Capitolo 7

Le eccezioni

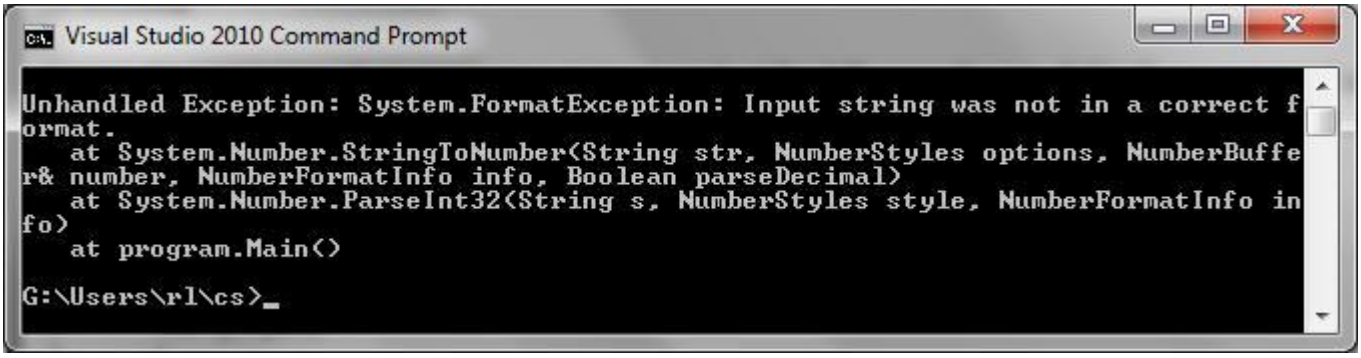
Si tratta di un argomento decisamente importante non solo ai fini dell'apprendimento di C#. Il meccanismo di gestione delle eccezioni infatti appartiene al framework, non è legato solo al singolo idioma per cui apprenderne i meccanismi aiuterà anche nel caso in cui si vogliano apprendere altri linguaggi .Net compliant.

Per introdurre questo argomento partiamo dal solito semplice esempio di base, premettendo fin da subito che gli esempi di questo paragrafo saranno piuttosto banali e non proprio "belli" da un punto di vista della programmazione in quanto in realtà non è così semplice creare delle situazioni adatte ad un uso proprio alle eccezioni:

C# Esempio 7.1

```
1   using System;
2   class program
3   {
4     public static void Main()
5     {
6       Console.Write("Inserisci un numero: ");
7       int x = Int32.Parse(Console.ReadLine());
8       Console.WriteLine("Il tuo numero + 1: " + (x + 1));
9     }
10  }
```

Questo codice funziona tranquillamente se l'utente immette un numero qualunque come risposta alla richiesta che gli viene posta; ma che succede se dà un invio senza immettere nulla? O se non scrive un carattere numerico ma una lettera qualsiasi? Succede questo:



Proprio così, il programma va in crash e termina lasciandoci solo un criptico messaggio funebre. Tuttavia la prima riga ci dice esattamente cosa è successo: si è verificata una eccezione che il programma non ha potuto gestire. Inoltre ci viene detto quale è il problema, la stringa in input non era nel formato giusto. Un altro esempio di base che troverete sovente nella lettura dedicata è la divisione per 0. Oppure è frequente, anche nella pratica, il tentativo di accedere ad un file o ad una risorsa per qualche motivo non disponibile.

Errori di questo tipo, non solo dovuti ovviamente a comportamenti errati dell'utente ma anche determinati da veri e propri bugs nei programmi (e chi non ne fa?) oppure causati da eventi non prevedibili (la rottura di una periferica, tanto per dirne uno) sono più comuni di quanto si pensi e la loro gestione deve essere presa seriamente in esame per la scrittura di applicazioni robuste. E' possibile certamente cercare di circoscrivere molte situazioni potenziali fonti di guai ma in alcuni casi viene utile ricorrere ad una completa gestione delle eccezioni. Il nostro framework ci propone come soluzione la Structured Exception Handling (SEH) un metodo di gestione come detto **unificato** per chiunque si interfacci con il framework nel senso che le relative risorse appartengono alla piattaforma non specificatamente al linguaggio quindi sono disponibili anche se non si usa C#.

