Eccezioni

Ambiente: .NET 2.0/C# 2.0

Anno 2015/2016

Indice generale

1 Introduzione

Nel linguaggio comune il termine *eccezione* denota qualcosa che devia da un andamento regolare e prevedibile. Nell'ambito della programmazione questo termine assume un significato più stringente:

> ***evento che interrompe il regolare svolgimento di un processo e che dev'essere gestito perché il processo possa continuare.***

Dal punto di vista del linguaggio, invece, un'eccezione è un:

> ***oggetto che viene creato in risposta all'impossibilità di eseguire una certa operazione.***

In questo caso il programma "scatena" (o "solleva") un'eccezione. Il tipo dell'eccezione dipende dalla natura dell'operazione che il programma non è stato in grado di eseguire.

.NET definisce svariati tipi di eccezioni, organizzati in una gerarchia. In cima alla gerarchia ci sono i tipi di natura generale, da cui derivano tutti gli altri, i quali rappresentano eccezioni più specializzate, sollevate in risposta al fallimento di operazioni ben precise.

1.1 Gestire le eccezioni

Se viene sollevata un'eccezione, l'esecuzione del programma è interrotta e possono presentarsi due casi:

1. l'eccezione viene gestita: esiste cioè del codice che "prende in mano la situazione", possibilmente consentendo al programma di continuare la propria esecuzione;
2. l'eccezione non viene gestita: il programma *crasha*.

> **Gestione delle eccezioni Visual Studio**
>
> Durante lo sviluppo del programma le eccezioni sono innanzitutto gestite da Visual Studio. Il comportamento di VS dipende dal tipo di applicazione e di esecuzione (con o senza *debug*).
>
> In modalità *debug* il programma viene sospeso sull'istruzione che ha prodotto l'eccezione.

1.2 Programmi che richiedono la gestione delle eccezioni

Alcuni programmi possono andare in *crash* soltanto:

1. per un'errata implementazione. In questo caso l'interruzione è opportuna, poiché fornisce informazioni sul codice mal funzionante;
2. per il mancato rispetto di determinati vincoli sui dati di ingresso. Ad esempio: un programma richiede un valore numerico e ne riceve uno alfanumerico.

I programmi realistici, d'altra parte, richiedono il rispetto di requisiti molto più stringenti, poiché:

1. accedono al *file system*:
2. accedono al registro di sistema;
3. condividono risorse con altri programmi;
4. accedono a database;
5. accedono a delle periferiche, ad esempio la stampante.
6. comunicano con altri programmi o servizi, che risiedono nello stesso computer o in computer remoti.

Le operazioni suddette possono fallire e scatenare un'eccezione. Un programma ben progettato deve prendere in considerazione questa possibilità, tentando di gestirla in modo sensato, eventualmente terminando l'esecuzione in modo da evitare perdite di dati.

2 Catturare le eccezioni: costrutto “try... catch”

La gestione delle eccezioni passa innanzitutto dalla loro “cattura”. Ciò avviene utilizzando il costrutto **try...catch**:

```
try
{
        // codice che si vuole proteggere dalle eccezioni
}
catch(tipo-eccezione nome-variabile_opz)
{
        // codice che gestisce l’eccezione
}
```

Il costrutto si divide in due blocchi:

1. il blocco **try** (tenta), racchiude il codice che si vuole “proteggere” dalle eccezioni. Qualsiasi eccezione si verifichi all’interno di questo blocco passa all’esame del blocco **catch** (cattura);
2. questo stabilisce un confronto tra il tipo di eccezione e il tipo specificato tra parentesi. Se i due tipi sono compatibili, allora il blocco viene eseguito, altrimenti l’eccezione viene lasciata passare.

Se l'eccezione viene gestita (esecuzione del blocco **catch**), l’esecuzione continuerà con la prima istruzione che segue il costrutto. La stessa cosa accade se non viene sollevata nessuna eccezione.

Se invece l'eccezione è ignorata dal blocco **catch**, viene ricercato un eventuale costrutto **try...catch** di livello superiore e il procedimento si ripete, finché l’eccezione viene gestita, oppure il programma si interrompe.

