=
l
2
.
k

Oggetti

Progettare oggetti

Tutorial referenziati: **Record**, **Moduli**.

Anno 2019/2020

Indice generale

1 Introduzione

Questo tutorial affronta il tema centrale della programmazione *object oriented*: l'implementazione di *oggetti* (classi).

Introdurrò le funzioni svolte dagli *oggetti* e alcuni principi che ispirano la loro progettazione. Contestualmente, approfondirò la sintassi e le funzionalità fornite dal linguaggio C# per la loro implementazione.

Partirò innanzitutto da un breve riepilogo del *modello di programmazione procedurale* e dalla funzione svolta da *record* e *moduli*. (Vedi i tutorial ***Record*** e ***Moduli***.)

1.1 Modello di programmazione procedurale

Da wikipedia:

> *La programmazione procedurale è un paradigma di programmazione che consiste nel creare dei blocchi di codice, identificati da un nome… Questi sono detti sottoprogrammi, procedure o funzioni, a seconda del linguaggio…*

In sostanza, il programma è strutturato in procedure (in C#, *metodi*) ognuna delle quali risolve un sotto problema. Nella forma più semplice, questo modello di programmazione prevede l'uso di tipi primitivi – intero, reale, stringa, *array*, etc – per memorizzare i dati del problema.

1.2 Estendere il modello procedurale: *record* e *moduli*

Qualsiasi linguaggio supera il modello di programmazione di base e aggiunge la possibilità di:

- Definire strutture dati in grado di rappresentare concetti che vadano oltre il singolo valore: i *record*.
- Implementare funzioni che vadano oltre il singolo procedimento, o metodo: i *moduli*.

I due costrutti non si sovrappongono. I *record* rappresentano dei dati, con l'implicita assunzione che, da "qualche parte" nel programma siano definiti i metodi necessari ad elaborarli. I *moduli* implementano delle funzioni mediante uno o più metodi; se necessario possono definire delle variabili, le quali sono però nascoste al codice che utilizza il *modulo*.

1.3 Limite del modello di programmazione procedurale

Tenendo ben separati *dati* e *funzioni*, questo modello di programmazione non consente di trattare facilmente concetti che abbracciano entrambi gli aspetti.

Per comprendere questo limite, supponi che C# sia un linguaggio esclusivamente procedurale e dunque non fornisca i tipi `List`, `Stack`, `Dictionary`, etc. Supponi, inoltre, di dover gestire una lista indicizzabile di nominativi.

Una *lista* implica la necessità di memorizzare gli elementi e di eseguire delle operazioni su di essi – inserimento, accesso indicizzato, rimozione, etc; incapsula, cioè, una funzione, e dunque può essere realizzata mediante un *modulo*.

1.3.1 Implementazione di una lista di stringhe mediante un modulo

Di seguito fornisco una soluzione semplificata:

- Un vettore, `items`, memorizza gli elementi della lista; la variabile intera `count` memorizza il numero di elementi effettivamente memorizzati (inizialmente, zero)
- Dei metodi, `Count()`, `Add()`, `GetItem()`, etc, implementano le tipiche operazioni che si compiono su una lista indicizzabile.

```
static class StringList
{
    static string[] items;  // memorizza gli elementi
    static int count;       // memorizza il numero di elementi (inizialmente zero)

    public static void Create()     // crea la lista (capacità predefinita: 100 elementi)
    {
        items = new string[100];   // è possibile memorizzare al massimo 100 elementi
        count = 0
    }

    public static int Count()     // restituisce il numero di elementi nella lista
    {
        return count;
    }

    public static void Add(string item)   // aggiunge un elemento
    {
        items[count] = item;
        count++;
    }

    public static string GetItem(int index)  // ritorna l'elemento all'indice specificato
    {
        return items[index];
    }
    ...   // altri metodi ( SetItem(), RemoveAt(), etc )
}
```

`StringList` consente, ad esempio, di elaborare un elenco di nominativi:

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

static void InserisciNominativi()
{
    StringList.Create();
    Console.WriteLine("Inserisci nominativi (invio per terminare)");
    string nome = Console.ReadLine();
    while (nome != "")
    {
        StringList.Add(nome);
        nome = Console.ReadLine();
    }
}

static void VisualizzaNominativi()
{
    for (int i = 0; i < StringList.Count(); i++)
    {
        string nome = StringList.GetItem(i);
        Console.WriteLine(nome);
    }
}
```

Questa soluzione presenta però un problema: `StringList` *gestisce* una lista, ma *non è in sé* una lista. Non è un tipo di dato e dunque non consente di definire variabili, cosa ovviamente necessaria, ad esempio, se il programma richiede di elaborare più liste di stringhe.

```
StringList nomi;        // un modulo non è un tipo e non permette di dichiarare variabili!
StringList cognomi;
```

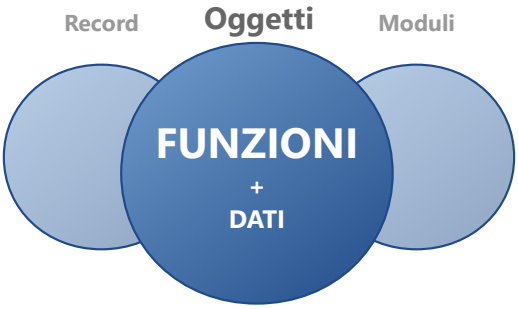
In realtà, il modello di programmazione procedurale consente di superare questo limite (una tecnica è mostrata in **Oggetti "Undercover"**), ma non in modo elegante e facile da utilizzare. Ciò che serve è un nuovo costrutto, che sia un tipo di dato (come il *record*) e che consenta di implementare delle funzioni (come il *modulo*).

2 Introduzione gli *oggetti* (classi)

Dal libro *Object-Oriented Analisys and Design* di G.Boock, una definizione (semplificata):

> *un oggetto definisce le caratteristiche di entità che hanno uno **stato** e un **comportamento**.*

Gli *oggetti* uniscono le caratteristiche di *record* (dati→*stato*) e *moduli*, (funzioni→ *comportamento*).



Nota bene: in figura ho dato maggior risalto al termine "FUNZIONI". Come vedremo, l'aspetto fondamentale di un *oggetto* è il suo "comportamento" e cioè l'insieme di operazioni che si possono eseguire su di esso.

2.1 Terminologia: elementi di base sugli oggetti

Nel tutorial uso il termine *oggetto* per designare ciò che, nella terminologia comune, è una ***classe***. Dunque, ad esempio, `Button` , `List<>`, `TextBox`, sono classi (*oggetti*, nella mia terminologia), e cioè *tipi*. Ho scelto di usare il termine *oggetto*, perché C# usa la parola chiave `class` anche per definire *record* e *moduli*.

L'importante è comprendere i concetti e non farsi fuorviare dalle parole chiave (altri linguaggi hanno parole chiave specifiche per i tre concetti). Nel testo uso il corsivo per dare a *oggetto* il significato di *classe*.

2.1.1 Istanze

Nella terminologia comune, gli oggetti che vengono creati durante l'esecuzione del programma sono chiamati *istanze*. Dunque, ad esempio, nelle istruzioni:

```
Button btn;            // non esiste ancora alcuna istanza
btn = new Button();    // l'istanza è stata creata e assegnata a "btn"
```

`Button` è una classe, `btn` è una variabile che referenzia un'*istanza* della classe `Button`.

2.1.2 Ciclo di vita delle istanze

Il *ciclo di vita* fa riferimento alla durata, o esistenza, di un'istanza, e cioè:

- A partire da quando viene creata mediante l'operatore `new`.
- Fino a quando non è più utilizzabile.

Facendo riferimento al codice precedente,

```
Button btn;            // non esiste ancora alcuna istanza
btn = new Button();    // l'istanza è stata creata e assegnata a "btn"
```

lo schema seguente riassume la situazione prima e dopo l'istruzione `new`:

(Nota bene: la variabile e l'istanza sono due entità distinte.)

Il ciclo di vita dell'istanza termina quando non è più referenziata da nessuna variabile. La memoria viene automaticamente recuperata (*garbage collected*) per essere riutilizzata in seguito.

2.1.3 *Stato di un* oggetto (istanza)

Lo *stato* di un'istanza è rappresentato dall'insieme dei valori memorizzati nei suoi campi.

Fin dalla fase di progettazione della classe è necessario stabilire quali sono gli stati ammessi e quali no. I metodi della classe devono "rifiutarsi" di assegnare ai campi dei valori che producano uno stato non ammesso. (4)

2.2 Definizione di un *oggetto*

La definizione di un *oggetto* è del tutto simile a quella di un *modulo*; scompare la parola `static`:

```
<modificatore> class <nome>
{                                opzionali
    <modificatore> <tipo> <campo>;
    <modificatore> <tipo> <campo>;
    ...
                                                opzionali
    <modificatore> <tipo> <nome metodo>(<argomenti>) {...}
    <modificatore> <tipo> <nome metodo>(<argomenti>) {...}
    ...
}
```

Ricapitolando:

1. Un *oggetto* è definito mediante la parola chiave `class`, che introduce un nuovo tipo, rappresentato dal nome dell'*oggetto*.
2. Definisce zero o più campi e zero o più metodi. Campi e metodi sono chiamati ***membri di istanza*** dell'*oggetto*.

   La parola *istanza* indica che il codice esterno potrà eseguire i metodi dell'*oggetto* soltanto attraverso un'istanza. In contrapposizione ai membri statici di un *modulo*, che vengono invocati attraverso il nome del *modulo*.

3. Ogni definizione può essere preceduta da un modificatore di accesso, (`public` o `private`). (In sua assenza viene considerato `private`.) Soltanto i membri pubblici sono accessibili al codice esterno all'*oggetto*.

Dalle precedenti regole si trae subito un'implicazione: il linguaggio C# non fa distinzione tra *oggetti* e *record;* l'unica distinzione è tra *classi* (*oggetti* e *record*) e *classi statiche* (*moduli*). È compito del programmatore, nel rispetto dei principi di programmazione *object oriented*, stabilire se implementare un *record* o un *oggetto*.

2.2.1 Interfaccia pubblica e implementazione privata

Il termine *interfaccia pubblica* designa l'insieme di membri pubblici di un *oggetto*. Progettare accuratamente l'interfaccia pubblica è l'aspetto centrale nella realizzazione di un *oggetto*, poiché è questa a stabilire la sua funzione e come sarà usato dal codice esterno. (3.2)

L'*implementazione privata* è data dall'insieme dei membri privati dell'oggetto. Questa può essere modificata senza che il codice esterno debba essere adattato a causa di queste modifiche. (3.3)

2.3 Usare gli oggetti

L'uso degli *oggetti* ricalca fedelmente quello dei *record*, poiché rappresentano dei tipi di dati (Ricordo nuovamente che C# non fa distinzione tra i due.) Diversamente dai *moduli*, l'invocazione dei metodi avviene attraverso una variabile che referenzia un'*istanza*. , che ovviamente deve essere prima creata.

3 Implementare oggetti: un esempio

Nel procedere all'implementazione di un *oggetto*, l'approccio da usare è quello del *designer*. Quando si vuole produrre un nuovo modello di auto, non si parte dalla realizzazione dei componenti o dalla progettazione della catena di montaggio, si parte dal disegno del veicolo; da un modello in scala; dalla progettazione degli interni; etc. In sostanza, si progetta il veicolo dal punto di vista di chi lo dovrà utilizzare: il cliente.

Nel progettare un *oggetto* si deve adottare un processo analogo: stabilire innanzitutto il concetto che l'*oggetto* rappresenta, la funzione che svolge, i requisiti, dal punto di vista dell'utilizzatore, che deve soddisfare. Dopodiché si può pensare al modo in cui *l'oggetto* può essere implementato per soddisfare i requisiti suddetti, compresi gli eventuali dati di cui necessita.

In conclusione: come per i *moduli*, è la funzione svolta dall'*oggetto* l'aspetto più importante.

Questo approccio è fondamentale per tenere separato *ciò che* l'*oggetto fa*, dal *come è in grado di farlo*. In sostanza: separa l'*interfaccia pubblica* (ciò che l'utilizzatore vede dell'*oggetto*) dalla sua *implementazione* (come l'oggetto funziona).

È il *principio di incapsulamento*, e non c'è principio più importante.

Di seguito riprendo l'esempio della lista di stringhe per mostrare in azione il processo di implementazione di un *oggetto* e l'applicazione del principio di incapsulamento.

3.1 Implementare una lista: classe StringList

Una *lista* è una collezione di elementi accessibili mediante la loro posizione. È dinamica e dunque consente di aggiungere nuovi elementi e rimuovere quelli esistenti. Perché soddisfi questi requisiti, una classe che implementa la *lista* deve fornire almeno le seguenti operazioni:

- Creazione → la lista è vuota.
- Ottenere il numero di elementi memorizzati.
- Inserimento di un elemento → la lista contiene un elemento in più.
- Rimozione di un elemento, data la sua posizione → la lista contiene un elemento in meno.
- Accesso a un elemento, data la sua posizione → viene restituito l'elemento richiesto.
- Modifica dell'elemento alla posizione specificata → viene memorizzato il nuovo elemento all'indice specificato.

In sostanza, deve rendere possibile scrivere codice come il seguente:

```
StringList nomi;    // dichiarazione variabile, ma non esiste ancora nessuna istanza.

static void Main(string[] args)
{
    InserisciNominativi();
    VisualizzaNominativi();
}

static void InserisciNominativi()
{
```

```
    // viene creata un istanza e assegnata a "nomi". La lista è vuota
    nomi = new StringList();
    Console.WriteLine("Inserisci nominativi (invio per terminare)");
    string nome = Console.ReadLine();
    while (nome != "")
    {
        nomi.Add(nome);   // viene aggiunto un elemento alla lista
        nome = Console.ReadLine();
    }
}

static void VisualizzaNominativi()
{
    for (int i = 0; i < nomi.Count(); i++)  // Count()-> n° elementi della lista
    {
        string nome = nomi.GetItem(i);   // accede all'"iesimo" elemento della lista
        Console.WriteLine(nome);
    }
}
```

3.2 Interfaccia pubblica della classe StringList

L'insieme dei membri pubblici rappresenta l'***interfaccia pubblica*** della classe e ne definisce la funzione. Dunque: *qualunque membro dell'oggetto non necessario a definirne la funzione deve essere privato!*

Segue l'interfaccia pubblica di `StringList`:

```
class StringList
{
    ...
    public StringList() { ... }
    public int Count() { ... }
    public void Add(string item) { ... }
    public void RemoveAt(int index) { ... }
    public string GetItem(int index) { ... }
    public void SetItem(int index, string item) { ... }
    ...
}
```

Per il codice esterno, `StringList` è completamente definita dalla sua interfaccia pubblica; non ha importanza il modo in cui i metodi sono implementati, né dove e come siano memorizzati gli elementi.

3.3 Implementazione (privata) della classe

I membri privati e il codice dei metodi contribuiscono all'***implementazione privata*** della classe. Come accade nei *moduli*, campi e metodi privi del modificatore `public` sono considerati privati.

Segue l'implementazione completa della classe (i membri privati sono evidenziati):

```
class StringList
{
    private string[] items;
    private int count;

    public StringList()
    {
        items = new string[10];
        count = 0;
    }

    public int Count()
    {
        return count;
    }

    public void Add(string item)
    {
        if (count == items.Length)
            EnsureCapacity();
        items[count] = item;
        count++;
    }

    public string GetItem(int index)
    {
        return items[index];
    }

    public void SetItem(int index, string item)
    {
        items[index] = item;
    }

    public void RemoveAt(int index)
    {
        for (int i = index; i < count-1; i++)
            items[i] = items[i + 1];
        count--;
    }
   // raddoppia la capacità della lista, copiando gli elementi in un nuovo vettore
    private void EnsureCapacity()
    {
        var newItems = new string[items.Length * 2];
        for (int i = 0; i < items.Length; i++)
            newItems[i] = items[i];
        items = newItems;
    }
}
```

La separazione tra *implementazione* e *interfaccia pubblica* è molto importante:

- Soltanto i membri che definiscono la funzione della classe sono utilizzabili dal codice esterno; dunque:
- è possibile modificare l'implementazione senza che il codice esterno ne sia influenzato.

3.4 *L'importanza di un'implementazione privata: refactoring*

In `StringList` il metodo `Add()` garantisce che il vettore contenente gli elementi abbia la capacità necessaria a memorizzare il nuovo elemento; se il vettore è pieno, ne viene creato uno nuovo mediante l'invocazione di `EnsureCapacity()`:

```
public void Add(string item)
{
    if (count == items.Length)  // il vettore "items" e' pieno?
        EnsureCapacity();
    items[count] = item;
    count++;
}
```

Il metodo `EnsureCapacity()` crea un vettore più capace, copiandovi gli elementi della lista; ma esiste già un metodo predefinito in grado di eseguire questo compito, `Array.Resize()`:

```
public void Add(string item)
{
    if (count == items.Length) // il vettore "items" e' pieno?
        Array.Resize(ref items, items.Length * 2);
    items[count] = item;
    count++;
}
```

Dopo questa modifica, `EnsureCapacity()` può essere eliminato senza timore di impattare sul codice esterno, *ma soltanto perché è privato!* Se fosse pubblico, eliminarlo significherebbe produrre un ***breaking change***, cioè una modifica in grado di invalidare il codice che usa la classe.

3.5 Costruttori

I costruttori sono metodi speciali che vengono eseguiti nella fase di creazione degli oggetti. Il loro scopo è stabilire lo stato iniziale dell'oggetto.

Nonostante siano assimilabili a dei metodi `void`, i costruttori hanno il nome della classe e non specificano un tipo di ritorno.

3.5.1 Costruttore predefinito

Un costruttore che non accetta parametri è detto ***costruttore predefinito***. Una classe che non definisce alcun costruttore "riceve" automaticamente dal linguaggio un *costruttore predefinito vuoto*, cioè privo di istruzioni.

Ad esempio, è possibile implementare `StringList` senza costruttori, ci penserà il linguaggio a fornirne uno senza parametri:

```
class StringList
{
    // l'inizializzazione dei campi avviene in fase di dichiarazione
    string[] items = new string[10];
    int count = 0;

    public StringList() {}  // viene fornito dal linguaggio
}
```

Ha senso fornire un *costruttore predefinito* (o accettare quello fornito dal linguaggio) soltanto se per l'*oggetto* esiste uno stato predefinito coerente con la funzione che svolge.

Nel caso di `StringList`, l'esistenza del solo costruttore predefinito è accettabile, poiché lo stato iniziale della lista è prodotto dall'inizializzazione dei campi: zero elementi in un vettore di capacità 10.

3.5.2 Costruttore con parametri

Se per un *oggetto* non esiste uno stato predefinito valido, oppure si vuol fornire al codice esterno la possibilità di configurare lo stato iniziale, occorre definire un costruttore che accetti uno o più argomenti.

Ad esempio, si potrebbe dotare `StringList` di un costruttore che stabilisca la capacità iniziale del vettore; ciò è utile per evitare l'allocazione di nuovi vettori quando si conosce in anticipo il numero di elementi da memorizzare nella lista.

```
class StringList
{
    string[] items;
    int count;

    public StringList(int capacity)  // il codice esterno stabilisce la capacità iniziale
    {
        items = new string[capacity];
        count = 0;
    }
}
```

Poiché, adesso, `StringList` ha un costruttore con parametri, il linguaggio non fornisce più quello predefinito; dunque, non è più possibile creare una lista senza specificare la capacità iniziale:

```
StringList elenco1 = new StringList(50) // OK!

StringList elenco2 = new StringList()  // ERRORE: non esiste il costruttore StringList()!
```

3.5.3 Definire più costruttori

Una classe può definire più costruttori, purché si differenzino per il numero e/o tipo dei parametri. Nel caso di `StringList` può essere opportuno fornire sia il costruttore predefinito, sia uno che accetti la capacità iniziale:

```
class StringList
{
    string[] items;
    int count;

    public StringList()
    {
        items = new string[10];
        count = 0;
    }

    public StringList(int capacity)
    {
        items = new string[capacity];
        count = 0;
    }
}
```

3.5.4 Chiamare un costruttore da un altro: parola chiave this.

All'interno della classe, un costruttore non può essere invocato, con un'eccezione: un costruttore può chiamarne un altro mediante la parola chiave `this`. Ciò è utile per evitare duplicazioni di codice, allo stesso modo di quando un metodo né chiama un altro per delegargli una parte del proprio compito.

In `StringList` i due costruttori condividono praticamente lo stesso codice; è possibile evitare questa duplicazione facendo sì che il *costruttore predefinito* chiami l'altro, specificando una costante come capacità iniziale:

```
class StringList
{
    string[] items;
    int count;
    public StringList():this(10) { }  // esegue StringList(int capacity)

    public StringList(int capacity)
    {
        items = new string[capacity];
        count = 0;
    }
}
```

Nota bene: la parola `this` funziona come alias del costruttore invocato e viene specificata prima del corpo del *costruttore predefinito*, che in questo caso è vuoto.

3.5.5 Definire dei parametri con valori predefiniti

Nella maggior parte dei casi non è necessario definire più costruttori, poiché l'uso di *parametri con valore predefinito* consente di ottenere lo stesso risultato:

```
class StringList
{
    string[] items;
    int count;

    public StringList(int capacity = 10)
    {
        items = new string[capacity];
        count = 0;
    }
    ...
}
```

Ora la classe ha un solo costruttore, che però può essere invocato senza fornire alcun argomento. Sarà il linguaggio, nell'istruzione di chiamata senza argomenti, a fornire il valore predefinito.

```
static void Main(string[] args)
{
    StringList lista = new StringList();     //  equivale a: StringList(10)
    StringList lista2 = new StringList(20);  //  equivale a: StringList(20)