Il SEH è una parte fondamentale del framework ed è costituito da una serie di classi dedicate all'intercettazione di una vasta base di eccezioni. Il tutto si trova sotto la grande ala protrettrice del namespace System. In praticolare troviamo le seguenti classi (siete pronti?):

- AccessViolationException - che si verifica per un tentativo di scrivere o modificare una zona di memoria protetta
- AggregateException - raggruppa più eccezioni che si verifichino durante l'esecuzione
- AppDomaninUnloadedException - che si verifica al tentativo di accedere ad una applicazione non disponibile
- ApplicationException - segnala un errore non fatale dell'applicazione
- ArgumentException - viene sollevata quando si passa un argomento non valido ad un metodo
- ArgumentNullException - si ha questa eccezione quando viene passato un riferimento null ad un metodo che non prevede null come valido
- ArgumentOutOfRangeException - Ovviamente si verifica quando un un valore viene passato fuori dal proprio range accettabile
- AritmeticException - Si ha quando vi verifica un errore di tipo aritmetico a seguito di una operazione, di un cast o di una conversione
- ArrayTypeMismatchException - Si ha quando cerchiamo di porre in un array un elemento di tipo non accettabile per l'array stesso
- BadImgeFormatException - Quando si cerca di lanciare un programma o una DLL con un formato non accettabile
- CannotUnloadAppDomainException - Si verifica se fallisce il tentativo di scaricare il dominio di una applicazione
- ContextMarshalException - se fallisce una applicazione di marshaling
- DataMisalignedException - si ha quando fallisce il tentativo di leggere o scrivere dei dati a causa di un errore di indirizzo che non è un multiplo previsto
- DivideByZeroException - il famoso tentativo di dividere un numero per zero
- DllNotFoundException - se non può essere caricata la DLL invocata
- DuplicateWaitObjectException - Quando si trova un pggetto duplicato in un array di sicronizzazione oggetti
- EntryPointNotFoundException - Questa eccezione viene lanciata quando fallisce il tentativo di caircare una classe per la mancanza di un entrypoint
- FieldAccessException - Si verifica quando si cerca di accedere ad un campo privato o

protetto di una classe

- FormatException - Come nel nostro esempio iniziale, classica eccezione che si verfica nel passare un parametro in un formato scorretto (appunto un carattere laddove era atteso un numero)
- IndexOutOfRangeException - un classico anche questo, uando si cerca di accedere ad un indice fuori range ad esempio in un array
- InsufficientExecutionStackexception - Il nome indica già tutto, manca spazio sullo stack per l'esecuzione di alcuni metodi che, evidentemente, sovraccaricherebbero lo stack di dati.
- InsufficientMemoryException - Indica l'insufficienza di memoria disponibile
- InvalidCastexception - Indica il tentativo di effettuare un cast illegale
- InvalidOperationException - Forse un po' generico indica di solito il tentativo di lanciare un metodo quando l'oggetto non è in stato valido per l'effettuazione di quella operazione
- InvalidProgramException - Errore molto temuto dai language writer, indica spesso una incoerenza nel linguaggio IL generato dal compilatore. Brutto affare, se capita
- InvalidTimeZomeException - Si riferisce ad informazioni nin valido relative a zona e orario
- MemberAccessException - Segnala il fallimento nel tentativo di accedere ad un membro di una classe
- MethodAccessException - Si ha quando si cerca di accedere ad un metodo privato o protetto di una classe
- MissingFieldException - tentativo di accedere dinamicamente ad un campo inesistente
- MissingMemberExcpetion - tentativo di accedere dinamicamente a un membro inesistente
- MissingMethodExcpetion - tentativo di accedere dinamicamente a un metodo inesistente
- MulticastNotSupportedException - eccezione che viene lanciata quando si tenta di combinare due delegai basati sul tipo Delegate invece MultiCastDelegate
- NotFiniteNumberException - si ha questa eccezione quando un floating point risulta essere un infinito positivo, un infinito negativo o in uno stato di NaN (Not a Number)
- NotImplementedException - si verifica quando si cerca di utilizzare un metodo o una operazione non implementate
- NotSupportedexception - si ha quando si cerca di chiamare un metodo non supportato
- NullReferenceException - si ha quando si cerca di dereferenziare un riferimento ad un oggetto null
- ObjectDisposeException - questa eccezione scatta quando si ha il tentativo di compiere una qualche operazione su un oggetto non più disponibile
- OperationCancelledException - Interessante eccezione che si verifica quando viene annullata una operazione che un thread stava eseguendo
- OutOfMemoryException - Ovviamente questa eccezione viene sollevata quando non c'è più memoria sufficiente per l'esecuzione del programma
- OverflowException - si ha quando un'operazione, un cast o una conversione originano un overflow, quindi un fuori scala numerico
- PlatformNotSupportedException - evidentemente quando viene richiesta l'esecuzione di un qualche cosa che non è supportato dalla piattaforma sottostante
- RankException - si ha questa eccezione quando viene passato un array con un numero errato di dimensioni
- StackOverflowException - pericolosa eccezione che ha luogo quando sullo stack sono