2.1 Gestione eccezioni: verificare l'input dell'utente

L'esempio che segue mostra come proteggere il programma da un input non valido:

```
List<Persona> anagrafica = new List<Persona>();
private void btnInserisci_Click(object sender, EventArgs e)
{
    try
    {
        Persona p = new Persona(txtNome.Text,
                                double.Parse(txtPeso.Text),
                                double.Parse(txtAltezza.Text));
        anagrafica.Add(p);
    }
    catch (FormatException)
    {
        MessageBox.Show("Dati non validi");
    }
}
```

L'utente inserisce i dati di una persona, che successivamente viene inserita in un elenco. Se peso ed età non sono numerici, viene sollevata l'eccezione **FormatException** dal metodo **int.Parse()** e l'esecuzione passa al blocco **catch**

Nota bene: se l'eccezione non è del tipo **FormatExcpetion**, il blocco **catch** la lascia passare e il programma va in *crash*.

2.2 Codice ammesso in un blocco "try"

Non esistono vincoli sulla natura del codice contenuto in un blocco **try**. Inoltre, i metodi invocati all'interno del blocco risultano a loro volta protetti:

```
private void btnInserisci_Click(object sender, EventArgs e)
{
    try
    {
        AggiungiPersona(txtNome.Text, txtPeso.txt, txtAltezza.txt);
    }
    catch (FormatException)
    {
        MessageBox.Show("Dati non validi");
    }
}

private void AggiungiPersona(string nome, string peso, string altezza)
{
    Persona p = new Persona(nome, double.Parse(peso), double.Parse(altezza));
    anagrafica.Add(p);
}
```

Un'eventuale eccezione sollevata da **AggiungiPersona()** viene gestita, poiché il metodo è chiamato all'interno di un blocco protetto

2.3 Proteggere il codice da più eccezioni

L'esempio precedente suppone che il codice possa scatenare solo una **FormatException**; qualunque altro tipo di eccezione *crasha* il programma. Ma mediante un **try...catch** è possibile proteggere il codice da più tipi di eccezioni.

2.3.1 Catturare più tipi di eccezione

Il costrutto **try...catch** ammette l'esistenza di più blocchi **catch**, ognuno dei quali specifica il tipo di eccezione che è in grado di catturare:

```
try {...}

catch(tipo-eccezione1) {...}

catch(tipo-eccezione2) {...}

catch(tipo-eccezionen) {...}
```

In caso di più blocchi **catch**, questi vengono esaminati nell'ordine in cui sono scritti, alla ricerca di uno che specifichi il tipo di eccezione corrispondente a quella scatenata.

2.3.2 Ordine di definizione dei blocchi "catch"

Quando si specificano più blocchi **catch** occorre:

> ***ordinarli in base al tipo di eccezione che specificano, dal più specializzato al più generico.***

Si consideri il seguente codice:

```
try
{
    Persona p = new Persona(txtNome.Text,
                            double.Parse(txtPeso.Text),
                            double.Parse(txtAltezza.Text));
    anagrafica.Add(p);
}
catch (FormatException)   // gestione specializzata dell'eccezione
{
    MessageBox.Show("Dati non validi");
}
catch (Exception)         // gestione generica
{
    MessageBox.Show("Errore sconosciuto");
}
```

Il primo blocco **catch** cattura soltanto eccezioni di tipo **FormatExecption**; il secondo blocco **catch** le cattura tutte (il tipo **Exception** è compatibile con tutte le eccezioni).

Lo scopo è quello di gestire in modo specifico le eccezioni **FormatException**, ma catturare comunque anche le altre.

2.4 Utilizzare l'oggetto eccezione: proprietà "Message"

Lo scatenarsi di un'eccezione implica la creazione di un oggetto, al quale si può accedere nel blocco **catch**.

```
try
{
    // codice protetto
}
catch (FormatException e)
{
    // gestione eccezione
}
```

> **Nome dell'oggetto eccezione**
>
> È possibile scegliere un nome qualsiasi. Per convenzione si usa **e** o **ex**.

Gli oggetti eccezione hanno diverse proprietà e metodi, che in parte dipendono dal tipo dell'oggetto. La proprietà più utilizzata è **Message**, che informa sulla causa che ha prodotto l'eccezione.

Il codice seguente usa la proprietà **Message** dell'oggetto `e` per informare l'utente sulla causa dell'errore.