    VisualizzaNominativi();
}
```

4 Invariante di classe

Nella definizione di G. Book, un oggetto (un'istanza di classe) ha uno *stato* e un *comportamento*. Lo *stato* è definito dall'insieme dei dati memorizzati nei campi; il comportamento è definito dall'insieme dei metodi.

Da questo punto di vista, un oggetto può essere visto come un *dispositivo*. Considera uno sportello bancomat:

- L'utente inserisce la tessera bancomat e seleziona il servizio richiesto (supponi un prelievo).
- L'utente inserisce il PIN, che viene verificato ed eventualmente rifiutato.
- L'utente imposta l'entità del prelievo.
- Lo sportello restituisce la tessera ed eroga i contanti richiesti.

In questa metafora, ogni operazione eseguita dall'utente equivale all'esecuzione di un metodo, il quale produce un risultato (un output sul display, il contante richiesto, etc) e modifica la memoria interna dello sportello – lo *stato* del dispositivo – che tiene traccia dello stato della transazione.

Il funzionamento dello sportello si basa su un assunto fondamentale: *ogni azione dell'utente viene verificata coerentemente con lo stato della transazione, ed eventualmente rifiutata*. L'utente non può digitare il PIN fintantoché non ha inserito il bancomat e selezionato il tipo di operazione. L'utente non può impostare la quantità da prelevare finché non è stato verificato il PIN. Viene verificata la disponibilità della somma richiesta, etc. In conclusione: la transazione viene annullata se non esistono la condizioni – input dell'utente, disponibilità contante, etc – perché possa essere completata con successo.

È la caratteristica fondamentale di un dispositivo ben progettato: garantisce che il proprio stato interno e l'output emesso siano corretti e coerenti con gli input ricevuti. Per ottenere questo, il dispositivo è vincolato a rifiutare di compiere operazioni che possono produrre un output errato o condurre a uno stato non valido.

Nella OOP, l'insieme di vincoli che garantiscono un corretto funzionamento è definito *invariante di classe*.

4.1 Invariante di classe

L'***invariante di classe*** stabilisce che un oggetto, a partire dalla sua creazione, deve sempre trovarsi in uno stato valido: prima e dopo l'esecuzione di un metodo, i suoi campi devono memorizzare dati coerenti con la funzione svolta. Se ciò non è possibile, il metodo deve interrompere la propria esecuzione, "rifiutandosi" di eseguire l'operazione richiesta.

L'invariante pone dei vincoli da rispettare sul funzionamento dei metodi pubblici, chiamati ***precondizioni*** e ***postcondizioni***. Questi rappresentano un "contratto" fra il codice esterno e la classe: il codice esterno "è tenuto" a chiamare i metodi fornendo dei dati validi, il codice della classe a fornire risultati e comportamenti corretti sulla base di quei dati.

4.2 *Precondizioni*

Le precondizioni sono vincoli che il codice esterno deve soddisfare: ha la responsabilità di fornire degli argomenti validi ai metodi chiamati, compresi i costruttori. Questi ultimi, d'altra parte, "non possono fidarsi" e dunque devono verificare che le *precondizioni* siano soddisfatte; in caso contrario devono sollevare un'eccezione, interrompendo l'esecuzione.

È importante sottolineare questo concetto: accettando di elaborare dati non validi, il metodo rischia di modificare scorrettamente lo stato dell'oggetto, oppure di produrre un risultato incoerente o non deterministico. È una situazione da evitare assolutamente, perché può condurre a bug molto difficili da trovare!

4.2.1 *Precondizioni che riguardano lo stato dell'oggetto*

In alcuni casi, la *precondizione* non riguarda gli argomenti del metodo, ma lo stato dell'oggetto: occorre verificare che sia coerente con il tipo di operazione richiesta; in caso contrario, il metodo deve sollevare un'eccezione, indicando così che, in quel contesto, l'operazione non può essere eseguita.

Un esempio tipico è rappresentato dal metodo `Pop()` della classe `Stack`. Se la pila è vuota, il metodo solleva l'eccezione `InvalidOperationException`: non è possibile estrarre un valore da una pila vuota.

4.3 *Postcondizioni*

Le *postcondizioni* stabiliscono i vincoli sullo *stato* che un metodo pubblico deve garantire *dopo la propria esecuzione*. Se un metodo restituisce un valore, le *postcondizioni* stabiliscono eventuali vincoli anche su di esso.

Ad esempio, il metodo `Add()` deve garantire che:

- Il numero di valori della lista sia aumentato di 1.
- Il nuovo valore sia l'ultimo della lista.

Mentre le *postcondizioni* del metodo `RemoveAt()` sono:

- Il numero di elementi della lista è diminuito di 1.
- La posizione degli elementi che si trovavano dopo l'elemento rimosso è decrementata di uno.

In sostanza, il rispetto delle *postcondizioni* fa da garanzia al corretto funzionamento del metodo.

4.3.1 *Non soddisfare le postcondizioni*

Non soddisfare le *postcondizioni* implica l'esistenza di un bug nel codice, oppure la mancanza della verifica delle *precondizioni*.

Considera la seguente versione di `Add()`, nella quale ho volutamente commentato l'istruzione `count++`:

```
public void Add(string item)
{
    if (count == items.Length)
        Array.Resize(ref items, items.Length*2);
    items[count] = item;
    // count++;  non viene incrementato il numero degli elementi!
}
```

In sé, l'esecuzione del metodo non produce alcun errore, ma una *postcondizione* non è soddisfatta: il numero degli elementi non è aumentato. Il risultato è che, pur senza produrre alcun errore durante l'esecuzione del metodo, la lista memorizzerà sempre un solo elemento, l'ultimo inserito.

4.4 Applicare l'invariante di classe a StringList

Nel progettare `StringList` non ho considerato l'invariante di classe; per farlo, si tratta di verificare le *precondizioni* sui metodi, costruttore compreso.

4.4.1 Creazione della lista: verifica della capacità iniziale

Il costruttore accetta la capacità iniziale della lista; occorre stabilire la *precondizione* da soddisfare sul parametro: `capacity > 0` o `capacity >= 0`. La prima garantisce sicuramente l'invariante di classe; apparentemente anche la seconda, invece provoca l'errato funzionamento del metodo `Add()`.

Considera il seguente codice:

```
StringList elenco = new StringList(0) // capacità iniziale zero
elenco.Add("Stefania");               // -> eccezione (non aumenta capacità del vettore)
```

L'esecuzione di `Add()` aumenta la dimensione del vettore moltiplicando la sua lunghezza per due. Naturalmente, se `Length` è zero, il nuovo vettore sarà anch'esso di lunghezza zero:

```
public void Add(string item)
{
    if (count == items.Length)  // la prima volta, Length è 0
        Array.Resize(ref items, items.Length * 2);    // Length * 2 -> 0!
    items[count] = item;   // -> produce un eccezione!
    count++;
}
```

Il costruttore e il metodo `Add()` sono corretti, ma incoerenti tra loro. La soluzione è quella di applicare una *precondizione* più restrittiva nel costruttore (`capacity > 0`), oppure nel modificare il metodo `Add()`. Scelgo la seconda soluzione:

```
public void Add(string item)
{
    if (count == items.Length)
    {
        int newCapacity = items.Length == 0 ? 10 : items.Length * 2;
        Array.Resize(ref items, newCapacity);
```

```
    }
    items[count] = item;
    count++;
}
```

Verifica della precondizione nel costruttore: capacity >= 0

Nel costruttore, stabilita la precondizione (`capacity >=0`), occorre verificare che sia soddisfatta:

```
public StringList(int capacity = 10)
{
    if (capacity < 0)
        throw new ArgumentOutOfRangeException(nameof(capacity));

    items = new string[capacity];
    count = 0;
}
```

L'istruzione `throw` crea un'eccezione del tipo specificato. L'operatore `nameof` produce una stringa contenente il nome della variabile.

In sostanza: se la *precondizione* non è soddisfatta, il costruttore si "rifiuta" di costruire l'oggetto.

4.4.2 Accesso all'elemento: GetItem() e SetItem()

I metodi `GetItem()` e `SetItem()` non verificano che l'indice specificato sia valido; ciò crea un problema di coerenza nel loro comportamento, come mostra il seguente codice:

```
StringList nomi = new StringList();  //-> capacità predefinita: 10
nomi.Add("Gianni");                  //-> numero elementi: 1
string nome2 = nomi.GetItem(3);      //-> restituisce null
string nome1 = nomi.GetItem(10);     //-> produce IndexOutOfRangeException
```

Nel primo caso viene restituito `null` perché l'indice 3 referenzia un elemento esistente, ma non inizializzato; nel secondo caso viene prodotta un'eccezione, perché l'elemento di indice 10 non esiste affatto nel vettore. In realtà entrambi gli elementi non esistono nella lista e dunque ci si aspetta che entrambe le chiamate a `GetItem()` producano lo stesso comportamento!

Occorre verificare che l'indice rientri nell'intervallo `0 <= indice < count`:

```
public string GetItem(int index)
{
    ValidateIndex(index);
    return items[index];
}

public void SetItem(int index, string item)
{
    ValidateIndex(index);
    items[index] = item;
}
```

```
private void ValidateIndex(int index)
{
    if (index < 0 || index >= count)
        throw new ArgumentOutOfRangeException(nameof(index));
}
```

4.4.3 Eliminazione di un elemento: RemoveAt()

Anche in questo caso occorre validare l'indice, altrimenti si ottiene un comportamento inconsistente:

```
StringList nomi = new StringList();
nomi.Add("Gianni");
nomi.Add("Sara");
nomi.RemoveAt(3);  //->non rimuove elemento, né produce alcun errore
```

Il fatto che `RemoveAt()` non produca errori in caso di indice non valido sembra un fatto positivo, ma non è così. Il codice esterno *si aspetta che un elemento venga eliminato* (*postcondizione*); dunque, se ciò non è possibile, il metodo deve sollevare un'eccezione:

```
public void RemoveAt(int index)
{
    ValidateIndex(index);
    for (int i = index; i < count-1; i++)
        items[i] = items[i + 1];
    count--;
}
```

4.5 Tipi di eccezioni più comuni nella verifica delle precondizioni

Segue un elenco delle eccezioni più comuni usate durante la verifica delle *precondizioni* (nei nomi delle eccezioni ho omesso il suffisso "**Exception**"):

Tipo	Descrizione
`ArgumentNull`	L'argomento è `null`.
`ArgumentOutOfRange`	Il valore dell'argomento non rientra in un determinato intervallo.
`Argument`	L'argomento non soddisfa un certo vincolo (normalmente, descritto nel messaggio associato all'eccezione).
`InvalidOperation`	Dato lo stato dell'oggetto, l'operazione non è valida.
`Format`	L'argomento è una stringa contenente un valore da convertire in un altro tipo (intero, data, tempo, etc), ma non rispetta il corretto formato stabilito per il tipo di destinazione.

Di norma, per i primi due tipi basta specificare il nome dell'argomento che non rispetta la *precondizione*. Negli altri è opportuno specificare un messaggio che descriva la *precondizione* che non è stata soddisfatta.

4.6 *Incapsulamento* e *invariante di classe*

Incapsulamento e *invariante di classe* sono due concetti distinti, anche se correlati. Un *oggetto* può rispettare il principio dell'incapsulamento, ma non l'invariante di classe (vedi la prima versione di di `StringList`). D'altra parte, se un oggetto non dichiara privati i propri campi, non può, per definizione, garantire l'invariante di classe. Infatti, in questo caso il codice esterno ha un accesso diretto allo stato delle istanze e dunque può modificarlo senza che il codice della classe possa verificarne la validità.

(Il discorso cambia se i campi sono *readonly*.)

4.7 Riepilogo

L'invariante di classe fa riferimento alla necessità di progettare le classi in modo che le istanze si trovino in stati validi e producano risultati corretti. Poiché nel progettare una classe non si ha il controllo sull'uso che ne sarà fatto, è necessario che tutti metodi pubblici verifichino che siano soddisfatte le condizioni per eseguire correttamente le operazioni richieste. Se una *precondizione* non è soddisfatta, il metodo deve produrre un'eccezione.

Le *postcondizioni* sono, di fatto, incorporate nei metodi e nel loro corretto (soddisfatte) o scorretto (non soddisfatte) funzionamento. In genere sono verificate mediante l'esecuzione di *unit test*, il cui scopo è, appunto, verificare il funzionamento di una classe in "isolamento", al di fuori dei programmi nei quali viene usata.[1]

1 Alcuni linguaggi di programmazione forniscono dei costrutti specifici per la verifica delle *postcondizioni* e, addirittura, per la verifica formale dell'invariante di classe.

5 Proprietà

Il fatto che l'interfaccia pubblica di una classe sia composta soltanto da metodi è condizione necessaria (anche se non sufficiente) per il rispetto del principio di incapsulamento e dell'invariante di classe. D'altra parte, in molti casi l'uso di un metodo non esprime in modo naturale l'intento di un'operazione. In molti casi, infatti, i metodi rappresentano soltanto un modo per accedere a un'informazione già esistente.

Considera l'interfaccia pubblica di `StringList`:

```
class StringList
{
    public StringList() { ... }
    public int Count() { ... }
    public void Add(string item) { ... }
    public void RemoveAt(int index) { ... }
    ...
}
```

I metodi `Add()` e `RemoveAt()` implementano un'operazione, aggiungere o rimuovere un elemento; il metodo `Count()`, d'altra parte, restituisce semplicemente un'informazione: il numero di elementi nella lista. In questo caso, il metodo esprime un intento di solito associato alle variabili: accedere e/o modificare un dato.

La *proprietà* esistono appunto per colmare il divario tra il costrutto usato (il metodo) e l'intento trasmesso (l'accesso a un dato).

5.1 Proprietà semplici

Le proprietà semplici sono costrutti che forniscono la funzionalità dei metodi insieme alla sintassi delle variabili.

Considera il metodo `Count()`, la cui funzione è restituire il numero di elementi della lista; restituire cioè il valore del campo `count`. Un proprietà permette di ottenere l'identico risultato:

```
class StringList
{
    string[] items;
    int count;
    ...
    public int Count
    {
        get { return count;}
    }
}
```

```
public int Count()
{
    return count;
}
```

Il codice esterno può utilizzare la proprietà come se fosse una variabile:

```
StringList nomi = new StringList();
...
for (int i = 0; i < nomi.Count; i++)
{
    Console.WriteLine(nomi.GetItem(i));
}
```

```
for (int i = 0; i < nomi.Count(); i++)
{
    Console.WriteLine(nomi.GetItem(i));
}
```

5.2 Definire una proprietà

Segue la sintassi (semplificata):

```
<modificatore> <tipo> <nome>
{
    <modificatore>opz get {...}
    <modificatore>opz set {...}
}
```

Opzionali (non entrambi, però)

Riepilogando:

- Diversamente dai metodi, una proprietà non definisce dei parametri.
- Può definire uno o due *accessor*, i quali corrispondono al corpo di un metodo.
- Ogni *accessor* può definire un livello di visibilità più restrittivo di quello della proprietà.

La precedente sintassi viene tradotta dal linguaggio nella definizione di uno o due metodi:

Sintassi proprietà

```
<modificatore> <tipo> <nome>
{
    <modificatore>opz get {...}
    <modificatore>opz set {...}
}
```

Traduzione linguaggio

```
<modificatore> <tipo> get_<nome>()
{
    {...}     get accessor
}
```

```
<modificatore> void set_<nome>(<tipo> value)
{
    {...}     set accessor
}
```

Dunque: il *get accessor* viene tradotto in un metodo senza parametri che restituisce un valore; il *set accessor* viene tradotto in un metodo con un parametro dello stesso tipo della proprietà.

Le proprietà rappresentano un classico esempio di applicazione del principio di incapsulamento: separare la funzione dalla sua implementazione. Di seguito esamino alcuni scenari che ne mostrano l'utilità.

5.3 Accesso in sola lettura a un campo: proprietà *get-only*

La proprietà `Count` regola l'accesso al campo `count`, evitando che il codice esterno possa modificarlo; a questo scopo dispone del solo *get accessor*.

