Gestione delle eccezioni

Eccezioni

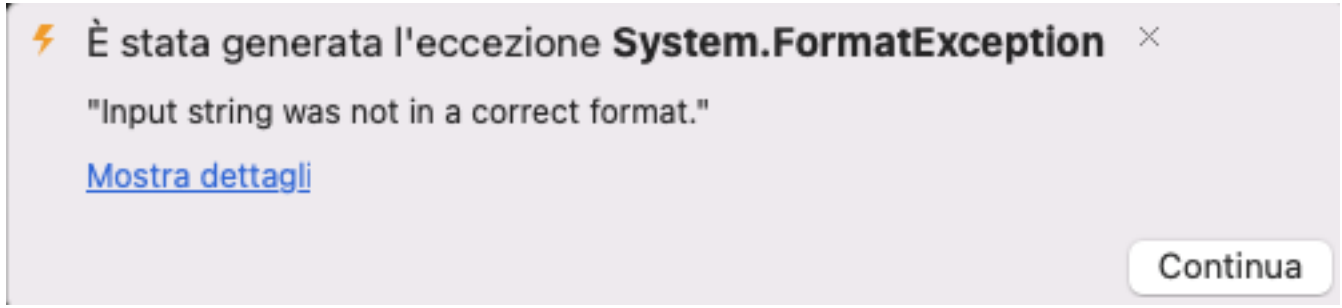
Durante l'esecuzione di un programma possono accadere situazioni non previste. Per esempio:

- Una divisione per zero
- L'accesso ad un elemento dell'array che non esiste
- Errore di conversione tra tipi
- Mancanza di connettività di rete
- Timeout durante una operazione lunga
- …

.NET permette di gestire questi casi con le Eccezioni. Nel momento in cui si verifica un errore viene generata una eccezione. Se nessuno si prende in carico di gestirla il programma termina con errore.

Eccezioni

```
class Program
{
    static void Main(string[] args)
    {
        int n = Convert.ToInt32("Questo non è un numero");
    }
}
```



Eccezioni

.NET ci permette di "intercettare" una eccezione per poterla gestire evitando che l'errore termini l'applicazione. Le istruzioni che potenzialmente possono generare errore vanno incluse in un blocco try { } seguito da un blocco catch { } nel quale inserire le istruzioni da eseguire in caso di errore.

Le eccezioni sono oggetti. Ne esistono di vario tipo ma tutte ereditano dalla classe System.Exception.

Nel blocco catch è possibile specificare quale eccezione gestire.

```
class Program
{
    static void Main(string[] args)
    {
      int n;
      try
      {
          n = Convert.ToInt32("Questo non è un numero");
      }
      catch(FormatException e)
      {
          Console.WriteLine(e.ToString());
      }
    }
}
```

Eccezioni

A fronte di un blocco try è possibile specificare più blocchi catch per la gestione di più tipi di eccezioni. Devono però essere scritti in un ordine che va dal più specifico al più generico.

I blocchi catch verranno quindi valutati in ordine e verrà eseguito il primo che gestisce una eccezione compatibile.

```
class Program
{
    static void Main(string[] args)
    {
      int n;
      try
      {
          n = Convert.ToInt32("Questo non è un numero");
      }
      catch(FormatException e)
      {
          Console.WriteLine(e.ToString());
      }
      catch(Exception e)
      {
          Console.WriteLine(e.ToString());
      }
    }
}
```

Eccezioni

Le eccezioni "risalgono" dal punto in cui sono generate ai metodi superiori fino ad arrivare al main. Possono essere gestite in qualsiasi punto e solo se arrivano al main non gestire causano il termina con errore.