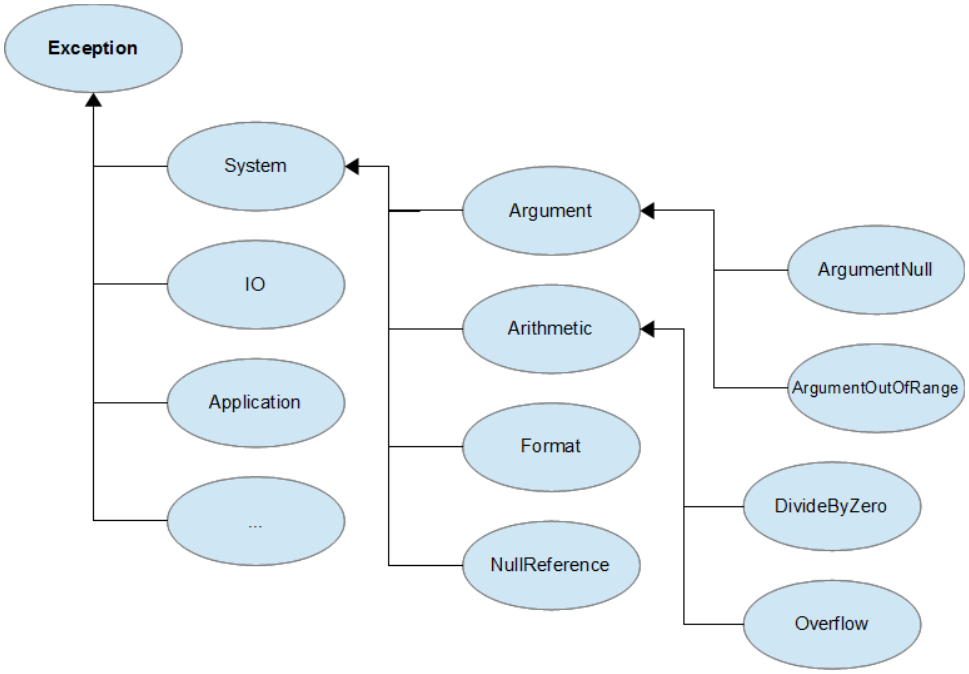
```
try
{
    // … creazione Persona e inserimento in lista
}
catch (FormatException e)
{
    MessageBox.Show("Dati non validi\n" + e.Message);
}
```

3 Tipi di eccezioni

Finora si è parlato in termini generali di *tipo di eccezione*; nella realtà:

1. i *tipi di eccezione* sono classi organizzate in una gerarchia, alla base della quale sta la classe **Exception**.
   (Tutti i tipi derivano, direttamente o indirettamente, dalla classe **Exception**.
2. I nomi dei tipi hanno come suffisso finale “Exception”.

Di seguito è mostrato uno schema che rappresenta un sotto insieme della gerarchia. (Per semplicità il suffisso “Exception” è stato omesso.)



Le relazioni mostrate dallo schema stabiliscono ad esempio che:

1. **FormatException** è un tipo specializzato di **SystemException**;
2. **DivideByZeroException** è un tipo specializzato di **ArithmeticException**.

I tipi collocati in fondo della gerarchia sono molto specializzati e dunque corrispondono al fallimento di operazioni specifiche. Salendo nella gerarchia si trovano tipi sempre più generali, che corrispondono al fallimento di una intera categoria di operazioni (ad esempio un generico errore aritmetico).

3.1 Corripondenza tra tipo eccezione e tipo specificato nel blocco “catch”

Per essere catturata da un blocco **catch**, l'eccezione deve corrispondere al tipo specificato. Con questo si intende che

> ***il tipo dell'eccezione sollevata deve essere uguale al tipo specificato oppure derivato da esso.***

Ad esempio, **FormatException** sarà catturata da un **catch** che specifichi **SystemException** o **FormatException**, ma non da uno che specifichi **ArithmeticException**.

3.2 Introduzione alle eccezioni più comuni

Segue un breve esame delle eccezioni più comuni.

3.2.1 ArgumentNullException

> ***viene sollevata se un metodo riceve un riferimento null come argomento, laddove è espressamente richiesto un valore non nullo.***

Molti metodi non accettano che gli argomenti abbiano il valore **null**. Quando ciò accade viene sollevata l'eccezione.