allocati troppi metodi innestatì, ovvero annidati.

- SystemException - è la classe base per le eccezioni predefinite
- TimeoutException - viene sollevata questa eccezione quando termine il tempo per un certo processo
- TimeZoneNotFound - Non viene trovata la zona a cui si riferisce l'orario
- TypeInizializationExceptio n - Evidenzia problemi di inizializzazione da parte di una classe
- TypeLoadException - Ovviamente quando vi è un problema nel caricamento o nel richiamo di un tipo
- TypeUnloadedException - si verifica quando si tenta di accedere ad una classe non caricata
- UnauthorizedAccessexception - si ha questa eccezione quando il sistema operativo nega l'accesso ad una risorsa per motivi tecnici o di sicurezza
- UriFormatException - segnala una incongruenza nel formato di un indirizzo Uri
- UriTemplateMatchException - errore di confronto tra un Uri ed un templatetable relativo.

Troviamo poi le eccezioni legate ad I/O sui dati e sui files, dai nomi abbastanza significativi:

- DirectoryNotFoundException
- DriveNotFoundException - che può riferirsi ad una risorsa locale o di rete
- EndOfStreamException
- FileFormatException - quando si carica un file che non è confome ad un formato atteso
- FileNotFoundException
- InternalBufferOverflowException
- InvalidDataException - quando un certo stream contiene dati scorretti rispetto a quelli attesi
- IOException - errore di I/O generico (e quindi temibile)
- PathTooLong Exception - se viene indicato un percorso di risorsa troppo lungo
- PipeException -

come vedete ce ne è per tutti i gusti. I casi sono veramente tanti e, con buona probabilità, nella vostra vita non le incontrerete mai tutte. Può essere comunque interessante scorrere l'elenco in modo da sapere quali e quanti eventi sono presi in esame dal framework e da quali si può eventualmente derivare nuove eccezioni.

Il framework quindi è in grado di segnalarci con buona approssimazione quale è il problema che la nostra applicazione ha incontrato. Tuttavia la gestione del problema stesso è delegato al programmatore che deve decidere le azioni da intraprendere. Al nostro servizio ci sono 4 parole chiave: **try**, **catch**, **throw**, **finally**. Il particolare throw, di cui parleremo più avanti, serve per lanciare, generare l'eccezione mentre le altre 3 sono dedicate alla messa in opera di azioni di supporto. O meglio:

- **try** incapsula il codice che deve essere tenuto sotto controllo
- **catch** comprende invece il codice che serve a gestire la condizione di errore incontrata nella zona controllata da try. E' possibile mettere più blocchi catch, per gestire diversamente varie possibili eccezioni ed è anche omettibile qualora si opti per la presenza del solo finally, anche se in questo caso non si ha più gestione della eccezione