```
int count;
public int Count
{
    get { return count;}
}
```

Proprietà simili vengono definite *get-only*, e sono abbastanza comuni.

5.3.1 Proprietà con "backing field"

Le proprietà come `Count` sono definite anche proprietà con *backing field*; (letteralmente: "campo che sta dietro"); questo termine indica, appunto, il campo privato contenente l'informazione resa accessibile dalla proprietà.

5.4 Restituzione di un valore derivato: proprietà derivata (*get-only)*

In alcuni casi il valore da restituire non è memorizzato in un campo, ma viene prodotto attraverso un calcolo, oppure ottenuto da un altro oggetto.

Supponiamo di voler aggiungere a `StringList` la funzione di restituzione della capacità della lista. Non c'è bisogno di un campo per memorizzare questa informazione, poiché è rappresentata dalla proprietà `Length` del vettore `items`.

```
public int Capacity
{
    get { return items.Length;}
}
```

Dunque, il valore di `Capacity` non è memorizzato su un campo, ma *deriva* da quello ottenuto da un altro oggetto, `items` in questo caso.

5.5 Proprietà "scrivibile": implementazione di *get* e *set accessor*

Nella classe `List<>` la proprietà `Capacity` è modificabile, dunque consente al codice esterno di di aumentare la capacità della lista. Prima di vedere come implementarla in `StringList`, considera l'uso della proprietà:

```
List<string> nomi = new List<string>(10); //: crea lista di capacità iniziale pari a 10
Console.WriteLine(nomi.Capacity);         //-> 10
...
nomi.Capacity = 20;
Console.WriteLine(nomi.Capacity);         //-> 20 (items.Length è 20)
```

È come se il codice esterno vedesse una variabile, mentre la proprietà restituisce e/o modifica la dimensione del vettore contenente gli elementi.

Segue l'implementazione:

```
public int Capacity
{
    get { return items.Length; }
    set
    {
        if (value < Count)
            throw new ArgumentOutOfRangeException(nameof(Capacity));
        Array.Resize(ref items, value);
    }
}
```

La parola chiave `value` rappresenta il parametro nascosto del metodo corrispondente al *set accessor*. Tutto ciò diventa più chiaro guardando in che modo il linguaggio traduce la proprietà:

```
public int Get_Capacity()    // corrisponde al get accessor
{
    return items.Length;
}

public void Set_Capacity(int value)  // corrisponde al set accessor
{
    if (value < Count)
        throw new ArgumentOutOfRangeException(nameof(Capacity));
    Array.Resize(ref items, value);
}
```

Naturalmente, il linguaggio traduce anche il codice che usa la proprietà:

Codice del programmatore

```
StringList nomi = new StringList(10);
Console.WriteLine(nomi.Capacity);
...
nomi.Capacity = 20;
Console.WriteLine(nomi.Capacity);
```

Traduzione del compilatore

```
StringList nomi = new StringList(10);
Console.WriteLine(nomi.Get_Capacity());
...
nomi.Set_Capacity(20);
Console.WriteLine(nomi.Get_Capacity());
```

5.6 Proprietà automatiche

In `StringList`, la proprietà `Count` fa da semplice *wrapper* (involucro) al campo `count`, senza eseguire codice o imporre alcuna *precondizione*. Si tratta di uno scenario comune, nel quale la proprietà funge da alias per un campo. In scenari simili, l'uso di *proprietà automatiche* semplifica il codice da scrivere.

Una *proprietà automatica* gestisce autonomamente il *backing field*, in modo che il codice possa fare riferimento unicamente alla proprietà. La sintassi ricalca quella di una normale proprietà, con la differenza che gli *accessor* sono vuoti:

```
<modificatore> <tipo> <nome>
{
    <modificatore>opz get;
    <modificatore>opz set;
}
```

opzionali

5.6.1 Definire Count come proprietà automatica

Il campo `count` scompare; resta la proprietà, che definisce un *set accessor* privato in modo che la proprietà sia modificabile soltanto all'interno della classe:

```
int count;
public int Count
{
    get;
    private set;
}
```

Il funzionamento della proprietà diventa chiaro analizzando la traduzione operata dal linguaggio:

Proprietà

```
public int Count
{
    get ;
    private set;
}
```

Traduzione linguaggio

```
// creato dal linguaggioint
<Count>k__BackingField;

public int Get_Count()
{
    return <Count>k__BackingField;
}


private void Set_Count(int value)
{
    <Count>k__BackingField = value;
}
```

Il codice a destra mostra come una proprietà automatica sia equivalente a una proprietà con *backing field*; in questo caso, però, il campo privato è generato automaticamente dal linguaggio e non è utilizzabile nel codice (che deve utilizzare la proprietà).

(Il nome del campo utilizza volutamente dei caratteri che in C# non sono ammessi come parte di un identificatore. Ciò garantisce che il nome non possa andare in conflitto con i nomi di altri campi definiti dal programmatore.)

6 Proprietà indicizzate

Mentre le proprietà semplici sono largamente impiegate nella programmazione, le *proprietà indicizzate*, o *indicizzatori*, trovano il loro naturale impiego soltanto in *oggetti* che rappresentano collezioni di elementi, accessibili mediante un indice o una chiave.

Un *indicizzatore* fornisce un meccanismo di accesso che impiega la sintassi usata dai vettori. In modo analogo alle proprietà semplici, è un costrutto che unisce le funzioni di due metodi.

6.1 Definire un' indicizzatore

Segue la sintassi (semplificata):

```
<modificatore> <tipo> this[<parametri>]
{
    <modificatore>opz get  {...}
    <modificatore>opz set  {...}
}
```

opzionali

Nota bene:

- L'indicizzatore non ha un nome; è sostituito dalla parola chiave `this`.
- Tra parentesi quadre specifica uno o più parametri (in genere uno), che sono usati come chiavi per accedere all'elemento.
- Può definire uno o due *accessor*, corrispondenti al corpo dei metodi che hanno la funzione di restituire o impostare l'elemento.
- Ogni *accessor* può definire un livello di visibilità più restrittivo di quello generale.

La sintassi viene tradotta dal linguaggio nel seguente modo:

Sintassi proprietà indicizzata

```
<modificatore> <tipo> this[<parametri>]
{
    <modificatore>opz get  {...}
    <modificatore>opz set  {...}
}
```

Traduzione linguaggio

```
<modificatore> <tipo> get_Item(<parametri>)
{
    {...}     get accessor
}
```

```
<modificatore> void set_Item(<parametri>,
                             <tipo> value)
{
    {...}     set accessor
}
```

6.2 Implementare un indicizzatore in StringList

Attualmente `StringList` è indicizzabile attraverso i metodi `GetItem()` e `SetItem()`. Il loro uso è poco naturale, poiché ci si aspetta di poter usare una collezione nello stesso modo in cui si usa un *array*.

Segue l'implementazione di un *indicizzatore* e la sua traduzione operata dal linguaggio:

Indicizzatore

```
public string this[int index]
{
    get
    {
        ValidateIndex(index);
        return items[index];
    }
    set
    {
        ValidateIndex(index);
        items[index] = value;
    }
}
```

Traduzione del linguaggio

```
public string Get_Item(int index)
{
    ValidateIndex(index);
    return items[index];
}

public void Set_Item(int index, string value)
{
     ValidateIndex(index);
     items[index] = value;
}
```

Dopo questa modifica, è possibile scrivere il seguente codice (a destra la traduzione operata dal linguaggio):

Codice del programmatore

```
StringList nomi = new StringList();
nomi.Add("Gianni");
nomi.Add("Sara");
 nomi[0] = "Andrea";
for (int i = 0; i < nomi.Count; i++)
{
    Console.WriteLine(nomi[i]);
}
```

Traduzione del linguaggio

```
StringList nomi = new StringList();
nomi.Add("Gianni");
nomi.Add("Sara");
 nomi.Set_Item(0, "Andrea");
for (int i = 0; i < nomi.Count; i++)
{
    Console.WriteLine(nomi.Get_Item(i));
}
```

6.3 Conclusioni

Quello mostrato è un esempio tipico di *proprietà indicizzata*, dove la chiave è, appunto, un indice intero. Ma non esistono vincoli su numero e tipo delle chiavi. Ad esempio, in un dizionario la chiave potrebbe essere di tipo stringa, o un altro tipo.

Analogamente alle proprietà semplici, anche gli *indicizzatori* sono costrutti sintattici che non aggiungono niente alle funzioni svolte dai metodi. Semplicemente: forniscono una sintassi d'uso che esprime con maggior naturalezza l'intento dell'operazione svolta.

7 Oggetti “vs” record

Quando si crea una classe occorre innanzitutto interrogarsi sul ruolo che svolge: rappresentare dati o implementare funzioni; infatti, dato che C# non fa alcuna distinzione tra *record* e *oggetti*, è possibile definire classi che, pur corrette dal punto di vista logico, sono il frutto di un progetto incoerente.

Di seguito porterò alcuni esempi di buona e cattiva progettazione.

7.1 Poligono regolare: *record* vs *oggetto*

Ipotizza di dover realizzare un programma che esegua alcuni calcoli su un insieme di poligoni regolari. Per rappresentare un poligono uso un *record* che definisce i campi fondamentali – *lato* e *n°lati* – e i campi *area* e *perimetro*, utilizzati nel programma.