Il seguente codice dichiara un vettore globale (il cui valore predefinito è **null**) e lo passa al metodo **Clear()** della classe **Array**:

```
static int[] vettore;                       // la variabile vettore è null!
static void Main(string[] args)
{
    Array.Clear(vettore, 0, 10);            // eccezione: vettore == null!
}
```

3.2.2 ArgumentOutOfRangeException

> ***viene sollevata se un metodo riceve un argomento il cui valore cade al di fuori dell'intervallo consentito.***

Alcuni metodi sollevano questa eccezione se gli argomenti cadono al di fuori di un certo intervallo. È questo il caso del metodo statico **Copy()** della classe **Array**:

```
int[] v1 = { 1, 2, 3, 4 };
int[] v2 = new int[v1.Length];
Array.Copy(v1, v2, -1);             // eccezione: "n° elementi" == -1!
```

3.2.3 ArgumentException

> ***viene sollevata se un metodo riceve un argomento il cui valore è inadeguato per il compito che deve svolgere.***

Questo tipo copre una gamma di casi più ampia dei tipi precedenti. Ad esempio, il metodo **Copy()** della classe **Array** solleva questa eccezione se il valore del terzo argomento eccede il numero degli elementi del vettore da copiare.

```
int[] v1 = { 1, 2, 3, 4 };
int[] v2 = new int[v1.Length];
Array.Copy(v1, v2, 5);              // eccezione: "n° elementi" >= v1.Lenth
```

3.2.4 FormatException

> ***viene sollevata durante la conversione di una stringa quando questa non rappresenta un valore ammissibile per il tipo nel quale viene convertita.***

Il caso tipico si presenta quando si tenta di convertire una stringa in numero (**int**, **double**, etc):

```
double d1 = double.Parse("1000");          // ok
double d2 = double.Parse("1000,,0");       // eccezione!
```

Ma ciò vale in generale. Ad esempio, il codice seguente mostra una conversione da stringa a **bool**:

```
bool b1 = bool.Parse("false");
bool b2 = bool.Parse("falsse");            // eccezione!
```

È importante non confondere questo tipo di eccezione con **InvalidCastException**. In entrambi i casi si ha un tentativo fallito di conversione, ma per motivi diversi.

3.2.5 InvalidOperationException

> ***viene sollevata quando un metodo non è in grado di svolgere il proprio compito per un motivo che non riguarda in modo specifico gli argomenti ricevuti.***

Questa eccezione è una risposta generica al fallimento di una determinata operazione.

Ad esempio, l'eccezione viene sollevata quando si tenta di ordinare un vettore di **object** i cui elementi non sono tutti dello stesso tipo:

```
object[] vettoreGenerico = { "Fermi", 10, "Einstein" };   // è ammesso
Array.Sort(vettoreGenerico);                              // eccezione!
```

3.2.6 IndexOutOfRangeException

> ***viene sollevata quando si tenta di accedere a un elemento di un array mediante un indice il cui valore è fuori dall'intervallo consentito.***

Lo scenario più comune si presenta nei cicli **for**:

```
for (int i = 0; i <= vettore.Length; i++)  // dovrebbe essere "<"
    somma += vettore[i];                   // eccezione!
```

3.2.7 IOException

> ***viene sollevata quando è impossibile eseguire un'operazione su un file o una cartella.***

Dal tipo **IOException** derivano quattro tipi specializzati, che coprono situazioni specifiche:

1. **DirectoryNotFoundException**: riferimento a una cartella inesistente;
2. **EndOfStreamException:** si è tentato di leggere oltre la fine di uno stream;

3. **FileLoadException:** è impossibile leggere il file (ad esempio, perché il formato del contenuto non corrisponde a quello richiesto);
4. **FileNotFoundException**: file non trovato.

Esistono delle situazioni nelle quali l'eccezione sollevata è **IOException**, poiché non esiste un tipo specifico appropriato. Ad esempio, è questo il caso della violazione dell'accesso esclusivo a un file.

Nel gestire queste eccezioni si dovrebbe specificare prima un blocco **catch** per catturare il tipo specializzato, seguito da uno per il tipo **IOException**. Ad esempio:

```
Console.WriteLine("Immetti il nome del file:");
string nomeFile = Console.ReadLine();
try
{
    FileStream fs = new FileStream(nomeFile, FileMode.Open);
    // ... qui viene elaborato il file
}
catch(FileNotFoundException)        // gestione specializzata dell'eccezione
{
    Console.WriteLine("File non trovato");
}
catch (IOException)                 // gestione generica dell'eccezione
{
    Console.WriteLine("Impossibile elaborare il file ");
}
```

3.3 Eccezioni provocate dall'accesso a risorse esterne

Il verificarsi di **IOException** e tipi analoghi non implica necessariamente un errore nel programma. Infatti, l'accesso a qualsiasi risorsa esterna è sempre soggetto alla possibilità di un fallimento.