- **finally** è opzionale ma è particolarmente utile in quanto viene usato per liberare quelle risorse o compiere quelle azioni che comunque il programmatore desiderava. . fosse compiute in assenza dell'eccezione, che un errore inatteso verificatosi potrebbe avere bloccato. Va anche aggiunto che finally viene comunque eseguito sia che si verifichi l'eccezione che in caso contrario dal momento che questa istruzione dovrebbe essere legata ad azioni per così dire di pulizia. Non è pertanto opportuno quindi mettere in esso codice da eseguirsi solo in caso di eccezione perchè, a meno di porre qualche filtro, sarebbe comunque eseguito.

Stando così le cose lo scheletro operativo di base è il seguente:

```
try
{
  codice da controllare
}
catch
{
  gestione dell'eventuale eccezione
}
finally
{
  operazioni di pulizia e chiusura
}
```

Per prendere confidenza con questi aspetti proviamo ora una semplice manovra correttiva del nostro programma utilizzando proprio la parola try

C# Esempio 7.2

```
1  using System;
2  class program
3  {
4    public static void Main()
5    {
6      try
7      {
8        Console.Write("Inserisci un numero: ");
9        int x = Int32.Parse(Console.ReadLine());
10       Console.WriteLine("Il tuo numero + 1: " + (x + 1));
11     }
12     catch (Exception e)
13     {
14       Console.WriteLine("ERRORE!!!");
15       Console.WriteLine();
16       // Console.WriteLine(e);
17     }
18   }
19 }
```

Il risultato non è esaltante in apparenza, il programma si arresta comunque (e che faceva se no?) ma siamo stati in grado di inserire un messagio personalizzato e l'eccezione non provoca più un crash. Togliendo il commento alla riga 16 ricompare esplicito il messaggio di errore dal framework. Sembra poco ma è già un primo passo in avanti verso un completo controllo di eventi inattesi Per completezza introduttiva vediamo, solo sintatticamente, l'uso di finally, sia pure in un caso assolutamente non significativo

C# Esempio 7.3

```
1  using System;
2  class program
3  {
4    public static void Main()
5    {
6      try
7      {
8        Console.Write("Inserisci un numero: ");
9        int x = Int32.Parse(Console.ReadLine());
10       Console.WriteLine("Il tuo numero + 1: " + (x + 1));
11     }
12     catch (Exception e)
13     {
14       Console.WriteLine("ERRORE!!!");
15       Console.WriteLine();
16     }
17     finally
18     {
19       Console.WriteLine("Faccio pulizia");
20     }
21   }
22 }
```

In fase di esecuzione, inserendo il solito input scorretto viene segnalato l'errore coerentemente con quanto contemplato dalla presenza del catch alla riga 12 quindi viene anche eseguito il codice introdotto da finally. Interessante però è vedere cosa succede commentando le righe dalla 12 alla 16 escludendo la presenza del catch e quindi lasciando in funzione soltanto l'accoppiata try + finally. In questo caso, è facile verificare, il programma andarà in crash, perchè l'eccezione non è più gestita, ma il blocco relativo a finally sarà comunque eseguito. Questo fa capire l'importante di questa keyword in quanto permette al sistema di completare procedure che potrebbero anche essere critiche nonostante il fallimento di una applicazione. Pensiamo ad esempio al rilascio di un file, o di una periferica in generale risorse le quali vengono immediatamente liberate e rese quindi disponibili per altre eventuali richieste.

