Esempi di polimorfismo

Scenari d'implementazione dei tipi astratti predefiniti

Compatibilità: dotNET 2.0+

Anno 2017/2018

Indice generale

1 Introduzione

In questo tutorial propongo alcuni esempi di programmazione che prevedono l'uso di tipi già definiti in .NET: interfacce, classi astratte, classi concrete con metodi virtuali. L'obiettivo è quello di affrontare scenari che mostrino l'importanza dei tipi astratti in generale e come questi siano integrati, insieme ai tipi concreti, all'interno del .NET Framework.

Ogni esempio proposto avrà come obiettivo la realizzazione di una classe che implementa (o deriva) uno o più tipi astratti allo scopo di poter essere utilizzabile in determinati scenari. Prenderò in considerazione i seguenti tipi:

- Classe (concreta) `Object`.
- Interfaccia `IFormattable`.
- Interfaccia `IEnumerable`.
- Classe (astratta) `Stream`.
- Classe (concreta) `Collection`.

2 Formattazione in stringa: object.ToString()

Considera il seguente codice:

```
double n = 12.5;
DateTime adesso = DateTime.Now;
string s = string.Format("[{0}] [{1}]", n, adesso);
Console.WriteLine(s);
```

che produce il seguente output:

```
[12,5] [05/03/2018 21:44:41]
```

Appare evidente che il metodo `Format()` è in grado di ottenere una stringa da ognuno dei parametri. La questione è in che modo ci riesca. Non si tratta di una questione banale; infatti, nel seguente caso l'output non è quello che ci aspetteremmo:

```
class Persona
{
    public string Nome;
    public string Cognome;
}
...
var p = new Persona { Nome = "Pippo", Cognome = "Spada"};
DateTime adesso = DateTime.Now;
var s = string.Format("[{0}] [{1}]", p, adesso);
Console.WriteLine(s);
```

```
[Polimorfismo.ConsoleUI.Persona] [05/03/2018 21:46:32]
```

La questione si complica se si considera la *signature* del metodo[1]:

```
public static string Format(string format, object arg0, object arg1);
```

Ricapitolando:

1. I parametri sono di tipo `object`, eppure `Format()` è in grado di elaborare qualsiasi tipo di dato.
2. Per ogni parametro, `Format()` ottiene una stringa, ma soltanto per i tipi predefiniti è una stringa che rappresenta il valore del parametro.

Entrambe le questioni richiedono una spiegazione, la cui radice risiede nella definizione e nel ruolo del tipo `Object`.

2.1 Tipo Object

`Object` (alias `object`) è il tipo fondamentale dell'intero *type system* di .NET; infatti, qualsiasi tipo di dato deriva da `Object` e dunque ne eredita i metodi. Segue lo scheletro della classe (non riporto i metodi statici):

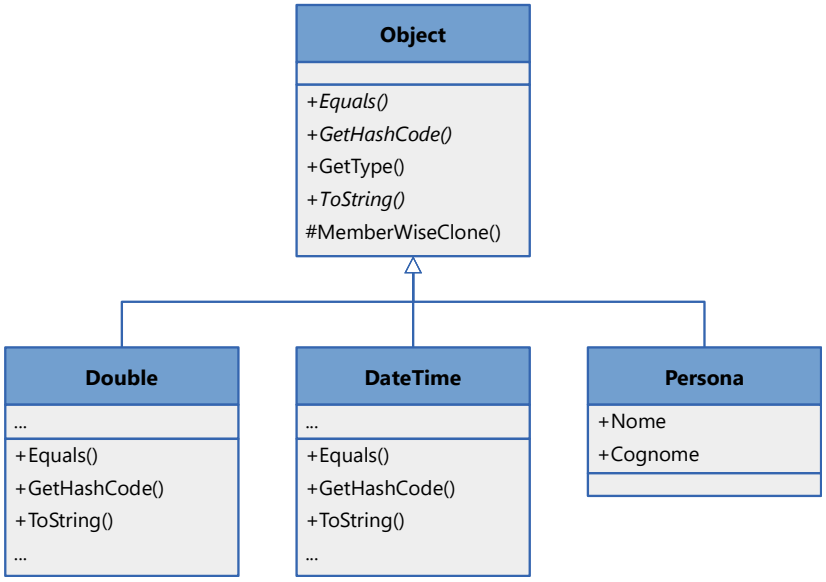
1 In realtà esistono 8 versioni del metodo.

```
public class Object
{
    public Object() { }
    public virtual bool Equals(Object obj) { }
    public virtual int GetHashCode() { }
    public Type GetType() { }

    public virtual string ToString()
    {
       return GetType().ToString();
    }

    protected Object MemberwiseClone() { }
}
```

