta nel considerare quali sono i thread in gioco e se un'eccezione non gestita sarà catturata da `ServiceBase` o no. Il rischio è di "perdere" informazioni importanti sugli errori che avvengono all'interno di un servizio.

Web Service e Remoting

L'uso delle eccezioni con Web Service e .NET Remoting è in qualche modo limitato dalle caratteristiche di questi tipi di chiamate remote. Con .NET Remoting il concetto di eccezione può essere trasparente, a patto di definire nuove eccezioni in modo che siano serializzabili. Con i Web Service, invece, bisogna abbandonare alcune delle informazioni che le eccezioni possono trasportare.

Web Service

Un Web Service può essere visto come un servizio applicativo offerto attraverso un'interfaccia standard indipendente da .NET. In questo senso una qualsiasi eccezione .NET che si verifichi durante la chiamata di un Web Service non può essere trasmessa all'esterno del "lato server", perché il client potrebbe non essere in grado di riconoscere l'errore. L'eventuale eccezione .NET è catturata e inviata al client inviando un `soap:Fault` nel `soap:Body` del messaggio SOAP.

Se il client è anch'esso .NET, questo errore è trasformato in un'eccezione `SoapException`, nella cui proprietà `Message` sono contenute informazioni sull'eccezione originale. Se sul server vi erano più eccezioni nella lista definita da `InnerException`, il client riceve comunque una sola `SoapException`, ma `Message` contiene le informazioni rilevate dall'analisi della lista di eccezioni in `InnerException`, incluse le informazioni di `StackTrace` di tutte le eccezioni della lista. Tali informazioni sono trasferite come stringa, quindi non è necessario avere a disposizione sul client i tipi a cui corrispondono le eccezioni (come invece avviene in Remoting).

Per capire cosa succede su un client che non sia .NET, è necessario analizzare cosa transita nel messaggio SOAP. Quello che segue è il risultato che si ottiene con gli

errori custom disabilitati¹⁸, è usata la specifica SOAP 1.1; la versione 1.2 implementa un maggiore dettaglio e *dovrebbe*¹⁹ essere supportata dalla versione 1.1 di .NET Framework.

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>System.Web.Services.Protocols.SoapException: Server was
unable to process request. ---> System.ArgumentException: Invalid frequency --->
System.DivideByZeroException: par1 can't be 0 at WsTest.wsTest.t2() in c:\inet-
pub\wwwroot\WsTest\wsTest.asmx.cs:line 66 at WsTest.wsTest.t1() in
c:\inetpub\wwwroot\WsTest\wsTest.asmx.cs:line 60 --- End of inner exception
stack trace --- at WsTest.wsTest.t1() in
c:\inetpub\wwwroot\WsTest\wsTest.asmx.cs:line 62 at WsTest.wsTest.test() in
c:\inetpub\wwwroot\WsTest\wsTest.asmx.cs:line 56 at
WsTest.wsTest.SampleFunc(Int64 nTimeout) in
c:\inetpub\wwwroot\WsTest\wsTest.asmx.cs:line 41 --- End of inner exception
stack trace ---</faultstring>
      <detail />
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Se invece gli errori custom sono abilitati, il contenuto di <faultstring> è un po' più scarso.

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>Server was unable to process request. --> Invalid frequency
--> par1 can't be 0</faultstring>
      <detail />
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

¹⁸ Per disabilitare gli errori custom si imposta in web.config il tag <customErrors mode="Off" />; si può anche usare <customErrors mode="RemoteOnly" /> disabilitando gli errori custom solo per le richieste locali.

¹⁹ Al momento della scrittura di questo capitolo (aprile 2003) .NET 1.1 è stato appena rilasciato: il codice per supportare SOAP 1.2 è presente ma non è abilitato, perché la relativa raccomandazione non è stata ancora rilasciata; quando ciò avverrà, un apposito Service Pack potrebbe abilitare tale supporto.

Capitolo 2

Un'eccezione è trasferita nel nodo `soap:Fault`, in cui si trova la descrizione di `SoapException` seguita dalla lista di eccezioni rilevate sul server, compresi tutti i vari `StackTrace`, se gli errori custom sono disabilitati. Il messaggio SOAP dell'esempio mostra un'eccezione originale `DivideByZeroException` trasformata in `ArgumentException`, coi relativi parametri e lo stack di chiamata. Dunque, un client che non sia .NET è comunque in grado di fornire un minimo di informazioni diagnostiche sul problema che è avvenuto sul server. Una soluzione del problema è comunque delegata al responsabile del componente sul lato server.

Attivando gli errori custom nella configurazione dell'applicazione ASP.NET si perdono alcune informazioni, come `StackTrace`, il che in parte è anche una scelta: trasferire informazioni simili via HTTP in chiaro non è molto saggio dal punto di vista della security (sarebbero pur sempre informazioni sull'implementazione interna di un servizio remoto). L'impostazione di default è `<customErrors mode="RemoteOnly" />`, che prevede di avere informazioni dettagliate per le richieste locali, mentre per le richieste provenienti da remoto è fornito solo il contenuto di `Message` delle eccezioni generate.

Remoting

Un'eccezione che avviene su un componente remoto è trasferita in modo trasparente al client, a patto che:

- la classe dell'eccezione sia decorata con l'attributo `[Serializable]`;
- la classe dell'eccezione implementi `ISerializable`;
- la classe dell'eccezione abbia un costruttore che riceve due parametri di tipo `SerializationInfo` e `StreamingContext`;
- il client disponga dei metadati della classe dell'eccezione

L'eccezione ricevuta dal client contiene in `StackTrace` informazioni sullo stato dello stack sia dal lato server che dal lato client: lo strato di Remoting che intercetta la chiamata sul client e la trasferisce al server provvede anche a intercettare l'eventuale eccezione sul server, rigenerandola sul client con tutte le proprietà associate (compresa la lista definita da `InnerException`, se esiste).

Se per qualche motivo il client non conosce il tipo dell'eccezione, l'eccezione è trasformata in `SerializationException`: il client tenta di caricare il tipo corrispondente alla classe dell'eccezione e in `Message` si troverà una descrizione del tipo e dell'assembly che non è stato trovato. È interessante notare che, col contenuto di `Message`, è comunque possibile stabilire il nome della classe dell'eccezione: si può capire quale sia stata l'eccezione generata sul server e in che punto (grazie a `StackTrace`).