```
public class Poligono
{
    public int NumLati;
    public double Lato;
    public double Area;
    public double Perimetro;
}
```

Si tratta di una scelta valida e coerente: il *record* definisce i dati del poligono, altrove si colloca il codice che li elabora – caricamento e calcolo di area e perimetro – e che li usa: visualizzazione, etc.

7.1.1 Transizione da record a oggetto: un primo tentativo

Poiché area e perimetro del poligono dipendono unicamente dagli altri due campi, diventa abbastanza naturale collocare nella classe il codice che calcola entrambi, evitando che sia il codice esterno a doverlo fare.

Vi sono varie soluzioni; ne mostro una che presenta molte problematiche:

```
public class Poligono
{
   //valori usati per calcolare le aree: triangolo <-> decagono
    double[] kPoligoni = { 0.433, 1, 1.720, 2.598, 3.634, 4.828, 6.182, 7.694 };

    public int NumLati;
    public double Lato;
    public double Area;
    public double Perimetro;
    public void CalcolaArea()
    {
        Area = Lato * Lato * kPoligoni[NumLati - 3];
    }

    public void CalcolaPerimetro()
    {
```

```
        Perimetro = Lato * NumLati;
    }
}
```

La classe è logicamente corretta, ma è frutto di un progetto scadente:

1. Ha una struttura più complessa: sei membri invece di quattro. (Non conto il vettore dei coefficienti usati per calcolare l'area, perché è privato.)
2. È più complicata da usare: il codice esterno deve prima creare l'istanza, inizializzando `Lato` e `NumLati`, e successivamente chiamare i metodi che calcolano area e perimetro.
3. Può indurre dei bug nel codice esterno. Considera il seguente codice:

   ```
   Poligono p = new Poligono();
   p.CalcolaArea();    // -> troppo presto: IndexOutOfRangeException!
   p.Lato = 10;
   p.NumLati = 3;
   p.CalcolaPerimetro();
   ```

   In pratica, la correttezza dei calcoli dipende dall'ordine con il quale vengono inizializzati i campi e chiamati i metodi. (È la ricetta ideale per il disastro.)
4. I campi, compresi `Area` e `Perimetro`, sono modificabili dal codice esterno, il quale può ignorare entrambi i metodi e calcolare direttamente i valori (E quindi: chi ha la responsabilità di calcolare area e perimetro del poligono?)

7.1.2 Migliorare la classe: costruire correttamente l'oggetto

È possibile risolvere i problemi descritti nei punti 1-3 aggiungendo un costruttore che richieda i parametri fondamentali di un poligono, *lato* e *n°lati*. Nel costruttore saranno calcolati anche `Area` e `Perimetro`, in modo che, appena creata, l'istanza memorizzi già tutti i valori necessari, senza bisogno di eseguire alcun metodo.

```
public class Poligono
{
   //usati per calcolare le aree: triangolo <-> decagono
    double[] kPoligoni = { 0.433, 1, 1.720, 2.598, 3.634, 4.828, 6.182, 7.694 };

    public int NumLati;
    public double Lato;
    public double Area;
    public double Perimetro;

    public Poligono(int numLati, double lato)
    {
        NumLati = numLati;
        Lato = lato;
        Area = Lato * Lato * kPoligoni[NumLati - 3];
        Perimetro = Lato * NumLati;
    }
}
```

Ora, la classe è molto più semplice da utilizzare:

```
Poligono p = new Poligono(3, 10);  //n°lati: 3    lato: 10
//-> p.Area = 43.3;   p.Perimetro = 30;
```

ma presenta ancora delle problematiche, poiché i campi sono modificabili dal codice esterno anche dopo la creazione dell'oggetto. Dunque, nulla impedisce di scrivere, ad esempio:

```
Poligono p = new Poligono(3, 10);  //n°lati: 3    lato: 10
...
p.Lato = 20;
//-> p.Area = 43.3;   p.Perimetro = 30... non sono coerenti con il nuovo lato!
```

In sostanza, la classe rende possibile scrivere del codice logicamente corretto, che produce però risultati incoerenti!

7.1.3 Implementare l'incapsulamento e l'invariante di classe

Implementare un *oggetto* significa scrivere *una classe completamente responsabile del proprio funzionamento*, indipendentemente dal codice esterno che la usa.

```
public class Poligono
{
    //usati per calcolare le aree: triangolo <-> decagono
    double[] kPoligoni = { 0.433, 1, 1.720, 2.598, 3.634, 4.828, 6.182, 7.694 };

    public readonly int NumLati;
    public readonly double Lato;
    public readonly double Area;
    public readonly double Perimetro;

    public Poligono(int numLati, double lato)
    {
        if (numLati < 3)
            throw new ArgumentException("Un poligono deve avere almeno 3 lati");
        if (numLati > 10)
            throw new ArgumentException("Non sono supportati poligoni con più di 10 lati");
        if (lato <= 0)
            throw new ArgumentException("Il lato deve essere maggiore di zero");

        NumLati = numLati;
        Lato = lato;
        Area = Lato * Lato * kPoligoni[NumLati - 3];
        Perimetro = Lato * NumLati;
    }
}
```

Il costruttore verifica innanzitutto se i valori forniti sono validi (*precondizioni*); in caso contrario interrompe la creazione dell'istanza. I campi sono *readonly* e dunque, una volta inizializzati nel costruttore, non possono essere più modificati.

In conclusione: non è possibile usare in modo scorretto, oppure ottenere valori incoerenti, con questa versione della classe.

7.1.4 "Rilassare" la verifica sul n° dei lati: uso di proprietà derivate

`Poligono` è piuttosto restrittiva nella verifica di `numLati`; interrompe la creazione dell'istanza se il valore è maggiore di 10, poiché, con l'attuale implementazione, la classe non sarebbe in grado di calcolare l'area. Ma si possono immaginare applicazioni che trarrebbero comunque vantaggio dall'uso di `Poligono` per rappresentare poligoni con 11 e più lati, perché non richiedono il calcolo dell'area, oppure perché potrebbero calcolarlo esternamente usando la formula generale, valida per qualsiasi poligono.

In questo caso è utile verificare la *precondizione* n°lati <= 10 soltanto quando viene richiesta la funzione che la rende necessaria. Si può farlo sostituendo i campi *readonly* con proprietà derivate, che restituiscono il valore calcolandolo "al volo" (per coerenza, ho trasformato in proprietà derivata anche `Perimetro`):

```
public class Poligono
{
    double[] kPoligoni = { 0.433, 1, 1.720, 2.598, 3.634, 4.828, 6.182, 7.694 };

    public readonly int NumLati;
    public readonly double Lato;

    public Poligono(int numLati, double lato)
    {
        if (numLati < 3)
            throw new ArgumentException("Un poligono deve avere almeno 3 lati");
        if (lato <= 0)
            throw new ArgumentException("Il lato deve essere maggiore di zero");

        NumLati = numLati;
        Lato = lato;
    }

    public double Area
    {
        get
        {
            if (NumLati > 10)
                throw new InvalidOperationException("Non è possibile calcolare l'area...");
            return Lato * Lato * kPoligoni[NumLati - 3];
        }
    }
    public double Perimetro
    {
        get { return Lato * NumLati; }
    }
}
```

Nota bene: dopo queste modifiche, la classe è strutturalmente diversa dalla versione precedente, poiché ha due campi soltanto e occupa meno della metà di memoria; ma non richiede alcuna modifica al codice esterno che la usa.[2] Sono i vantaggi dell'incapsulamento e dell'uso di proprietà.

2 In realtà, esistono alcuni scenari nei quali il cambiamento da campi *readonly* a proprietà derivate (o *readonly*) produrrebbe un *breaking change*, ma sono casi minoritari.

7.2 Uso (e abuso) di costruttori nei *record*

Poiché C# non fa distinzione tra *record* e *oggetti*, ci si può chiedere se abbia senso implementare i costruttori nei *record*. Ebbene, nella maggior parte dei casi sono completamente superflui, ma in alcuni scenari possono risultare utili.

Considera il *record* `Alunno`, nel quale ho definito un costruttore:

```
public class Alunno
{
    public string Nominativo;
    public string Classe;
    public Alunno(string nominativo, string classe)
    {
        Nominativo = nominativo;
        Classe = classe;
    }
}
```

Il costruttore è inutile, perché ha soltanto lo scopo di inizializzare i campi, cosa già possibile con l'istruzione di inizializzazione:

Costruttore

```
Alunno a = new Alunno("Filippo Rossi", "3A");
```

Inizializzazione

```
Alunno a = new Alunno
{
    Nominativo = "Filippo Rossi",
    Classe = "3A"
};
```

La seconda forma è più prolissa, ma anche più esplicita, ed evita errori come il seguente:

```
Alunno a = new Alunno("3A", "Filippo Rossi");   // attenzione: scambio degli argomenti!
```

Errori che possono essere evitati anche con l'uso del costruttore e il passaggio esplicito dei parametri:

```
Alunno a = new Alunno               // nota bene: sono parentesi tonde;
(                                   // viene chiamato il costruttore
    Nominativo: "Filippo Rossi",
    Classe: "3A"
);
```

Ma, in questo caso, senza alcun guadagno di sintesi.

In conclusione, il costruttore ha la funzione di eseguire la logica di inizializzazione di un *oggetto*, validando gli argomenti, se vengono forniti. Nei *record* ciò si riduce alla semplice assegnazione ai campi, e dunque è quasi sempre inutile.

7.2.1 Semplificare la creazione di record "ben definiti"

Alcuni *record* rappresentano concetti universalmente noti; in questi casi i costruttori possono essere una buona alternativa all'inizializzazione.

Considera il tipo `Punto`:

```
public class Punto
{
    public double X;
    public double Y;
    public Punto(double x, double y)
    {
        X = x;
        Y = y;
    }
}
```

La creazione mediante il costruttore risulta più semplice e non presenta ambiguità, poiché un punto è universalmente rappresentato da una coppia di coordinate, X e Y.

Uso del costruttore	Uso inizializzatore
`Punto p = new Punto(1, -5);`	`Punto p = new Punto {X = 1, Y = -5};`

Un discorso analogo si potrebbe fare per il tipo `Angolo`:

```
public class Angolo
{
    public double Gradi;
    public double Primi;
    public double Secondi;
    public Angolo(double gradi, double primi, double secondi)
    {
        Gradi = gradi;
        Primi = primi;
        Secondi = secondi;
    }
}
```

Si tratta comunque di eccezioni; in generale si dovrebbero definire dei costruttori soltanto se:

- È richiesta una logica di inizializzazione.
- È necessario validare gli argomenti.
- Occorre costruire un *oggetto*/*record* immutabile.

7.2.2 Record immutabili: obbligo di inizializzazione mediante costruttore

Un *record* (o un *oggetto*) immutabile rende impossibile modificare le istanze dopo che sono state create. Ciò si ottiene mediante campi o proprietà *readonly*, i quali possono essere assegnati solo nel costruttore e non mediante l'istruzione di inizializzazione.

Segue una doppia versione immutabile del record `Punto`, la prima che usa campi *readonly*, la seconda che usa proprietà *readonly*:

Campi *readonly*

```
public class Punto
{
    public readonly double X;
    public readonly double Y;
    public Punto(double x, double y)
    {
        X = x;
        Y = y;
    }
}
```

Proprietà *readonly*

```
public class Punto
{
    public double X { get; }
    public double Y { get; }
    public Punto(double x, double y)
    {
        X = x;
        Y = y;
    }
}
```

(8)

7.3 Proprietà derivate nei *record*

I *record* dovrebbero limitarsi a definire dei dati, senza implementare alcuna logica, ma in alcuni scenari può essere utile usare proprietà derivate per rappresentare informazioni non memorizzate nei campi. Considera il *record* Dipendente:

```
public class Dipendente
{
    public string Nome;
    public string Cognome;
    public DateTime DataNascita;
}
```

Supponi che l'applicazione richieda anche di conoscere il nome completo e l'età del dipendente. Sono informazioni facilmente ricavabili, ma poiché derivano dai campi del *record*, può essere utile che sia `Dipendente` a fornirle:

```
public class Dipendente
{
    public string Nome;
    public string Cognome;
    public DateTime DataNascita;

    public string Nominativo { get { return $"{Cognome},{Nome}"; } }

    public int Età {get { return DateTime.Now.Year - DataNascita.Year; }}
}
```

7.4 *Record che definiscono proprietà automatiche*

Poiché le proprietà sono assimilabili a metodi e poiché i *record* devono limitarsi a definire dei dati, non si vede alcun motivo, né vantaggio, nell'usare delle proprietà automatiche invece di semplici campi.

Considera il *record* `Studente`, definito in due versioni:

Uso di campi

```
public class Studente
{
    public string Nominativo;
    public string Classe;
}
```

Uso di proprietà automatiche

```
public class Studente
{
    public string Nominativo {get; set;}
    public string Classe {get; set;}
}
```

Le due versioni sono identiche nella sostanza; in generale, dunque, è preferibile la prima, meno prolissa. Ma esistono scenari nei quali è vantaggioso, se non addirittura necessario, utilizzare la seconda forma.[3] Qui mi limito a considerare gli scenari che ricadono nel cosiddetto *data-binding*

7.4.1 Data-binding

Il termine *data-binding* designa la capacità di molti controlli di applicazioni GUI di visualizzare automaticamente i dati memorizzati nelle istanze di *record* e *oggetti*, purché siano utilizzate delle proprietà automatiche.

Supponi, ad esempio, di voler visualizzare un elenco di studenti mediante un *listbox*. L'approccio "ingenuo" è il seguente:

```
List<Studente> studenti = new List<Studente>();
...
lstStudenti.Items.Clear();
foreach(var s in studenti)
{
    lstStudenti.Items.Add(s.Nominativo);  // "items" colleziona delle stringhe
}
```

Ma, se il *record* usa delle proprietà, è possibile sfruttare la funzione di *data-binding*:

```
List<Studente> studenti = new List<Studente>();
...
lstStudenti.DisplayMember = "Nominativo";  // stabilisce quale dato visualizzare
lstStudente.DataSource = studenti;         // items collezionerà degli studenti
```

Sarà il *listbox* a scorrere la lista `studenti` e, per ogni elemento, a "estrarre" il dato memorizzato in `Nominativo`.

Un grande vantaggio di questa tecnica, oltre all'estrema semplicità di applicazione, risiede nel fatto che mentre nel primo caso `lstStudenti.Items` memorizza soltanto i nomi degli studenti, nel secondo caso memorizza gli studenti veri e propri. Ciò significa, ad esempio, che l'uso di `lstStudenti.SelectedItem` produce lo studente selezionato, e non soltanto il suo nome.

3 Se un *record* è destinato ad essere usato in più applicazioni (cioè, ad essere usato pubblicamente), si preferisce in genere l'uso di proprietà automatiche.

8 *Oggetti* immutabili

Un *oggetto* è *immutabile* quando impedisce che le istanze, una volta create, possano essere modificate.

Un simile *oggetto* è utile in molti scenari, poiché consente a parti diverse del programma di utilizzare la stessa istanza con la garanzia che non vi siano effetti collaterali causati da modifiche ai dati. Ciò è ancora più importante in scenari *multi-threading*.

Di seguito mostro un semplice esempio, che ricade in un pattern molto comune, che è quello di caricare dei dati con la garanzia che in seguito non possano essere modificati, ma solo utilizzati.

8.1 Collezione di *oggetti* immutabili

Ipotizza di avere un file di testo contenente una serie di dati nella forma:

<nome-dato> = <valore

```
#Gol realizzati
Messi=12
Ronaldo=10
...
```

I dati devono essere caricati in memoria allo scopo di produrre alcune statistiche; l'approccio migliore è quello di ottenere una collezione di *oggetti*.

```
class Giocatore
{
    public readonly string Nome;
    public readonly int Gol;

    public Giocatore(string testo)  //esempio: "Messi = 12"
    {
        var campi = testo.Split(new char[] { '=' }, StringSplitOptions.RemoveEmptyEntries);
        if (campi.Length != 2)
            throw new FormatException($"Numero campi non valido: [{testo}]");

        Nome = campi[0].Trim();
        int gol = int.Parse(campi[1]);
        if (gol < 0)
           throw new ArgumentOutOfRangeException($"N° gol < 0: [{testo}]");
        Gol = gol;
    }
}
```

(Nota bene: il costruttore riceve una riga del file ed estrae e valida i dati)

Essendo `Nome` e `Gol` decorati con `readonly`, è possibile assegnarli soltanto nel costruttore; dopodiché non possono essere più modificati. (È l'unica eccezione alla regola di non avere campi pubblici negli *oggetti*.)

Alternativamente, si può decidere di lasciare al codice esterno il compito di estrarre i dati; ciò evita che la classe `Giocatore` dipenda da un particolare formato di memorizzazione dei dati:

```
class Giocatore
{
    public readonly string Nome;
    public readonly int Gol;

    public Giocatore(string nome, int gol)
    {
        if (string.IsNullOrEmpty(nome))
            throw new ArgumentException($"Il nome è vuoto o nullo");

        if (gol < 0)
           throw new ArgumentOutOfRangeException($"N° gol < 0: [{testo}]");
        Gol = gol;
    }
}
```

8.1.1 Uso di proprietà read-only

Alternativamente ai campi è possibile usare delle proprietà *readonly*:

```
class Giocatore
{
    public string Nome {get;}
    public int Gol {get;}

    public Giocatore(string nome, int gol)
    {
        ...
    }
}
```

9 *Oggetti* con membri statici

Siamo abituati a considerare i membri statici – campi e metodi – come una caratteristica specifica dei *moduli*, ma anche gli *oggetti* possono definirli. Di seguito, attraverso degli esempi, mostrerò il ruolo che possono svolgere e i vincoli ai quali sono sottoposti.

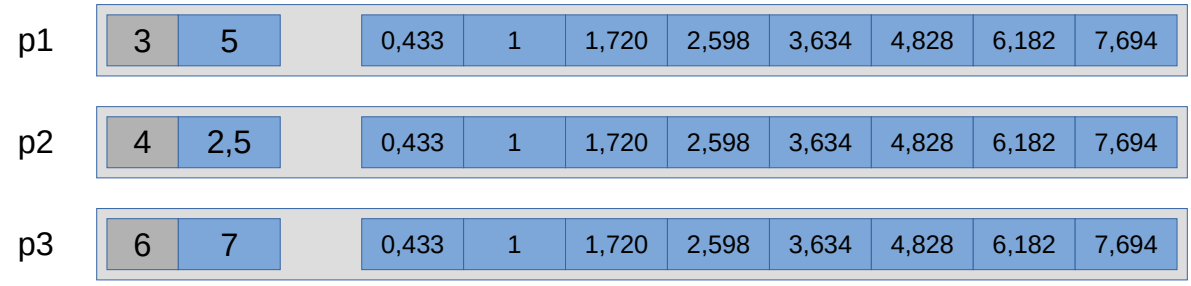
9.1 Poligono: vettore dei coefficienti statico

In 7.1.4 la classe `Poligono` definisce tre campi: `Lato`, `NumLati` e il vettore `kPoligoni`, contenente i coefficienti utilizzati per calcolare l'area con la formula breve. Mentre i primi due campi sono specifici di ogni istanza, il vettore è lo stesso per tutte le istanze, poiché i coefficienti valgono per qualsiasi poligono. Ciò solleva una questione sull'efficienza della classe.

Considera il seguente codice, che crea tre istanze della classe:

```
Poligono p1 = new Poligono(3, 5);    //n°lati: 3    lato: 5
Poligono p2 = new Poligono(4, 2.5);  //n°lati: 4    lato: 2,5
Poligono p3 = new Poligono(6, 7);    //n°lati: 6    lato: 7
```

In memoria, la situazione può essere schematizzata in questo modo:



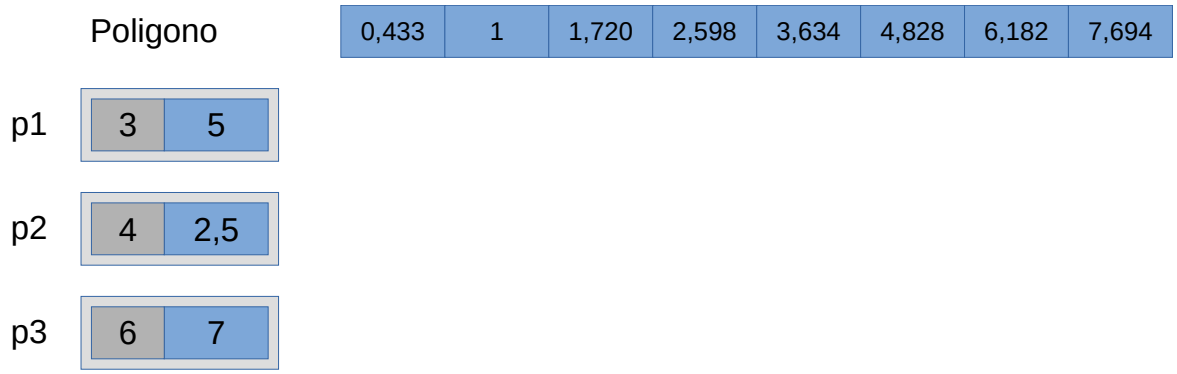
Siamo di fronte a un grande spreco di memoria, dato che basterebbe un unico vettore, utilizzato da tutte le istanze. Ebbene, per ottenere questo risultato, basta dichiarare il vettore `static`:

```
public class Poligono
{
    static double[] kPoligoni = { 0.433, 1, 1.720, 2.598, 3.634, 4.828, 6.182, 7.694 };

    public readonly int NumLati;
    public readonly double Lato;
    ...
}
```

Ora che è un membro statico, il vettore non appartiene più a nessuna istanza e dunque viene creato una volta per tutte, la prima volta che viene richiesto in una qualunque istanza della classe.

Lo schema a pagina successiva mostra la nuova situazione in memoria:



Dunque, il campo `kCoefficienti` si comporta come il campo di un *modulo*: è unico all'interno dell'intera applicazione. Essendo privato, inoltre, resta inaccessibile al codice esterno alla classe.

9.2 Fornire l'accesso a istanze predefinite

Per alcuni tipi di dati esistono dei valori predefiniti utilizzati in molte applicazioni. Si pensi ai valori minimo e massimo del tipo `int`, i quali possono essere usati per la ricerca del minimo e del massimo. Ebbene, tali valori sono -2147483648 e 2147483647, ma non esiste alcun bisogno di conoscerli, perché il tipo `int` fornisce due campi statici che li memorizzano: `MinValue` e `MaxValue`.

Considera il tipo `Angolo`, introdotto come *record* in 7.2.1. Alcuni angoli sono ben conosciuti e ampiamente utilizzati nel software, quindi è utile renderli immediatamente disponibili, evitando al codice esterno l'onere di crearli.

In questa nuova versione della classe definisco due campi statici che memorizzano istanze predefinite contenenti l'angolo *piatto* e quello *giro*.

```
public class Angolo
{
    public static readonly Angolo Piatto = new Angolo(180, 0, 0);
    public static readonly Angolo Giro = new Angolo(360, 0, 0);

    public readonly int Gradi;
    public readonly int Primi;
    public readonly int Secondi;
    public Angolo(int gradi, int primi, int secondi)
    {
        //... valida parametri
        Gradi = gradi;
        Primi = primi;
        Secondi = secondi;
    }
}
```

Nota bene: i due campi sono *readonly*; ciò impedisce al codice esterno di modificarli, facendo sì che referenzino un angolo diverso.

Nel codice applicativo, quando si vuole ottenere un angolo giro, si può scrivere:

```
Angolo angolo = Angolo.Giro;
```

Questa soluzione, oltre a semplificare il codice, aumenta l'efficienza quando gli angoli predefiniti sono utilizzati più volte all'interno del programma, come mostra il seguente confronto[4]:

Creazione tre istanze (3 x 24 = 72 byte)

```
Angolo p1 = new Angolo(180, 0, 0);
Angolo p2 = new Angolo(180, 0, 0);
Angolo p3 = new Angolo(180, 0, 0);
```

Condivisione unica istanza predefinita (24 byte)

```
Angolo p1 = Angolo.Piatto;
Angolo p2 = Angolo.Piatto;
Angolo p3 = Angolo.Piatto;
```

Il frammento a destra crea tre istanze di angolo piatto, mentre il frammento a sinistra assegna a tre variabili il riferimento all'unica istanza creata. Dal punto di vista esterno non cambia niente: tre variabili del programma referenziano un angolo piatto.

(Nota bene: una soluzione simile, molto utilizzata nella programmazione, è accettabile soltanto se l'*oggetto* è immutabile.)

9.3 Implementare un contatore di istanze

Immagina di voler conoscere, in qualsiasi momento, quante istanze della classe `Poligono` sono state create. Una soluzione molto semplice è far sì che sia la stessa classe a tenere traccia di ogni istanza creata, utilizzando un contatore che viene incrementato nel costruttore. Ma perché ciò possa funzionare, è necessario che il contatore sia unico e condiviso da tutte le istanze, cioè statico:

```
public class Poligono
{
    static double[] kPoligoni = { 0.433, 1, 1.720, 2.598, 3.634, 4.828, 6.182, 7.694 };

    public readonly int NumLati;
    public readonly double Lato;

    public Poligono(int numLati, double lato)
    {
        ...
        numIstanze++;
    }

    static int numIstanze;
    public static int NumIstanze {get {return numIstanze;} }
    ...
}
```

Nota bene: la classe definisce un campo statico privato e una proprietà statica pubblica che restituisce il campo, in modo che sia inaccessibile al codice esterno.

Si sarebbe potuto ottenere lo stesso risultato con una proprietà automatica con *set accessor* privato:

4 Il realtà il risparmio in termini di memoria è superiore, poiché per ogni istanza sono è riservato uno spazio di memoria necessario al suo funzionamento.