Nota bene: ho evidenziato i metodi virtuali e riportato l'implementazione del metodo `ToString()`. Questa visualizza il nome, completo di *namespace*, del tipo effettivo dell'oggetto, come mostra il seguente frammento di codice che usa `GetType()`:

```
var persona = new Persona { Nominativo = "Pippo, Spada" };
DateTime adesso = DateTime.Now;
string nomeTipoAdesso = adesso.GetType().ToString();  //-> "System.DateTime"
string nomeTipoPersona = persona.GetType().ToString();//-> "Polimorfismo.ConcoleUI.Persona"
```

Lo schema UML mostra la relazione tra `double`, `DateTime`, `Persona` e `Object`:



Nota bene: `double` e `DateTime` ridefiniscono i metodi virtuali di `Object`, mentre `Persona` si limita a ereditarli.

2.2 Funzionamento del metodo string.Format()

Segue uno pseudo codice super semplificato del metodo `Format()`[2].

```
static string Format(string format, object arg0, object arg1)
{
    // per ogni carattere di "format"
        // se trova un segnaposto
            // formatta argomento corrispondente: (arg0 o arg1).ToString()
            // e aggiungi risultato alla stringa finale
        // ...
}
```

La parte evidenziata spiega il funzionamento del metodo: viene eseguito `ToString()` sul parametro corrispondente al segnaposto.

In conclusione, `Format()` *ignora completamente il tipo effettivo degli argomenti*; si limita a invocare `ToString()` su di essi. La versione di `ToString()` effettivamente eseguita dipende dal fatto che il tipo dell'argomento fornisca una propria versione oppure erediti quella definita in `Object`.

2.3 Rendere un tipo "Formattabile"

Per farlo è sufficiente ridefinire il metodo `ToString()`:

```
class Persona
{
    public string Nome;
    public string Cognome;

    public override string ToString()
    {
        return $"{Cognome}, {Nome}";
    }
}
```

Dopo questa modifica, l'output rispetta le attese, poiché il metodo eseguito sugli oggetti di tipo `Persona` non è più quello ereditato da `Object`, ma quello definito nella classe:

```
[Spada, Pippo] [05/03/2018 21:46:32]
```

2 Il codice reale è piuttosto lungo e complicato, e coinvolge diversi metodi.

3 Formattazione personalizzata: IFormattable

Considera il codice proposto all'inizio del precedente capitolo, con qualche piccola modifica:

```
double n = 12.5;
DateTime adesso = DateTime.Now;
string s = string.Format("[{0:F2}] [{1:d}] [{1:D}]", n, adesso);
Console.WriteLine(s);
```

Ecco l'output, che è chiaramente diverso da quello prodotto in precedenza:

```
[12,50] [06/03/2018] [martedì 6 marzo 2018]
```

Questo è regolato dalle *stringhe di formato* (`F2`, `d`, `D`), che specificano la formattazione da applicare ai parametri. Ebbene, la spiegazione fornita nel precedente capitolo non può giustificare questo risultato, poiché `ToString()` non accetta parametri e dunque non può produrre un risultato diverso in base a un fattore esterno.

Infatti, `Format()` funziona secondo un meccanismo leggermente diverso da quello presentato.

3.1 Caratteristica della "formattabilità"

Considera il seguente problema: vogliamo dare la possibilità di creare dei tipi di dati capaci di produrre una stringa in base a un parametro fornito dall'esterno (la *stringa di formato*). Il metodo virtuale `ToString()` definito in `Object` non va bene, poiché non accetta parametri. Ma è possibile aggirare il problema:

- Modifichiamo `ToString()`, aggiungendo un parametro. Si tratta di una soluzione problematica, poiché in molti scenari non esiste la necessità di specificare una stringa di formato, ma saremmo costretti a farlo. Ad esempio, saremmo costretti a scrivere:

  ```
  lblNumeroOrdini.Text = numOrdini.ToString(null);
  ```

- Definire due metodi `ToString()`, di cui uno con parametro. A questo punto, però, tutti i tipi erediterebbero due metodi `ToString()`, di cui uno probabilmente inutile.[3]

La questione centrale è: soltanto per alcuni tipi è significativo implementare una formattazione personalizzabile. In sostanza è utile introdurre il concetto astratto di *formattabilità*, cioè un *tratto distintivo che caratterizza i tipi in grado di produrre una conversione in stringa personalizzabile*.