Come abbiamo detto in precedenza è possibile inserire più clausole catch. Questo ha lo scopo di catturare, evidentemente, eccezioni diverse. Vediamo l'esempio seguente e poi lo commenteremo:

C# Esempio 7.4

```
1  using System;
2
3  class Test
4  {
5  public static void Main()
6  {
7  Console.WriteLine("Questo programma esegue delle divisioni di interi");
8  Console.WriteLine("Premi CTRL + C per interrompere\n");
9  while(true)
10 {
11 try
12 {
13 Console.Write("Inserisci il dividendo: ");
14 int dividendo = int.Parse(Console.ReadLine());
15 Console.Write("Inserisci il divisore: ");
16 int divisore = int.Parse(Console.ReadLine());
```

```
17  int risultato = dividendo / divisore;
18  int resto = dividendo % divisore;
19  Console.WriteLine("Risultato: " + risultato.ToString() + " resto: " +
20  resto.ToString());
21  }
22  catch(DivideByZeroException e) // intercetta la divisione per 0
23  {
24  Console.WriteLine();
25  Console.WriteLine("Errore: divisione x zero");
26  Console.WriteLine();
27  }
28  catch(FormatException e)
29  // intercetta l'inserimento di un carattere errato, o l'invio immediato
30  {
31  Console.WriteLine();
32  Console.WriteLine("Devi inserire solo valori numerici!");
33  Console.WriteLine();
34  }
35  }
36  }
37  }
```

Questo programma calcola la divisione tra due interi. Niente di particolare ma si possono verificare in particolare due errori: l'inserimento un divisore uguale a zero e la digittazione di un valore non numerico come dividendo o come divisore. Le due eccezioni (riga 22 e riga 28) sono, come da codice, DivideByZero Exception e FormatException. I due catch inviano dei messaggi che indicano chiaramente la natura dell'errore a seconda di quale di questi si è verificata. Una precauzione che è necessario adottare è quella di costruire la catena di catch in modo adeguato. Ovvero le eccezioni di carattere più specifico vanno messe prima di quelle di carattere generico. Infatti se modificassimo la rga 21 in modo che essa esprima una eccezione di carattere generale ad es.

catch(Exception e)

il compilatore avrebbe da obiettare qualcosa come

error CS0160: A previous catch clause already catches all
exceptions of this or of a super type ('System.Exception')

Che in pratica ci avvisa che esiste già un catch che le cattura tutte. Questo perchè una volta che si verifichi un'eccezione il programma incontrerebbe come primo catch quello che fa riferimento ad una eccezione generica e sarebbe pertanto inutile specificarne di ulteriori. Il compilatore come detto ci da una mano in questo senso.

Va detto che il codice precedente non è un esempio di bella programmazione, nemmeno e soprattutto da un punto di vista logico ed è stato creato solo per fini didattici. In effetti si sarebbe potuto gestire l'input dell'utente in modo più semplice. Le eccezioni, come dice il loro nome, andrebbero utilizzate per coprire situazioni molto particolari ed imprevedibili, l'input scorretto da parte di un utente non lo è dato che abbiamo sempre l'obbligo di prevedere che possano essere commessi banali errori di digitazione, mentre ad esempio la rottura di un hard-disk è un evento già più appropriato per essere gestito da una eccezione. Insomma non bisogna abusare dell'uso delle eccezioni ma applicarle quando vale davvero la pena. Le eccezioni

nascono per sostituire molte tecniche usate per gestire situazioni critiche improvvise nel corso del funzionamento del programmi (ad esempio i famosi valori di ritorno) proponendo un modello unificato e corerente. Che però tende ad appesantire e complicare non di poco il codice se si abusa di esso.

Non ci siamo ancora occupati dell'istruzione throw che invece è veramente molto utile anche per creare delle eccezioni personalizzate. Il framework ci mette sotto controllo parecchi stati anomali ma altrettanto siamo liberi di crearci le nostre eccezioni. Vediamo dapprima come funziona il nostro throw.