Eccezioni

Il flusso delle eccezioni può essere bloccato in qualsiasi punto.

```
static void Main(string[] args)
{
    Metodo1();
}
```

Flusso di esecuzione

```
static void Metodo1(string[] args)
{
  try
  {
    Metodo2();
  }
  catch(Exception)
  {
    Console.WriteLine("Si è verificato un errore");
  }
}
```

Flusso delle eccezioni

```
static void Metodo1(string[] args)
{
    n = Convert.ToInt32("Questo non è un numero");
}
```

Eccezioni

Nella gestione delle eccezioni è possibile specificare un blocco di codice eseguito in ogni caso, indipendentemente dal generarsi i meno dell'eccezione. Il blocco è specificato con la parola chiave finally ed è tipicamente usato per rilasciare risorse non gestite dal Garbage Collector, come per esempio le risorse di rete

```
static void Metodo1(string[] args)
{
  try
  {
    Metodo2();
  }
  catch(Exception)
  {
    Console.WriteLine("Si è verificato un errore");
  }
  finally
  {
    //Operazioni di pulizia
  }
}
```

Eccezioni

Possiamo manualmente generare e sollevare eccezioni tramite la parola chiave throw, dopo aver creato del tipo opportuno.

Eccezioni

E' possibile creare eccezioni personalizzate per meglio modellare le tipologie di errore che l'applicazione potrebbe generare. E' sufficiente creare una nuova classe che estende la classe System.Exception o una sua classe derivata.

E' consuetudine nominare le eccezioni ponendo la parola Exception alla fine del nome.

```
public class CustomException : Exception
{
    public CustomException() : base()
    {

    }

    public CustomException(string message) : base(message)
    {

    }
}
```

Programmazione generica

Programmazione generica

In applicazioni reali sono frequenti casi in cui abbiamo proliferazione di codice "identico" che lavora però su tipi di dati differenti. E' il caso tipico delle strutture dati.

```
//Libreria in sola aggiunta di Libri
public class Libreria
{
    private DateTime _dataCreazione;
    private List<Libro> _dati;

    public int Count
    {
        get
        {
            return _dati.Count;
        }
    }

    public Libreria()
    {
        _dataCreazione = DateTime.Now;
        _dati = new List<Libro>();
    }

    public void Add(Libro l)
    {
        _dati.Add(l);
    }
}
```



```
//Registro in sola aggiunta di Libri
public class Registro
{
    private DateTime _dataCreazione;
    private List<Studente> _dati;

    public int Count
    {
        get
        {
            return _dati.Count;
        }
    }

    public Registro()
    {
        _dataCreazione = DateTime.Now;
        _dati = new List<Studente>();
    }

    public void Add(Studente s)
    {
        _dati.Add(t);
    }
}
```

Programmazione generica

Con la programmazione generica è possibile scrivere codice facendo riferimento a tipi di dati generici (astratti, non reali), specificando il tipo di dato reale solamente in fase di utilizzo

```
//Registro in sola aggiunta di Libri
public class ListaSolaAggiunta<T>
{
    private DateTime _dataCreazione;
    private List<T> _dati;

    public int Count
    {
        get
        {
            return _dati.Count;
        }
    }

    public ListaSolaAggiunta()
    {
        _dataCreazione = DateTime.Now;
        _dati = new List<T>();
    }

    public void Add(T t)
    {
        _dati.Add(t);
    }
}
```

Specifico che la classe che all'interno della classe verrà utilizzato un tipo generico chiamato T

Utilizzo il tipo T come se fosse un tipo reale

```
ListaSolaAggiunta<Studente> Registro = new ListaSolaAggiunta<Studente>();
ListaSolaAggiunta<Libro> Libreria = new ListaSolaAggiunta<Libro>();
```

Programmazione generica

La specifica del tipo di dato generico può avvenire sulla classe (e quindi essere utilizzabile ovunque all'interno di essa) o sul singolo metodo (e quindi essere utilizzabile solo su di esso).

```
public static class Utils
{
  public static T Reflect<T>(T item)
  {
    return item;
  }
}
```

Programmazione generica

Talvolta è necessario inserire dei vincoli sulla tipologia di dati utilizzabili al posto dei dati generici.