Sono proprio questi i casi che devono essere gestiti, mettendo in atto un comportamento appropriato:

1. Ritentare l'operazione. Ad esempio, il drive USB non è inserito: si informa l'utente di inserirlo e di ripetere l'operazione di salvataggio.
2. Continuare il programma. Ad esempio, è impossibile connettersi a un computer remoto: si informa l'utente di ritentare dopo aver eseguito alcune verifiche sul corretto funzionamento della rete.
3. Terminare il programma. Ad esempio, è fallito il tentativo di caricare in memoria un file cruciale per la corretta esecuzione del programma: l'utente viene informato del problema e il programma viene terminato.

4 Garantire l'esecuzione del codice

Esistono situazioni nelle quali è necessario proteggere il codice dalle eccezioni, ma è opportuno che la loro gestione avvenga in un altro punto del programma.

Si immagini il seguente scenario. Un metodo legge un file e lo carica in un vettore. Durante la lettura si verifica un'eccezione e dunque il metodo termina anticipatamente. Il codice chiamante intercetta l'eccezione con un **try...catch**, ma ciò lascia un problema: il file è stato aperto ma non è stato chiuso.

In sostanza: il metodo deve garantire la chiusura del file, senza però catturare le eccezioni, poiché la loro gestione spetta al codice chiamante.

A questo scopo è possibile usare il costrutto **try...finally**.

4.1 Costrutto “try...finally”

Il costrutto **try...finally** garantisce che un blocco di codice venga eseguito a prescindere da eventuali eccezioni. Si presenta nella seguente forma:

```
try
{
    // codice protetto
}
finally
{
    // codice di finalizzazione: viene sempre eseguito!
}
```

In caso di eccezione l'esecuzione passa immediatamente al blocco **finally** (*finalizzatore*). L’eccezione viene comunque considerata non gestita, e dunque comincia la ricerca di un **try...catch**.

Se non ci sono eccezioni, il blocco **finally** viene eseguito dopo il completamento del blocco **try**.

Il blocco **finally** viene chiamato di *finalizzatore* poiché il suo scopo è in genere quello di eseguire le operazioni che terminano un procedimento, nella fattispecie rilasciare risorse esterne al programma.

4.1.1 Esempio d'uso del costrutto “try...finally”

Segue un metodo restituisce le righe di un file di testo, garantendo la chiusura del file:

```
static string[] CaricaTesto(string nomeFile)
{
    List<string> testo = new List<string>();
    StreamReader sr = new StreamReader(nomeFile);
    try
    {
        while(sr.EndOfStream == false)
        {
            string linea = sr.ReadLine();
```

```
            testo.Add(linea);
        }
        return testo.ToArray();
    }
    finally
    {
        if (sr != null)
            sr.Close();      // questa istruzione viene eseguita in ogni caso
    }
}
```

Nota bene: l'operazione di apertura del file non è protetta; infatti il metodo deve garantire soltanto che il file venga chiuso.

4.2 Istruzione “return” e blocco “finally”

Esistono due questioni legate all’istruzione **return** e al blocco **finally:**

- **istruzione return nel blocco try**: il metodo viene terminato, *ma soltanto dopo che è stato eseguito il blocco **finally***.
- **Istruzione return nel blocco finally**: non è ammessa! Il motivo è semplice: deve essere garantita l'esecuzione di tutto il codice di finalizzazione.

4.3 Costrutto “using”

Il costrutto **try...finally** è così importante che il linguaggio C# fornisce una parola chiave apposita per implementarlo: **using**.

Di seguito è mostrata la corrispondenza tra **using** e **try...finally**:

```
// creazione oggetto
try
{
    // uso oggetto
}
finally  // finalizzatore
{

}
```

```
using (creazione oggetto)
{
    // uso oggetto
}
```

using può essere usato soltanto con oggetti che richiedono la fase di *finalizzazione*, fase che viene eseguita automaticamente.

> ### Oggetti “finalizzabili”
>
> In pratica **using** può essere usato soltanto con oggetti che implementano l'interfaccia **IDisposable**.
>
> Questi utilizzano internamente delle risorse del sistema operativo, risorse che devono essere rilasciate.