3 Per inciso è ciò che effettivamente accade per tutti i metodi virtuali di **Object**, e alcuni ingegneri Microsoft sostengono che non sia stata una buona scelta definire in **Object** tali metodi, poiché la maggiore parte dei tipi non ne hanno bisogno.

3.2 Interfaccia IFormattable

L'interfaccia `IFormattable` definisce il concetto astratto di *formattabilità*. Un tipo che implementa questa interfaccia è in grado di fornire una rappresentazione stringa personalizzabile.

```
public interface IFormattable
{
    string ToString(string format, IFormatProvider formatProvider);
}
```

Il metodo `ToString()` accetta come primo parametro una *stringa di formato* (possiamo ignorare il secondo parametro). È compito dei tipi che implementano l'interfaccia utilizzare il parametro per stabilire il tipo di formattazione da applicare.

3.2.1 Tipo TimeRange: applicazione di IFormattable

Il tipo `TimeRange` implementa un intervallo di tempo (vedi tutorial **TimeRange**) ed è un buon candidato per mostrare in azione il concetto di *formattabilità*. L'attuale versione ridefinisce già il metodo `ToString()` ereditato da `Object`:

```
public struct TimeRange
{
    ...
    public override string ToString()
    {
        string segno = (Ore < 0 || Minuti < 0 || Secondi < 0) ? "-" : "";
        return string.Format("{0}{1}:{2}:{3}", segno, Math.Abs(Ore), Math.Abs(Minuti),
                                               Math.Abs(Secondi));
    }
}
```

Ciò consente di scrivere codice come il seguente:

```
var tr = new TimeRange(1, 2, 3); // (1 ora, 2 minuti e 3 secondi)
Console.WriteLine(tr);
```

e ottenere l'output:

```
1:2:3
```

Desidero aggiungere a `TimeRange` la possibilità di utilizzare un diverso formato, che, nell'esempio, produrrebbe il seguente output[4]: **1 ore 2 minuti 3 secondi**.

Il formato è specificato dalla lettera `T`; pertanto mi aspetto che il codice:

```
var tr = new TimeRange(1, 2, 3); //->1:2:3
var s = string.Format("[{0}] [{0:T}]", tr);
Console.WriteLine(s);
```

produca l'output:

4 Formato senz'altro migliorabile, ma rappresenta soltanto un esempio per introdurre un concetto.

```
[1:2:3] [1 ore 2 minuti 3 secondi]
```

Ecco la nuova versione di `TimeRange`:

```
public struct TimeRange: IFormattable
{
    ...
    public override string ToString() {...}

    public string ToString(string format, IFormatProvider formatProvider)
    {
        if (format == "T")
        {
            string segno = (Ore < 0 || Minuti < 0 || Secondi < 0) ? "-" : "";
            return string.Format("{0}{1} ore {2} minuti {3} secondi", segno, Math.Abs(Ore),
                                                Math.Abs(Minuti), Math.Abs(Secondi));
        }
        return ToString();  //qualunque altro formato produce l'output standard!
    }
}
```

3.3 Uso dell'interfaccia nel metodo Format()

Il tratto della *formattabilità* ha una caratteristica importante: è opzionale. Mentre `Format()` può invocare `ToString()` per ogni argomento, poiché qualsiasi tipo definisce questo metodo (lo eredita da `Object`), non può fare altrettanto con `ToString(string format...)`, poiché soltanto i tipi che implementano `IFormattable` lo definiscono.