Questo può essere fatto utilizzando la parola chiave `where` seguita dalla classe o dall'interfaccia che il tipo di dato deve implementare.

```
public static class Utils
{
  public static T Max<T>(T item1, T item2) where T : IComparable
  {
     return item1.CompareTo(item2) > 0 ? item1 : item2;
  }
}
```

`where T : class`	T deve essere un tipo a riferimento
`where T : struct`	T deve essere un tipo a valore
`where T : Interfaccia`	T deve implementare Interfaccia
`where T : ClasseBase`	T deve ereditare da ClasseBase

Programmazione generica

C# fa un utilizzo molto spinto della programmazione generica, fornendo alcune strutture dati avanzate, tra le quali:

List<T>	Lista di elementi
Stack<T>	Struttura a stack con accesso LIFO. E' possibile solamente aggiungere elementi (Push) o toglierne (Pop)
Queue<T>	Struttura a coda con accesso FIFO. E' solamente possibile aggiungere elementi (Enqueue) o toglierne (Dequeue)
Dictionary<Tkey, Tvalue>	Dizionario che permette di associare valori a chiavi (molto performante per le ricerche)
HashSet<T>	Rappresenta un insieme di elementi univoci
Tuple<T1,…,T8>	Rappresenta una struttura dati formata da, al massimo, 8 tipi di dati.

Delegati, funzioni anonime, lambda expressions

Delegati

In C#, come in tutti i linguaggi di programmazione moderni, anche le funzioni/procedure sono tipi di dato.

E' cioè possibile creare un identificatore che fa riferimento a una porzione di codice da eseguire.

Il codice così referenziato può quindi essere trattato come una variabile, passandolo come parametro ad un metodo, restituendolo come valore di ritorno e, ovviamente, eseguendolo.

Una funzione/procedura utilizzata come tipo di dato viene detta, in C#, delegato ed è definito specificando l'intera firma della funzione, nome compreso, senza però specificare l'implementazione.

```
public delegate void LogMessage(string message);
```

Delegati

Una volta definito, il delegato può essere utilizzato esattamente come un tipo di dato standard:

```
public delegate void LogMessage(string message);

public class Logger
{
    private LogMessage _logger;

    public Logger(LogMessage logger)
    {
        _logger = logger;
    }

    public void Log(string message)
    {
        _logger(message);
    }
}
```

Delegati

Ai parametri di tipo delegato devono essere passate funzioni con firma compatibile, in termini di parametri e valore di ritorno. Il nome non è rilevante.

```
class Program
{
    private static void ConsoleLogger(string messagge)
    {
        Console.WriteLine($"Ho loggato {messagge}");
    }

    static void Main(string[] args)
    {
        Logger AppLogger = new Logger(ConsoleLogger);
        AppLogger.Log("Test");
    }
}
```

Delegati e generics

Essendo ininfluente il nome, i delegati si differenziano solamente per numero e tipo dei parametri e per il tipo dei valori di ritorno. E' pertanto possibile usare i generics per ridurre il numero di delegati

```
public delegate void LogMessage<T>(T message);

public class Logger
{
    private LogMessage<string> _logger;

    public Logger(LogMessage<string> logger)
    {
        _logger = logger;
    }

    public void Log(string message)
    {
        _logger(message);
    }
}
```

```
public delegate void LogMessage<T>(T message);

public class Logger<T>
{
    private LogMessage<T> _logger;

    public Logger(LogMessage<T> logger)
    {
        _logger = logger;
    }

    public void Log(T message)
    {
        _logger(message);
    }
}
```

Delegati e generics

.NET mette a disposizione un numero molto alto di delegati generici pronti all'uso:

- Action<T1…T8> delegati che rappresentano procedure, ovvero senza valori di ritorno
- Func<T1,…Tresult> delegati che rappresentano funzioni, ovvero con valore di ritorno
- Predicate<T> delegato che accetta un parametro e restituisce bool (eq a Func<T,bool>)

```
//public delegate void LogMessage<T>(T message);

public class Logger
{
    private Action<string> _logger;

    public Logger(Action<string> logger)
    {
        _logger = logger;
    }

    public void Log(string message)
    {
        _logger(message);
    }
}
```

```
//public delegate void LogMessage<T>(T message);

public class Logger<T>
{
    private Action<T> _logger;

    public Logger(Action<T> logger)
    {
        _logger = logger;
    }

    public void Log(T message)
    {
        _logger(message);
    }
}
```

Funzioni anonime

Non è obbligatorio creare esplicitamente un metodo da passare ad un delegato.