C# Esempio 7.5

```
1   using System;
2
3   class Test
4   {
5     public static void Main()
6     {
7       int a = 10;
8       int b = 2;
9       int x = a / b;
10      Console.WriteLine(x);
11      throw new Exception("Ecco l'eccezione");
12    }
13  }
```

Questo programma di per se non contiene errori ma ugualmente throw, alla riga 11 lancia una eccezione generica. Il risultato è il seguente:

ovvero c'è il risultato giusto seguito dal lancio dell'eccezione. Da notare la presenza del messaggio personalizzato. La prima osservazione che si può fare è che throw può venire lanciato anche al di fuori di un blocco try in pratica ovunque nel codice. Il comportamento del CLR in casi come questi è quello di **risalire lo stack alla ricerca del metodo tra i chiamanti che possa gestire l'eccezione**. In difetto abbiamo una eccezione unhandled, come in questo caso.

Vediamo invece nell'esempio che segue il caso in cui il chiamante gestisce (riga 12) con la modalità suddetta il throw che si trova in un altro metodo (riga 20), mentre a seguire presento poi un altro esempio in cui throw viene lanciato direttamente all'interno di un try (riga 13). Come detto non è facile, almeno per me, costruire esempi davvero significativi per cui vi prego di considerare tutto il materiale di questo paragrafo puramente informativo sul funzionamento delle eccezioni.

C# Esempio 7.6

```
1  using System;
2
3  class Test
4  {
5    public static void Main()
6    {
7      Console.WriteLine("Chiamo l'eccezione");
8      try
9      {
10       LanciaEccezione();
11     }
12     catch (Exception e)
13     {
14       Console.WriteLine("Catturata");
15     }
16   }
17
18   public static void LanciaEccezione()
19   {
20     throw new Exception();
21   }
22 }
```

Ed ecco l'output:

Il trhow viene lanciato all'interno di un metodo che è stato chiamato, alla riga 9, da un altro. Il CLR risale lo stack e trova il catch che gestisce l'eccezione, catch che si trova nel chiamante che in questo caso è il Main.

C# Esempio 7.7

```
1  using System;
2  class Test
3  {
4    public static void Main()
5    {
6      try
7      {
8        int a = 10;
9        int b = 2;
10       int x = a / b;
11       Console.WriteLine(x);
12       throw new Exception("Ecco l'eccezione");
13     }
14     catch (Exception ex)
15     {
16       Console.WriteLine("E' arrivata l'eccezione");
17     }
18   }
19 }
```

Qui si ha invece una normale gestione try-catch.

Attraverso throw si ha come detto in precedenza, la possibilità di creare delle eccezioni del tutto personalizzate per gestire eventi a nostro piacimento. Con l'avviso che non tutto di questo esampio sarà chiaro e che sarà bene riguardarlo quando si saranno viste le classi e le interfaccie vediamo cosa succede:

C# Esempio 7.8

```
1   using System;
2
3   public class HaiPremutoIlTastoH: Exception // eredita da Exception
4   {
5     public HaiPremutoIlTastoH(string message):base(message)
6     {
7     }
8   }
9
10  class Test
11  {
12    public static void Main()
13    {
14      Console.Write("Inserisci una lettera: ");
15      string s = Console.ReadLine();
16      try
17      {
18  if (s == "h") throw(new HaiPremutoIlTastoH("\nNon pemere il tasto 'h'"));
19      }
20      catch (HaiPremutoIlTastoH e)
21      {
22        Console.WriteLine("Errore! {0}", e);
23      }
24    }
25  }
```

Questo programma legge un tasto inserito normalmente da tastiera. Se è diverso dalla lettera "h" non fa assolutamente nulla altrimenti viene lanciata una eccezione creata appositamente per gestire tale input. Nella pratica questo codice, assolutamente minimale, non serve ovviamente a nulla ma in via didattica ci fa capire come ereditando dalla classe base Exception (riga 3) sia possibile creare una nuova eccezione, esattamente come quelle fornite dal framework che viene gestita con le stesse identiche modalità di quelle normali. La riga 22 lancia la nuova eccezione e la 24 la gestisce. Oltre che da Exception possiamo derivare da una qualsiasi delle altre eccezioni, se ci pare più appropriato. La derivazione sarà introdotta insieme alle classi.