Occorre dunque sapere se un oggetto appartiene a un tipo che implementa `IFormattable`. Ecco una nuova versione di `Format()` che gestisce questo problema:

```
static string Format(string format, object arg0, object arg1)
{
    // per ogni carattere di "format"
        // se trova un segnaposto
            string formatString = <estrae stringa di formato da segnaposto>;
            object arg = <seleziona argo0 o arg1>;
            string ris = FormatArg(formatString, arg)
            // aggiungi risultato alla stringa finale
        // ...
}
```

Dopo aver estratto l'eventuale stringa di formato e aver selezionato il parametro corrispondente al segnaposto, il lavoro di formattazione viene eseguito da `FormatArg()`:

```
static string FormarArg(string formatString, object arg)
{
    IFormattable fmt = arg as IFormattable;
    if (fmt != null)   // il tipo run-time di arg implementa IFormattable?
        return fmt.ToString(formatString, null);
```

```
    return arg.ToString();
}
```

Le prime due righe sono fondamentali: l'operatore di cast `as` converte `arg` nel tipo `IFormattable`. Se l'operazione fallisce (il tipo effettivo di `arg` non implementa l'interfaccia), la variabile `fmt` viene inizializzata a `null`. Se l'operazione ha successo, viene invocato il metodo `ToString(string format...)`, altrimenti il metodo `ToString()` ereditato (ed eventualmente ridefinito) da `Object`.

Nota bene, in tutto questo il valore della *stringa di formato* non è rilevante: se `arg` implementa `IFormattable` sarà comunque invocato il metodo dell'interfaccia.

3.4 Conclusioni

La definizione in `Object` del metodo virtuale `ToString()` e, soprattutto, l'esistenza dell'interfaccia `IFormattable` mostrano in azione il fondamentale risultato prodotto dell'uso di astrazioni: *poter scrivere un procedimento che non dipende dal tipo effettivo degli oggetti che elabora*.

Il metodo `Format()` produce una stringa sulla base del valore di alcuni argomenti; *ma non conosce il tipo degli argomenti*, né implementa un procedimento per convertire un argomento in stringa. D'altra parte, `Format()` "sa":

- Che ogni argomento definisce il metodo `ToString()`, poiché qualunque tipo di dato deriva da `Object`.
- Che un argomento può implementare l'interfaccia `IFormattable`, e dunque definire il metodo `ToString(string format…)`.

Sulla base di questi due punti è stato possibile scrivere del codice in grado di formattare in stringa qualsiasi tipo di oggetto. Ciò rende possibile implementare tipi convertibili in stringa e che abbiano la caratteristica della *formattabilità*.

Naturalmente, l'intera questione non riguarda soltanto `Format()`, ma l'intero "ecosistema" che si basa sulla conversione in stringa degli oggetti.

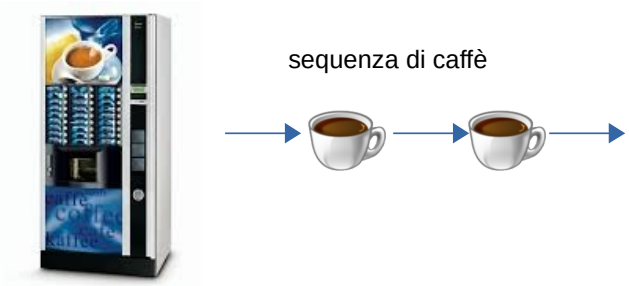
```
var tr = new TimeRange(1, 2, 3);            // (1 ora, 2 minuti, 3 secondi)
Console.WriteLine(tr);                      //->1:2:3
Console.WriteLine("{0}", tr);               //->1:2:3
Console.WriteLine("{0:T}", tr);             //->1 ore  2 minuti  3 secondi
var s = $"[{tr}] [{tr:T}]";                 // (1 ora, 2 minuti, 3 secondi)
Console.WriteLine(s);                       //->[1:2:3]  [1 ore  2 minuti  3 secondi]
```

```
1:2:3
1:2:3
1 ore 2 minuti 3 secondi
[1:2:3] [1 ore 2 minuti 3 secondi]
```

4 Processare sequenze: IEnumerable

In molti scenari il codice deve rispondere alla necessità di processare una sequenza di oggetti. Per comprendere l'importanza di questi scenari occorre innanzitutto comprendere la differenza tra i concetti di *sequenza* e *collezione*, che spesso vengono usati in modo intercambiabile. Per farlo ricorrerò a una metafora.