```
public class Poligono
{
    ...
    public static int NumIstanze {get; private set; }


    public Poligono(int numLati, double lato)
    {
        ...
        NumIstanze++;
    }
    ...
}
```

Dopo questa modifica, il codice esterno potrà sempre conoscere il numero di istanze create, indipendentemente da dove vengano memorizzate:

```
Poligono p1 = new Poligono(3, 5);
Poligono p2 = new Poligono(4, 2.5);
Poligono p3 = new Poligono(6, 7);

var poligoni = new List<Poligono>();
poligoni.Add(new Poligono(3, 5));
poligoni.Add(new Poligono(7, 1.5));

Poligono p4 = new Poligono(8, 5);

Console.WriteLine(Poligono.NumPoligoni);  //-> 6
```

Nota bene: come per qualsiasi membro statico, il codice esterno accede alla proprietà mediante il nome della classe.

9.4 Metodi di servizio: *parsing*

Alcuni tipi di dati – `int`, `double`, `DateTime`, `TimeSpan`, etc – rappresentano dei valori che possono essere convertiti in stringa. Per questo motivo definiscono un metodo, `Parse()`, che esegue l'operazione inversa: data una stringa, crea un'istanza con il valore corrispondente. Un metodo simile non può essere di istanza, poiché il suo scopo, appunto, *è quello di crearla un'istanza*.

Supponi di voler fornire una rappresentazione stringa per il tipo `Angolo`, precisamente:

"<gradi>;<primi>;<secondi>"

In questo caso è opportuno fornire un metodo di *parsing* che crei un'istanza di tipo `Angolo` data la sua rappresentazione stringa:

```
public class Angolo
{
    public static readonly Angolo Piatto = new Angolo(180, 0, 0);
    public static readonly Angolo Giro = new Angolo(360, 0, 0);

    public readonly int Gradi;
```

```
    public readonly int Primi;
    public readonly int Secondi;
    public Angolo(int gradi, int primi, int secondi)
    {
        //... valida parametri
        Gradi = gradi;
        Primi = primi;
        Secondi = secondi;
    }

    public static Angolo Parse(string valore)
    {
        string[] campi = valore.Split(';');
        if (campi.Length != 3)
            throw new FormatException("Formato del valore errato");
        return new Angolo(int.Parse(campi[0]),
                          int.Parse(campi[1]),
                          int.Parse(campi[2]));
    }
}
```

Il codice esterno può eseguire il metodo attraverso il nome della classe:

```
Angolo a =  Angolo.Parse("90;0;0");  // ->90°
```

9.5 *Vincoli sui metodi statici*

Poiché un metodo statico non viene eseguito attraverso un'istanza della classe, *non può, per definizione, accedere ai membri d'istanza*, mentre può utilizzare eventuali membri statici.

Considera l'ipotesi di aggiungere un metodo statico `Trasla()` alla classe `Punto`, con la funzione di modificare le coordinate di un valore stabilito.

```
public class Punto
{
    public double X {get; private set;}
    public double Y {get; private set;}
    public Punto(double x, double y)
    {
        X = x;
        Y = y;
    }
    public static void Trasla(double dx, double dy)
    {
        X += dx;    //errore: an object reference is required for non-static field...
        Y += dy;    //errore: an object reference is required for non-static field...
    }
}
```

Per comprendere il tipo di errore, considera il seguente codice:

```
Punto.Trasla(10, 10);  //non c'è istanza che dovrebbe memorizzare le coordinate!
```

Il metodo dovrebbe modificare le coordinate X,Y di un'istanza di `Punto`, ma non esiste alcuna istanza! Infatti, il codice dovrebbe essere simile al seguente:

```
Punto p = new Punto(0, 0);  //X: 0   Y: 0
p.Trasla(10, 10);           //X: 10   Y: 10
```

Ma questo è possibile soltanto se `Trasla()` è un metodo d'istanza.

(L'argomento è analizzato in maggior profondità nel tutorial **Oggetti undercover**.)

9.6 Operatori

Gli operatori – `+`, `-`, `/`, `%`, etc – sono assimilabili a metodi statici che definiscono uno o due parametri (in base all'*arità* dell'operatore) e restituiscono un risultato. Per ogni tipo predefinito è fornita la versione specifica degli operatori applicabili. Dunque, ad esempio, per i tipi `double` e `int` esiste una versione specifica dell'operatore `+`, in modo che, nelle espressioni, sia eseguita l'operazione corretta, somma intera o somma in virgola mobile.

Ebbene, posti alcuni vincoli, C# consente di utilizzare gli operatori anche con i tipi definiti dal programmatore (*operator's overloading*).

9.6.1 Confronto per uguaglianza tra istanze: operatori == e !=

Per default, il confronto tra due variabili *oggetto* (o *record*) stabilisce se le variabili referenziano la stessa istanza e non se due istanze sono uguali; ma in molti casi è proprio questa l'informazione richiesta. Naturalmente si può fornire questa funzione con un metodo, come mostro di seguito prendendo a esempio la classe `Angolo`:

```
public class Angolo
{
    ...
    public bool UgualeA(Angolo a)
    {
        return Gradi == a.Gradi && Primi == a.Primi && Secondi == a.Secondi;
    }
}
```

Per verificare se due angoli sono uguali, occorre scrivere:

```
Angolo a1 =  new Angolo(180, 15, 35);
Angolo a2 =  new Angolo(180, 15, 35);
if (a1.UgualeA(a2))      //-> true
{
    ...
}
```

Si tratta però di una soluzione poco elegante e pratica, soprattutto se applicata a più operazioni e tipi di dati (sarebbe un proliferare di metodi). Inoltre, quando si confrontano due valori ci si aspetta di poter usare l'operatore `==`:

```
Angolo a1 =  new Angolo(180, 15, 35);
Angolo a2 =  new Angolo(180, 15, 35);
if (a1 == a2)      //per default -> false (ci aspetterebbe true)
{
    ...
}
```

In pratica, dal confronto per uguaglianza ci si aspetta di verificare se `a1` e `a2` memorizzano lo stesso angolo, ma l'operatore `==` stabilisce se `a1` e `a2` referenziano lo stesso oggetto.

Per ottenere questo intento, è necessario ridefinire l'operatore `==` nella classe `Angolo`:

```
public class Angolo
{
    ...
    public static bool operator == (Angolo l, Angolo r) // "l": left   "r": right
    {
        return l.Gradi == r.Gradi && l.Primi == r.Primi && l.Secondi == r.Secondi;
    }
}
```

Come si vede, l'operatore è rappresentato da un metodo statico, con la differenza che il nome del metodo è sostituito dalla parola chiave `operator` seguita dal simbolo dell'operatore.

La soluzione presentata è corretta, ma incompleta. Allo scopo di garantire un uso coerente degli operatori, C# nel caso di ridefinizione di ==, impone di ridefinire anche `!=`.

```
public class Angolo
{
    ...
    public static bool operator == (Angolo l, Angolo r)
    {
        return l.Gradi == r.Gradi && l.Primi == r.Primi && l.Secondi == r.Secondi;
    }

    public static bool operator != (Angolo l, Angolo r)
    {
        return !(l == r);
    }
}
```

Ora, il seguente codice produce il risultato atteso:

```
Angolo a1 =  new Angolo(180, 0, 0);
Angolo a2 =  new Angolo(180, 0, 0);
Angolo a3 =  new Angolo(360, 0, 0);

bool f1 = a1 == a2;  //-> true;
bool f2 = a1 == a3;  //-> false;
bool f3 = a1 != a2;  //-> false;
```

10 Principi di progettazione: S.R.P.

Nei precedenti paragrafi ho introdotto le regole del linguaggio, il principio di *incapsulamento* e l'*invariante di classe*. L'ho fatto adottando il punto di vista del singolo *oggetto* e delle regole del suo corretto funzionamento, senza considerare i principi generali ai quali dovrebbe ispirarsi la sua progettazione. Qui prendo in considerazione anche questo aspetto, introducendo il *Single Responsability Principle*.

Il SRP fornisce una precisa formulazione di un aspetto fondamentale della programmazione: ogni *unità di codice* (metodo, *modulo*, *oggetto*) dovrebbe implementare una singola funzione[5]:

> **ogni modulo deve avere una sola *responsabilità* e questa deve essere interamente incapsulata dentro di esso. Tutti i servizi offerti dovrebbero essere strettamente allineati a tale responsabilità.**

Nel SRP il termine responsabilità viene definito come *un motivo per cambiare*. Dunque, uno *oggetto* dovrebbe essere progettato in modo da avere un solo *motivo per cambiare*. Si tratta di una definizione criptica, che cercherò di spiegare utilizzando come esempio una nuova versione di `StringList`.

10.1 *Aggiungere una funzione a StringList: persistenza dei dati*

Supponi di aver implementato la classe `StringList` come risposta ai requisiti di un determinato problema[6]. Supponi anche, come requisito aggiuntivo, di dover implementare la persistenza dei dati in un file, contenente un elenco di nominativi da gestire.

Poiché è necessario caricare i nomi in uno `StringList`, appare intuitivo implementare questa funzione nella classe, con un metodo per il caricamento e uno per il salvataggio:

```
class StringList
{
    string[] items;
    int count;
    ...
    public void LoadFromFile(string fileName)
    {
        items = File.ReadAllLines(fileName);
        count = items.Length;
    }

    public void SaveToFile(string fileName)
    {
        File.WriteAllLines(fileName, items);
    }
}
```

5 Il termine modulo identifica qui un'*unità di codice* in generale.

6 Naturalmente il ragionamento, come l'intero tutorial, si basa sul presupposto che non esistano oggetti come **List<>**, **Dictionary<>**, etc.

La nuova classe semplifica il codice applicativo che richiede caricamento e salvataggio dei dati, come mostra il seguente codice:

```
StringList list = new StringList();
list.LoadFromFile("Nominativi.txt"); //-> la lista conterrà i nominativi del file
...
list.Add("Borghi, Andrea");
list.Add("Ferri, Antonio");
...
list.SaveToFile("Nominativi.txt"); //-> il file conterrà anche i nuovi nominativi
```

Ma diminuisce la coesione della classe e introduce dei problemi legati alla sua eventuale evoluzione.

10.2 Bassa coesione: problemi legati all'evoluzione di StringList

Per *coesione* si intende la *caratteristica delle parti di un sistema di integrarsi e collaborare per il raggiungimento dello stesso fine*. Un sistema poco coeso contiene parti che, benché funzionanti, perseguono un fine distinto da quello generale; ciò può produrre dei problemi nel momento in cui occorre effettuare dei cambiamenti al sistema.

Il software si evolve, per soddisfare nuovi requisiti, per aumentare le prestazioni o semplicemente per migliorarne la qualità a parità di funzionamento (*refactoring*). Ma modificare il software non è un'operazione indolore; richiede tempo e, soprattutto, presenta il rischio di introdurre dei bug in un sistema prima funzionante.

In sostanza, il software dovrebbero esibire una certa "inerzia" verso i cambiamenti. Per favorire questa caratteristica, i vari componenti (metodi, *moduli*, *oggetti*) dovrebbero:

- Essere implementati correttamente. (Perché si dovrebbe cambiare del codice ben scritto, performante e che soddisfa tutti i requisiti sulla base dei quali è stato realizzato?)
- *Implementare una sola funzione*.

Considera l'interfaccia pubblica di `StringList`:

```
class StringList
{
    ...
    public StringList() { ... }
    public int Count() { ... }
    public void Add(string item) { ... }
    public void RemoveAt(int index) { ... }
    public string GetItem(int index) { ... }
    public void SetItem(int index, string item) { ... }

    public void LoadFromFile(string fileName)  { ... }
    public void SaveToFile(string fileName)  { ... }
}
```

La classe implementa due funzioni – gestire una collezione di elementi; persisterla su file – che possono evolvere separatamente.

Per quanto riguarda la prima funzione, sarebbe utile aggiungere i metodi `Clear()` (svuota la lista) e `Remove()` (rimuove l'elemento specificato). Per quanto riguarda la persistenza su file, si potrebbe decidere di "saltare" eventuali righe vuote, oppure che iniziano con un determinato carattere, etc.

Sono cambiamenti "ortogonali" tra loro: modificare il processo di caricamento e salvataggio dei dati non riguarda la funzione di gestione in memoria degli elementi, e viceversa. In sostanza: *esistono due motivi distinti che potrebbero portare a modificare la classe*. Dunque, `StringList` ha due responsabilità e quindi viola il SRP.

10.3 *Riutilizzare StringList*

`StringList` è una classe d'uso generale ed è utile in molti scenari. Immagina di dover scrivere un programma che richieda il caricamento da un file usato per configurare l'interfaccia utente mediante l'impostazione di determinate proprietà:

```
#File di configurazione
#colors
forecolor = yellow
backcolor = blue

#layout
width = 50
height = 30
...
```

Il file contiene delle definizioni da caricare (in grassetto), miste a commenti (prefissati da `#`) da scartare; ma il metodo `LoadFromFile()` carica tutte le righe del file, dunque non è utilizzabile. Si presentano quindi due scelte: modificare il metodo `LoadFromFile()`, oppure non usarlo affatto e caricare i dati nella lista mediante codice esterno.

La prima soluzione non è praticabile, perché l'uso di `StringList` in scenari diversi condurrebbe a una proliferazione di versioni distinte della classe. Resta la seconda soluzione, di certo praticabile, ma che evidenzia il problema progettuale di `StringList`: parte della sua interfaccia pubblica non è utilizzabile in determinati scenari.

10.4 Conclusioni

Il *Single Responsability Principle* non riguarda il funzionamento del software. Dei componenti potrebbero rispettarlo, ma avere dei bug; viceversa, potrebbero funzionare perfettamente e allo stesso tempo violare il principio. Il SRP stabilisce che un componente deve essere "responsabile" di una sola funzione. Ciò porta i seguenti vantaggi:

- Riduce la complessità del componente. (Riduce anche il suo ***ingombro*** e cioè il numero di altri componenti che deve usare per assolvere alla propria funzione.)
- Gli conferisce una maggiore "inerzia" ai cambiamenti, e dunque riduce la probabilità che successive modifiche introducano dei bug.
- Lo rende più facilmente testabile, perché è soltanto una la funzione della quale occorre verificare l'implementazione.
- Lo rende più facilmente riutilizzabile.

11 Frequently Asked Questions

1. Come si implementa un *oggetto*?

Un *oggetto* viene implementato mediante il costrutto `class`. Nella terminologia comune, infatti, si usa il termine *classe*. La parola "oggetto" viene di solito impiegata per indicare l'*istanza* di una classe. (2)

2. In sostanza, cosa distingue i *record* dagli *oggetti*?

I *record* sono costrutti completamente orientati ai dati, necessari per rappresentare concetti per i quali i tipi primitivi non sono sufficienti, oppure semplificare l'elaborazione di dati correlati tra loro. Gli *oggetti* sono costrutti utilizzati per creare nuove funzionalità.

Il codice che elabora i dati di un *record* non è definito nel *record*, ma altrove Un *oggetto* definisce sia i dati che i metodi necessari per implementare la funzione richiesta. (7)

3. Che cosa unisce *record* e *oggetti*?

Sono entrambi dei costrutti `class` e dunque, nel codice, sono utilizzati nello stesso modo. In sintesi: rappresentano entrambi dei *tipi*. (2.2)(7)

4. In che cosa, il linguaggio C#, fa distinzione tra *record* e *oggetti*?

Non fa alcuna distinzione. (2.2)(7)

5. In sostanza, cosa distingue i *moduli* dagli *oggetti*?

Diversamente dagli *oggetti*, i *moduli* non sono dei tipi, ma esclusivamente dei contenitori di codice. Non è possibile (né avrebbe alcun senso) usarli per dichiarare variabili. (1.3.1)

6. Che cosa unisce *moduli* e *oggetti*?

Entrambi possono essere usati per implementare nuove funzionalità. A entrambi si applica il *principio di incapsulamento*. (1.3.1)

7. In che cosa, il linguaggio C#, fa distinzione tra *moduli* e *oggetti*?

Non esiste un costrutto specifico per disegnare i *moduli* (mentre esiste in altri linguaggi), ma è possibile, anteponendo la parola `static` a `class`, definire una *classe statica*, è cioè un costrutto che ammette solo membri statici e non consente di dichiarare variabili e creare istanze.

Ciò detto, sono la funzione e l'uso a qualificare un'unità di codice come un *oggetto* o un *modulo*. Ad esempio, `Program` non è una classe statica, ma resta un *modulo* a tutti gli effetti. (1.3.1)

8. Cosa si intende per ciclo di vita di un oggetto (un'istanza)

Il ciclo di vita si riferisce alle istanze di *oggetti* e *record* e rappresenta la loro "durata", o esistenza, cioè la distanza tra la loro creazione e la loro distruzione.

Ad esempio, l'istanza assegnata a una variabile statica ha lo stesso ciclo di vita del programma. Il ciclo di vita di un'istanza assegnata a una variabile locale coincide con l'esecuzione del metodo nel quale la variabile è utilizzata (presupponendo che sia l'unica variabile a referenziare l'istanza).

9. Quando un'istanza viene "distrutta"?

Quando non è più referenziata da nessuna variabile.

10. Cosa si intende per *stato di un oggetto*?

Il termine *stato* indica i valori memorizzati nei campi di un'istanza (l'oggetto) in un dato momento. (2.1.3)

11. Cosa rappresenta l'*invariante di classe*?

Il termine *invariante di classe* designa l'insieme di vincoli (*precondizioni* e *postcondizioni*), applicati nei metodi della classe e necessari al suo corretto funzionamento. Lo scopo è garantire che le istanze della classe non possano mai trovarsi in uno *stato* non valido e che i metodi restituiscano sempre dei risultati corretti. (4)

12. Cos'è una *precondizione*?

È un vincolo che deve essere soddisfatto perché un metodo possa continuare la propria esecuzione. Deve essere verificata all'inizio del metodo; se la verifica è negativa, occorre sollevare un'eccezione. (4.2)

13. Le *precondizioni* si applicano necessariamente a tutti i metodi?

NO, è importante applicarle soltanto nei metodi pubblici. L'idea è la seguente: un metodo privato non ha bisogno di verificare lo *stato* dell'oggetto o gli argomenti ricevuti, poiché può presupporre che siano validi, in quanto già verificati dai metodi pubblici. (4.2)

14. Cosa sono le *postcondizioni*?

Le *postcondizioni* sono dei vincoli sullo *stato* dell'oggetto (e sui risultati, se ne producono) che i metodi pubblici devono soddisfare *al termine della propria esecuzione*. Di norma, le *postcondizioni* non sono verificate esplicitamente nei metodi, poiché si assume che siano soddisfatte se il codice è scritto correttamente[7]. (4.3)

7 Alcuni linguaggi (non C#) forniscono dei costrutti specifici per verificare le *pre* e *postcondizioni*. .NET Framework fornisce una funzionalità di simile, chiamata *code-contracts*.

15. Cos'è il *principio di incapsulamento*?

Il principio afferma *che il codice che usa una funzione non dovrebbe dipendere da come questa è implementata*. In questo modo, sarà possibile modificare l'implementazione della funzione senza dover modificare anche il codice che la usa.

16. Il *principio di incapsulamento* riguarda soltanto la programmazione *object oriented*?

NO, riguarda qualunque unità di codice: metodo, *modulo*, *oggetto*.

17. Il *principio di incapsulamento* è importante nella programmazione *object oriented*?

È fondamentale. Occorre immaginare gli *oggetti* come dispositivi il cui uso *non deve dipendere in alcun modo dalla loro realizzazione interna*. In questo modo sarà possibile modificare la realizzazione senza dover cambiare anche il modo in cui il dispositivo viene usato.

18. Cos'è l'*interfaccia pubblica* di una *oggetto*?

È l'insieme dei membri pubblici e stabilisce la funzione svolta dall'*oggetto*. (2.2.1)

19. Un campo può far parte dell'interfaccia pubblica di un *oggetto*?

NO, ma il linguaggio non lo impedisce. Definire dei campi pubblici rende impossibile garantire l'*invariante di classe* e viola il *principio di incapsulamento*, poiché fornisce al codice esterno l'accesso all'implementazione dell'*oggetto*.

(Gli *oggetti immutabili* rappresentano un'eccezione.) (8)

20. È sbagliato definire pubblico un metodo che serve al funzionamento dell'oggetto, ma non deve essere usato dal codice esterno?

SI, è sbagliato, poiché viola il *principio di incapsulamento*! (3.4)

21. Cosa significa il termine "*breaking change*"

Il termine si applica quando il cambiamento di una classe (ma anche di un *modulo* o di un metodo) causa l'invalidità o il malfunzionamento del codice che la usa, il quale deve quindi essere modificato a sua volta per conformarsi alla nuova situazione.

22. Il termine "*breaking change*" vale solo per le modifiche ai membri pubblici?

IN GENERALE SI; in pratica anche le modifiche all'implementazione possono produrre un *breaking change*, se alterano il modo in cui la classe funziona.

Ad ogni modo: una modifica all'interfaccia pubblica è sicuramente un *breaking change*.

23. Cos'è un costruttore?

È un metodo speciale che viene eseguito automaticamente nella creazione delle istanze della classe. La sua funzione è impostare uno *stato* iniziale valido per le istanze. (3.5)

24. Una classe può definire più di un costruttore?

SI, esattamente come può definire più metodi con lo stesso nome, purché abbiano una *firma* diversa. Lo stesso vale per i costruttori. (3.5.3)

25. Cos'è il *costruttore predefinito*? E il costruttore vuoto?

Il termine *costruttore predefinito* designa un costruttore senza parametri. Il *costruttore vuoto* è un costruttore senza parametri e privo di istruzioni. (3.5.1)

26. Una classe è obbligata a definire almeno un costruttore?

NO. Ogni classe deve avere almeno un costruttore, ma in caso non ne venga definito nessuno, il linguaggio definirà automaticamente un *costruttore vuoto*. (3.5.1)

27. Se una classe non ha il costruttore predefinito, lo fornisce il linguaggio?

SOLO, se non definisce alcun costruttore. (3.5.1) (3.5.2)

28. È possibile eseguire il costruttore all'interno di un metodo?

NO. Un costruttore non può essere eseguito esplicitamente; viene chiamato automaticamente nella fase di creazione delle istanze.

Esiste comunque un'eccezione: è possibile per un costruttore chiamarne un altro attraverso l'alias `this`; l'alias deve essere specificato nell'intestazione, dopo la lista dei parametri. (3.5.4)

29. Un *oggetto* può definire metodi statici?

SI, e sono usati nello stesso metodo dei metodi dei *moduli*. (In questo senso, C# non fa differenza tra *moduli* e *oggetti*.)

30. Un metodo statico può utilizzare un membro d'istanza?

NO, poiché nell'esecuzione di un metodo statico non è coinvolta nessuna istanza.

31. Quando un *oggetto* si dice immutabile?

Quando lo *stato* delle istanze non può essere modificato dopo la creazione. In molti scenari, gli oggetti immutabili sono perché rendono sicura la condivisione della stessa istanza tra parti diverse del programma, con la garanzia che non vi siano delle modifiche indesiderate. (8)

32. Cos’è il Single Responsability Principle?

In sostanza, afferma che un *oggetto* (come un metodo e un *modulo*) dovrebbe implementare una sola funzione. (10)