E' possibile definire il codice "al volo" creando funzioni anonime che contengono solamente parametri e codice da eseguire. Il valore di ritorno non è necessario in quanto determinato in automatico da .NET valutando l'utilizzo di return.

Le funzioni anonime si specificano pertanto:

1. Utilizzando la parola chiave delegate al posto del nome
2. Specificando normalmente parametri e codice
3. Senza specificare il valore di ritorno

La funzione anonima può essere associata ad una variabile compatibile o essere passata come parametro.

Funzioni anonime

```
class Program
{

    static void Main(string[] args)
    {
        Action<string> ConsoleLogger = delegate(string message) {

            Console.WriteLine($"Ho loggato {message}");
        };

        Logger AppLogger = new Logger(ConsoleLogger);
        AppLogger.Log("Test");
    }
}
```

Funzioni anonime

```
class Program
{

    static void Main(string[] args)
    {
        Logger AppLogger = new Logger(delegate (string message)
        {
            Console.WriteLine($"Ho loggato {message}");
        });
        AppLogger.Log("Test");
    }
}
```

Lambda expressions

La scrittura di funzioni anonime può essere notevolmente semplificata utilizzando la notazione della lambda expressions. Una funzione anonima può essere convertita in lambda expression utilizzando le seguenti regole:

- La parola chiave delegate può essere omessa
- Nell'elenco dei parametri i tipi di dato possono essere omessi (verranno dedotti in automatico)
- L'elenco dei parametri e il codice deve essere separato dall'operatore arrow =>

```
Action<string> ConsoleLogger = delegate(string message) {
    Console.WriteLine($"Ho loggato {message}");
};
//diventa
Action<string> ConsoleLogger = (message) => { Console.WriteLine($"Ho loggato {message}"); };
```

Lambda expressions

Le lambda expression possono essere ulteriormente semplificate in alcuni specifici casi:

- Se c'è solo un parametro le parentesi tonde possono essere omesse
- Se il corpo della funzione è composto da una sola istruzione le parentesi graffe possono essere omesse, così come l'eventuale parola chiave return

```
Action<string> ConsoleLogger = delegate(string message) {
    Console.WriteLine($"Ho loggato {message}");
};
//diventa
Action<string> ConsoleLogger = (message) => { Console.WriteLine($"Ho loggato {message}"); };

//semplifico
Action<string> ConsoleLogger = message => Console.WriteLine($"Ho loggato {message}");
```

Delegati, funzioni anonime e lambda expressions - vantaggi

L'utilizzo combinato di delegati, funzioni anonime e lambda expression permette di creare codice più generico e più riutilizzabile, mantenendo contemporaneamente una alta leggibilità.

Esempio: Supponiamo di avere una lista di oggetti (per esempio di classe Libro) e di voler effettuare vari tipi di ricerca: Ricerca per Cognome, per Nome, per numero di recensioni…

Possiamo procedere in due modi:

- Creazione di un metodo di ricerca per ogni tipologia
- Creazione di un unico metodo di ricerca che accetta un delegato tramite il quale specificare l'algoritmo

Delegati, funzioni anonime e lambda expressions - vantaggi

L'approccio della slide precedente è ampiamente utilizzato da .NET, in combinazione con i tipi generici. Per esempio:

```
List<T>.Find(Predicate<T> match);
List<T>.FindAll(Predicate<T> match);
List<T>.FindLast(Predicate<T> match);
List<T>.RemoveAll(Predicate<T> match);
```

Un utilizzo massivo delle lambda expression avviene nella libreria LINQ