Considera il distributore di caffè:



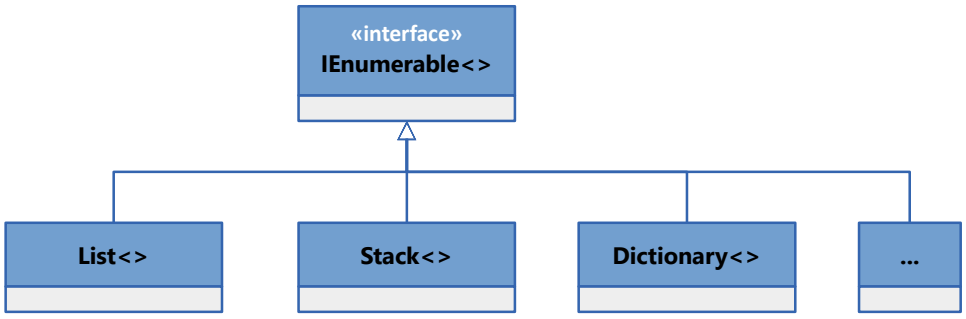
È in grado di produrre una *sequenza di caffè*, ma tutti noi sappiamo che dentro non nasconde una collezione di caffè già pronti: questi vengono preparati, e consumati, man mano che sono richiesti. In sostanza: *la sequenza di caffè non è una cosa che esiste da qualche parte, né prima (dentro il distributore), né dopo (i caffè sono consumati, man mano che sono erogati)*.

D'altra parte, possiamo "collezionare" i caffè in un vassoio per portarli agli amici (e ottenere così una *collezione di caffè*). Sappiamo, inoltre, che il distributore delle bibite, pur comportandosi come quello dei caffè (produce una *sequenza di bibite*), contiene effettivamente una *collezione di bibite*.

In conclusione, *sequenza* è un concetto più astratto di *collezione*. Il secondo suggerisce l'idea di una struttura che contiene degli oggetti; il primo, invece, fa semplicemente riferimento alla capacità di produrre un oggetto per volta, man mano che viene richiesto.

.NET implementa il concetto di sequenza attraverso l'interfaccia generica `IEnumerable`. Il termine può essere tradotto in "iterabile" e cioè: un oggetto è *iterabile* se è in grado di produrre una sequenza di oggetti, uno per volta.

Naturalmente, tutti i tipi di collezione sono *iterabili*, e cioè in grado di produrre in sequenza gli elementi che contengono.



Per accedere agli elementi di una sequenza (dunque, di un oggetto iterabile), occorre usare il costrutto `foreach`. Questo "scorre" gli elementi uno per volta, indipendentemente dal tipo effettivo dell'oggetto iterabile che produce la sequenza: vettore, lista, pila, dizionario, etc.

4.1 Elaborare sequenze di oggetti: usare IEnumerable

Il concetto di sequenza è indipendente dal modo con il quale i dati sono gestiti, è dunque vantaggioso scrivere codice che elabori sequenze senza dipendere dalla loro origine. Lo si fa utilizzando esplicitamente il tipo `IEnumerable`.

Il seguente metodo visualizza una sequenza di nomi:

```
static void Visualizza(IEnumerable<string> listaNomi)
{
    foreach (var nome in listaNomi)
    {
        Console.WriteLine(nome);
    }
}
```

Il metodo non dipende dal tipo effettivo di `listaNomi`, poiché questa variabile referenzia un oggetto iterabile, che è possibile "scorrere" mediante un `foreach`.

`Visualizza()` è utilizzabile con qualsiasi oggetto iterabile, come mostra il seguente codice:

```
string[] nomi = { "filippo", "andrea", "sara" };

List<string> listaNomi = new List<string>(nomi);
Stack<string> pilaNomi = new Stack<string>(nomi);
Queue<string> codaNomi = new Queue<string>(nomi);

Visualizza(nomi);       // -> filippo, andrea, sara
Visualizza(listaNomi);  // -> filippo, andrea, sara
Visualizza(pilaNomi);   // -> sara, andrea, filippo
Visualizza(codaNomi);   // -> filippo, andrea, sara
```

Nota bene: gli elementi della sequenza non vengono prodotti sempre nello stesso ordine; dipende dal tipo effettivo.

4.2 Implementare l'interfaccia IEnumerable

Il meccanismo sottostante all'*iterabilità* è piuttosto complicato da comprendere appieno; qui mi limito a introdurre quanto basta per poterla implementare in classi che gestiscono i dati in una collezione.