Il metodo precedente può essere implementato nel seguente modo:

```
static string[] CaricaTesto2(string nomeFile)
{
    List<string> testo = new List<string>();
    using (StreamReader sr = new StreamReader(nomeFile))
    {

        while (sr.EndOfStream == false)
        {
            string linea = sr.ReadLine();

            testo.Add(linea);
        }
        return testo.ToArray();
    }
}
```

5 Gestione avanzata delle eccezioni

Di seguito saranno esaminati alcuni scenari avanzati e considerate alcune problematiche connesse all'uso dei costrutti **try...catch...finally**.

5.1 Eccezioni sollevate nei blocchi “catch” e “finally”

Non esiste niente di speciale nel codice contenuto nei blocchi **catch** e **finally** e dunque anche al loro interno può scatenarsi un’eccezione.

Di seguito non ci occuperemo del modo in cui può essere affrontata una simile evenienza, ma di comprendere cosa avviene se si verifica.

5.1.1 Eccezione nel blocco “catch”

Se si verifica un'eccezione nel blocco **catch**, l'esecuzione abbandona immediatamente il blocco e inizia la ricerca di un blocco **try** in grado di catturarla.

La nuova eccezione si sostituisce a quella in corso, che viene persa.

5.1.2 Eccezione sollevata nel blocco “finally”

Possono verificarsi due scenari:

1. ***non c’è ancora alcuna eccezione in corso*:** l'esecuzione abbandona il blocco **finally** e comincia la ricerca di un blocco **try** in grado di catturare l’eccezione;
2. ***c’è già un’eccezione in corso*:** l'eccezione sollevata si sostituisce a quella in corso; il flusso di esecuzione abbandona il blocco **finally** e comincia la ricerca di un blocco **try** in grado di catturare la nuova eccezione.
   (Anche in questo caso l’eccezione originale viene persa.)

5.2 Gestire parzialmente l’eccezione e rilanciarla: “throw”

In alcuni scenari si desidera intercettare l'eccezione allo scopo di eseguire determinate operazioni, ma senza bloccarla, in modo che il codice chiamante possa gestirla.

In questo caso, il blocco **catch** ha lo scopo di catturare l'eccezione, eseguire del codice e “rilanciarla” come se in realtà non fosse stata intercettata:

```
try
{

}
catch (Exception e)
{
    // esegui determinate operazioni
    throw;      // rilancia l'eccezione originale
}
```

Nota bene: **throw** (lancia) non specifica il tipo di eccezione da scatenare, poiché ha lo scopo di rilanciare quella appena catturata.

6 Sollevare un'eccezione

La generazione di un'eccezione è estremamente semplice e si risolve nell'utilizzo della parola chiave **throw** (lancia), seguita dal tipo di eccezione da scatenare:

```
throw new tipo-eccezione(lista-argomenti_opz)_opz
```

È così che i metodi sollevano le eccezioni. Ad esempio, il metodo **Array.Clear()** contiene il seguente codice:

```
static void Clear(Array array, int index, int length)
{
  if (array == null) // verifica l'argomento
      throw new ArgumentNullException("Array cannot be null.\n Parameter name: array");
    ...
}
```

Se il valore dell'argomento `array` è nullo viene generata **ArgumentNullException**. Ciò si traduce nell'immediata terminazione del metodo e nella ricerca di un blocco **try** in grado di catturare l'eccezione.

6.1 Uso dell'istruzione "throw"

Nella programmazione esiste un presupposto fondamentale: se un metodo non è in grado di compiere il proprio lavoro, l'esecuzione non può procedere normalmente.

Ad esempio, un metodo che risolve un'equazione di 2° grado dovrebbe verificare il segno del termine sotto radice; se è negativo dovrebbe lanciare un eccezione:

```
static void RisolviEquazione(double a, double b, double c,
                    out double x1, out double x2)
{
    double delta = b*b - 4*a*c;
    if (delta < 0)
        throw new InvalidOperationException("Il delta è negativo\nImpossibile...");

    // calcolo delle due radici
}
```

Nota bene: nel creare l'eccezione fornisco un messaggio che ne indica la causa.

Il messaggio non influenza il meccanismo di gestione delle eccezioni, ma è lo stesso molto utile, poiché aiuta il lavoro del programmatore e rappresenta un'informazione per l'utente finale nel caso in cui l'eccezione non venga gestita dal programma.