Viene d'obbligo ora presentare proprio la classe madre di tutte le eccezioni, Exception; si tratta di una classe importante e abbastanza semplice nell'uso, lo abbiamo visto nell'esempio precedente alla riga 3. Essa espone alcune proprietà e alcuni metodi pubblici, quindi liberamente utilizzabili che è bene conoscere e che solo elencati qui di seguito:

Proprieta':

- Data restituisce una collezione di coppie chiave-valore con informazioni addizionali sull'eccezione

- HelpLink definisce o restiuisce il link ad un file di help

- HResult definisce o restituisce un codice numerico associato alla eccezione

- InnerException restiutuisce l'eccezione che ha causato la corrente eccezione. Se pratica se una eccezione corrente è stata lanciata da un'altra precedente sarà quest'ultima che la proprietà InnerException dell'eccezione corrente restituisce.

- Message espone un messaggio descrittivo dell'eccezione, come nell'esempio precedente

- Source definisce o restituisce il nome dell'applicazione o dell'oggetto causa dell'eccezione

- StackTrace restituisce una stringa in cui sono rappresentati gli strati dello stack al momento dell'eccezione

- TargetSite ritorna il metodo che ha generato l'eccezione.

tra i metodi invece, oltre a quelli eraditati di default da Object (Equals, Finalize, MemberwiseClone, GetHasCode, ToString, GetType) troviamo GetBaseException (ritorna la eccezione-radice di una o più eccezioni) e GetObjectData (fornisce ulteriori informazioni, la vedremo all'opera in un esempio) che sono di uso più raro rispetto alle proprietà.

CHECKED e UNCHECKED

A contatto con le eccezioni ci sono anche le due istruzioni **checked** e **unchecked** che aiutano ad intercettare potenziali fonti di problemi. Un esempio tipico è il seguente, già visto:

C# Esempio 7-9

```
1  using System;
2
3  class Program
4  {
5    public static void Main()
6    {
7      byte b = 100;
8      b += 200;
9      Console.WriteLine(b);
10   }
11 }
```

Questo programma espone a video il numero 44 che è il risultato dell'operazione 100 + 200 su un byte evidentemente scorretto, come e perchè lo abbaimo già spiegato altrove. Fin che si tratta solo di stampare a video un numero non è poi così grave ma in altre situazioni questo risultato e comunque questo tipo di anomalia può originare seri problemi. Esistono varie soluzioni in merito una possibile tra queste consiste nel porre sotto l'occhio di una sentinella

quelle porzioni di codice che potrebbero orginare situazioni pericolose; almeno per quanto concerne i range numerici abbiamo una soluzione rapida ed abbastanza efficace.
Modifichiamo il codice come segue:

C# Esempio 7.10

```
1   using System;
2
3   class Program
4   {
5     public static void Main()
6     {
7       checked
8       {
9         byte b = 100;
10        b += 200;
11        Console.WriteLine(b);
12      }
13    }
14  }
```

facendo girare il programma così modificato viene sollevata una eccezione. Questo è determinato proprio dalla presenza di quel checked alla riga 7 il cui scopo è quello di abilitare il check degli overflow nelle operazioni sugli interi o per dir meglio effettua un controllo sui potenziali overflow che possono avvenire sulle variabili di destinazione compatibilmente col loro tipo. Nel nostro esempio infatti il programma si accorge di un overflow sul tipo byte e lo segnala. Lo stesso risultato, meno elegantemente e anche con qualche prezzo da pagare dal punto di vista prestazionale lo si sarebbe potuto ragiungere compilando il codice dell'esempio 7.9 con la direttiva **/checked+**

csc /checked+ nomefile.cs

che aggiunge un controllo sugli overflow. In questo caso se si fa uso della parola **unchecked** la sezione che quest'ultima comprende non sarà oggetto di check anche nel caso in cui sia stata impartita la direttiva /checked+. Quindi unchecked ha il solo scopo di prevenire il check sugli overflow nella sezione di codice che esso controlla.