Segue un frammento della classe `AlbumFoto`, che gestisce un elenco di fotografie:

```
public class AlbumFoto
{
    List<Foto> fotoList = new List<Foto>();
    ...
    public AlbumFoto(string folder)
    {
        ...
    }
```

```
    public int Count
    {
        get { return fotoList.Count; }
    }

    public Foto this[int index]
    {
        get
        {
            return fotoList[index];
        }
    }
}
```

La classe incapsula una lista e fornisce l'accesso alle foto mediante un indicizzatore. Questa implementazione ha un limite: un oggetto `AlbumFoto` non può essere processato come una sequenza di foto, nonostante di fatto lo sia.

Perché ciò sia possibile deve implementare `IEnumerable`:

```
public class AlbumFoto : IEnumerable<Foto>
{
    List<Foto> fotoList = new List<Foto>();
    ...
    public IEnumerator<Foto> GetEnumerator()
    {
        return fotoList.GetEnumerator();
    }

    IEnumerator Ienumerable.GetEnumerator()  // obsoleto
    {
        throw new NotImplementedException();
    }
}
```

L'interfaccia definisce il solo metodo `GetEnumerator()`[5], il quale restituisce un `IEnumerator`. Quest'ultimo è un *iteratore*, e cioè l'oggetto che ha la funzione di produrre la sequenza.

Comprendere il funzionamento degli *iteratori* non è semplice, e in questo caso inutile; infatti, `AlbumFoto` si limita a restituire l'*iteratore* utilizzato da `fotoList`, il quale produrrà una sequenza di foto nell'ordine con il quale sono memorizzate nella lista.

Segue un frammento di codice che sfrutta l'iterabilità di `AlbumFoto`:

```
AlbumFoto album = new AlbumFoto("Immagini");
...
void LoadGallery()
{
    pnlGallery.Controls.Clear();
```

5 In realtà definisce due metodi **GetEnumerator()**, ma il secondo è obsoleto e può essere ignorato (ma non cancellato).

```
    foreach (Foto foto in album)
    {
        var fotoView = CreateFotoView(foto);
        pnlGallery.Controls.Add(fotoView);
    }
}
```

4.3 Conclusioni

Ci sarebbe molto da aggiungere sull'implementazione degli *iteratori*, ma qui è importante comprendere il fondamentale concetto dell'*iterabilità*, e cioè la capacità di produrre una sequenza di oggetti; fondamentale perché, in moltissimi scenari, tutto ciò che serve è processare i dati in sequenza.

`IEnumerable` rappresenta questo concetto. La sua applicazione implica comprendere due punti:

- Il codice che elabora i dati in sequenza (mediante `foreach`) dovrebbe usare il tipo `IEnumerable`; in questo modo non dipenderà da un particolare tipo concreto.
- Qualunque tipo che gestisce, o è in grado di produrre, una collezione di oggetti, dovrebbe implementare l'interfaccia `IEnumerable`.

La sua implementazione in classi che gestiscono una collezione è estremamente semplice, poiché è sufficiente, nel metodo `GetEnumerator()`, chiamare l'omologo metodo della collezione.

5 Testare in isolamento: classe astratta stream

Di seguito disegno uno scenario tipico, nel quale la possibilità di usare astrazioni è molto importante. Si vuole realizzare un'applicazione client-server che usa il protocollo di trasporto TCP. Il processo di comunicazione viene realizzato mediante un oggetto, `Channel`, che definisce i metodi di scrittura/lettura sullo *stream* di rete. Inoltre, le richieste del client e le risposte del server sono incapsulate in due record, `Request` e `Response`, i quali definiscono i campi contenuti nei messaggi.