7 Eccezioni definite dal programmatore

Le eccezioni predefinite rappresentano una risposta al fallimento di determinate classi di operazioni. D'altra parte, può essere utile caratterizzare l'operazione fallita in relazione al significato che possiede per l'applicazione. A questo scopo è possibile definire nuovi tipi di eccezione: è sufficiente implementare una classe che derivi da uno dei tipi di predefiniti.

Poiché lo scopo è quello di definire tipi che abbiano un significato preciso per il programma, non è opportuno "mescolare" i nuovi tipi con quelli predefiniti. A questo scopo esiste un tipo di eccezione, **ApplicationException**, da usare come classe base per la definizione di nuovi tipi.

7.1 Derivare un tipo da "ApplicationException"

Supponiamo che l'applicazione utilizzi un file di configurazione. Vogliamo definire un nuovo tipo di eccezione che identifichi il fallimento del caricamento del file:

```
public class LoadConfigException : ApplicationException
{
}
```

> **Nomi delle classi eccezione**
>
> Il suffisso "Exception" nel nome non è obbligatorio, ma rappresenta una convenzione universalmente condivisa.

7.1.1 Esempio: uso di "LoadConfigException"

Il metodo **LoadConfig()** carica il file di configurazione; se l'operazione fallisce viene sollevata **LoadConfigException**:

```
static void LoadConfig(string fileName)
{
    try
    {
        using (var sr = new StreamReader(fileName))  //garantisce la finalizzazione
        {
            string line = sr.ReadLine();
            while (line != null)
            {
                //elabora riga
                line = sr.ReadLine();
            }
        }
    }
    catch(IOException)
    {
        throw new LoadConfigException();  // rilancia la nuova eccezione
    }
}
```

Il metodo cattura le eccezioni di IO durante la lettura del file, e al loro posto solleva **LoadConfigException**.

Il nuovo tipo fornisce un significato preciso all'evento che si è verificato: il programma non è riuscito a caricare il file di configurazione.

7.1.2 Accettare un messaggio

I nuovi tipi di eccezione possono fornire un costruttore che accetti un messaggio informativo.

```
public class LoadConfigException : ApplicationException
{
    public LoadConfigException(){ }
    public LoadConfigException(string msg) : base(msg) { }
}
```

Nota bene: il costruttore si limita a passare l'argomento al costruttore della classe base.

Dopo questa modifica, l'eccezione può essere qualificata con un messaggio:

```
...
catch(IOException)
{
    throw new LoadConfigException("Impossibile caricare file di configurazione");
}
```

7.1.3 Memorizzare l'eccezione originale

In **LoadConfig()** l'eccezione **LoadConfigException** sostituisce quella originale, che dunque viene persa. Ma è utile preservare l'eccezione originale, poiché fornisce informazioni sull'errore che si è verificato: file inesistente, lettura oltre la fine del file, eccetera.

A questo scopo le eccezioni definiscono la proprietà **InnerException**, che memorizza un riferimento all'eccezione originale (sempre che esista). Per valorizzare la proprietà esiste un costruttore apposito:

```
public class LoadConfigException : ApplicationException
{
    public LoadConfigException(){ }
    public LoadConfigException(string msg) : base(msg) { }
    public LoadConfigException(string msg, Exception e) : base(msg, e) { }
}
```

Ecco come sollevare l'eccezione mantenendo il riferimento a quella originale:

```
...
catch(IOException e)
{
    throw new LoadConfigException("Impossibile caricare file di configurazione", e);
}
```

Nota bene: nel blocco **catch** è necessario definire non soltanto il tipo di eccezione, ma

anche la variabile, che sarà passata al costruttore durante la creazione della nuova eccezione.

Di seguito viene mostrato come usare il riferimento all'eccezione originale:

```
static void Main(string[] args)
{
    try
    {
        LoadConfig("config.ini");
    }
    catch (LoadConfigException e)          // usa l'oggetto eccezione
    {
        Console.WriteLine(e.Message);
        Console.WriteLine("E' stato riscontrato il seguente problema:");
        Console.WriteLine(e.InnerException.Message);
    }
}
```

Nota bene: qui si da per scontato che **InnerException** sia sempre diverso da **null**. In generale, però, non è un presupposto valido.