Di seguito mostro il codice che incapsula le funzioni di scrittura/lettura delle richieste:

```
public class Request
{
    public string Type;
    public string Body;
}

...

public class Channel
{
    NetwokStream stream;
    public Channel(NetwokStream stream)
    {
        this.stream = stream;
    }

    public Request ReadRequest()
    {
        string[] items = ReadLine().Split(' ');
        return new Request { Type = items[0], Body = items[1] };
    }

    public void WriteRequest(Request req)
    {
        WriteLine($"{req.Type} {req.Body}");
    }

    private string ReadLine() {...}  // legge una riga da stream
    private void WriteLine( string text) {...}  // scrive il testo su stream
}
```

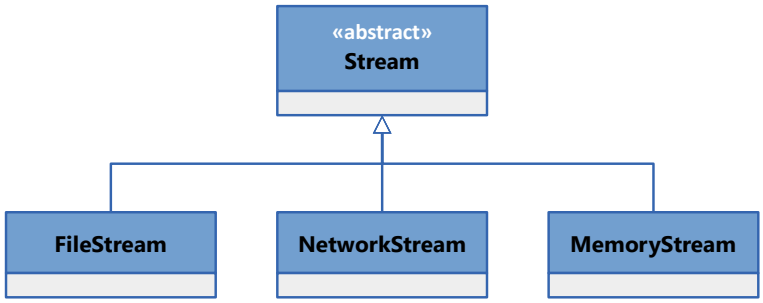
`Channel` viene usata nel client per inviare le richieste, nel server per leggerle; il problema è: come testare il suo funzionamento. È un problema tipico, quando un modulo è inserito in un contesto più ampio e ha dipendenze architetturali (database, *filesystem*, rete, etc).

Nell'esempio, occorre eseguire i due processi client e server, inviare una richiesta dal client e verificare il risultato sul server; ma così facendo non si testa il singolo componente, ma l'intera sequenza di operazioni che hanno luogo durante la comunicazione via TCP tra i due processi. L'obiettivo, al contrario, è riuscire a verificare il corretto funzionamento di `Channel` isolandolo dal contesto in cui viene utilizzato. Ciò semplifica il test e garantisce che il resto del sistema possa fare affidamento su un componente funzionante.

5.1 Rendere Channel indipendente dallo stream di rete

L'attuale versione di `Channel` dipende da un `NetworkStream`, il quale viene ottenuto da un oggetto `TcpClient`; ma si tratta di una dipendenza non necessaria, perché tutto ciò che fa `Channel` è leggere / scrivere su uno *stream*, e non necessariamente uno *stream* di rete.

Stream è un concetto astratto: rappresenta un flusso di byte che può essere scritto e letto. In .NET è implementato attraverso la classe astratta `Stream`, dalla quale derivano vari tipi concreti, che di differenziano per la loro origine / destinazione: file, rete, memoria.



Il primo passo, dunque, è sostituire il tipo concreto `NetworkStream` con la classe astratta `Stream`:

```
public class Channel
{
    Stream stream;
    public Channel(Stream stream)
    {
        this.stream = stream;
    }
    ...
}
```

5.2 Testare Channel in "isolamento"

Poiché `Channel` non dipende più da una connessione di rete, ma da un tipo astratto, è possibile utilizzarlo con qualsiasi tipo di *stream*; in questo caso un `MemoryStream` si presta perfettamente, poiché gestisce il flusso di byte in memoria. Di seguito mostro un metodo che:

- Crea un `Channel` che agisce su un `MemoryStream`.
- Scrive una richiesta sul `Channel`.
- Resetta la posizione dello *stream* (necessario per rileggere i dati appena scritti).
- Legge la richiesta dal `Channel`.
- Visualizza la richiesta scritta e quella letta, verificando che siano uguali.

(L'approccio corretto sarebbe quello di creare uno *unit test*.)

```
static void TestChannel()
{
    var ms = new MemoryStream();
    var ch = new Channel(ms);
    var write = new Request { Type = "LOGIN", Body = "sara" };

    ch.WriteRequest(write);
    ms.Position = 0;   // fa sì che la lettura cominci dal primo byte
    var read = ch.ReadRequest();

    Console.WriteLine($"{write.Type} {write.Body}"); //-> "LOGIN sara"
    Console.WriteLine($"{read.Type} {read.Body}");   //-> "LOGIN sara"
}
```

5.3 Conclusioni

Questo semplice esempio mostra le potenzialità nell'uso di tipi astratti. Ci sono anni luce di distanza tra l'implementazione delle classi `NetworkStream` e `MemoryStream`. La prima deve gestire dei *socket* per la comunicazione bidirezionale di rete, la seconda rappresenta un semplice wrapper su un vettore di byte. Ma, dal punto di vista di `Channel`, questa differenza è irrilevante, poiché sono entrambe dei tipi di *stream*. Ciò consente di applicare una strategia tipica utilizzata nello sviluppo del software: testare il funzionamento di un componente in un contesto diverso da quello reale, semplificando il test e garantendone correttezza e completezza.

6 Riutilizzare l'implementazione: Collection

L'ereditarietà è uno dei due meccanismi usati in .NET per l'implementazione di astrazioni (l'altro è l'uso di interfacce). Alla base di questo meccanismo c'è la possibilità di riutilizzare un'implementazione esistente, estendendola e/o modificandola per adattarla alle proprie esigenze. Questo evita di "dover reinventare la ruota" ogni volta.

Considera l'ipotesi di una classe che gestisce internamente una collezione di oggetti e che, mediante un evento, notifichi eventuali modifiche. Oppure considera l'ipotesi di una collezione che registra in un *log* i propri cambiamenti. Questa avrebbe una funzione simile a quella ipotizzata in precedenza, ma reagirebbe in modo diverso alle eventuali modifiche.

In questi e altri scenari è utile avere a disposizione un tipo che implementi una funzione generale, ma che possa essere adattato a esigenze specifiche, evitando di dover riscrivere un nuovo componente da zero. Un esempio è dato dalla classe generica `Collection`.

`Collection` definisce le tipiche funzioni di una collezione, con una particolarità: i metodi che eseguono delle modifiche sono virtuali, in modo che i tipi derivati possano "decorarli" con ulteriori operazioni, o, al contrario, modificarne il comportamento.

6.1 Realizzare una collezione "osservabile"

Si dice "osservabile" una collezione che notifica i propri cambiamenti, in genere attraverso un evento. Per realizzarla, il modo più semplice è derivarla da `Collection`, poiché basta ridefinire i quattro metodi che modificano gli elementi.

Innanzitutto implemento i tipi necessari per definire l'evento, il quale notifica quattro tipi di cambiamento:

```
public enum ChangeAction
{
    Reset,   // -> la collezione è stata svuotata
    Add,     // -> è stato inserito un elemento
    Remove,  // -> è stato rimosso un elemento
    Replace  // -> è stato modificato un elemento
}

public class ListChangedEventArgs
{
    public ChangeAction Action { get; set; }
    public int Index { get; set; }  // -> indice elemento coinvolto (-1 se Reset)
}
```

Dopodiché implemento una classe generica che deriva da `Collection` e che ridefinisce i metodi virtuali `ClearItems()`, `InsertItem()`, `RemoveItem()` e `SetItem()`:

```
class ObservableList<T> : Collection<T>
{
    public event EventHandler<ListChangedEventArgs> ListChanged;

    protected override void ClearItems()
    {
        base.ClearItems();
```

```
        RaiseEvent(ChangeAction.Reset);
    }

    protected override void InsertItem(int index, T item)
    {
        base.InsertItem(index, item);
        RaiseEvent(ChangeAction.Add, index);

    }
    protected override void RemoveItem(int index)
    {
        base.RemoveItem(index);
        RaiseEvent(ChangeAction.Remove, index);
    }
    protected override void SetItem(int index, T item)
    {
        base.SetItem(index, item);
        RaiseEvent(ChangeAction.Replace, index);
    }

    private void RaiseEvent(ChangeAction action, int index = -1)
    {
        var e = new ListChangedEventArgs { Action = action, Index = index };
        ListChanged?.Invoke(this, e);
    }
}
```

Alcune considerazioni:

- I metodi virtuali sono protetti e dunque accessibili soltanto nelle classi derivate; vengono chiamati dai metodi pubblici (non virtuali) soltanto dopo la validazione degli argomenti. Tutto questo semplifica l'implementazione della classe.
- In ogni metodo invoco innanzitutto il metodo base; il mio scopo non è modificare il funzionamento della classe base, ma aggiungere un nuovo comportamento.

6.1.1 Uso della collezione

Segue un frammento di codice che usa la classe:

```
ObservableList<string> list = new ObservableList<string>();
list.ListChanged += List_ListChanged;
list.Add("sara");      // -> Add 0
list.Add("filippo");   // -> Add 1
list[1] = "andrea";    // -> Replace 1
list.RemoveAt(0);      // -> Remove 0
list.Clear();          // -> Reset -1

private static void List_ListChanged(object sender, ListChangedEventArgs e)
{
    Console.WriteLine($"{e.Action}  {e.Index}");
}
```