

OPP: oggetti II

Progettare oggetti

Anno 2018/2019

Indice generale

1	Introduzione.....	4
1.1	Rappresentare dati e implementare funzioni: <i>record</i> e <i>moduli</i>	4
2	Introduzione gli <i>oggetti</i> (classi).....	6
2.1	Terminologia: <i>oggetti</i> (classi) e <i>istanze</i> di <i>oggetti</i>	6
2.2	Sintassi.....	6
2.3	Progettare oggetti.....	7
3	<i>Oggetti</i> che incapsulano strutture dati.....	8
3.1	Implementare una lista: classe <i>StringList</i>	8
3.2	Interfaccia pubblica della classe <i>StringList</i>	8
3.3	Implementazione (privata).....	9
3.3.1	Refactoring: l'importanza di un'implementazione privata.....	10
3.4	Costruttori.....	10
3.4.1	Costruttore predefinito.....	10
3.4.2	Costruttore con parametri.....	11
3.4.3	Definire più costruttori.....	11
3.4.4	Chiamare un costruttore da un altro: parola chiave <i>this</i>	12
3.4.5	Definire dei parametri con valori predefiniti.....	12
4	Invariante di classe.....	13
4.1	Postcondizioni.....	13
4.1.1	Non soddisfare le postcondizioni.....	13
4.2	<i>Precondizioni</i>	14
4.3	Applicare l'invariante di classe a <i>StringList</i>	14
4.3.1	Creazione della lista: verifica della capacità iniziale.....	14
4.3.2	Accesso all'elemento: <i>GetItem()</i> e <i>SetItem()</i>	15
4.3.3	Eliminazione di un elemento: <i>RemoveAt()</i>	16
4.4	Campi pubblici e <i>invariante di classe</i>	16
4.5	Conclusioni.....	16
5	Proprietà.....	17
5.1	Proprietà semplici.....	17
5.2	Definizione di una proprietà.....	18
5.3	Accesso in sola lettura a un campo: proprietà <i>get-only</i>	19

5.3.1	Proprietà con “backing field”	19
5.4	Restituzione di un valore derivato: proprietà <i>get-only</i>	19
5.5	Proprietà “scrivibile”: implementazione di <i>get</i> e <i>set accessor</i>	19
5.6	Proprietà automatiche.....	20
5.6.1	Definizione di Count come proprietà automatica.....	21
6	Proprietà indicizzate.....	22
6.1	Definizione di un indicizzatore.....	22
6.2	Implementazione di un indicizzatore in <i>StringList</i>	23
6.3	Conclusioni.....	23

1 Introduzione

Questo tutorial affronta il tema centrale della programmazione *object oriented*: la progettazione di *oggetti* (classi). Introdurrò le funzioni svolte dagli *oggetti* e i principi che ispirano la loro progettazione. Contestualmente, approfondirò la sintassi e le funzionalità fornite dal linguaggio C# per la loro implementazione.

Partendo dal modello di programmazione procedurale e dalla funzione di *record* e i *moduli* di rappresentare i dati e incapsulare i procedimenti, mostrerò come gli *oggetti* estendano questo modello e semplifichino la rappresentazione dei dati e la separazione delle funzioni.

Il tutorial si collega ad altri tre:

- **Record**, nel quale introduco i *record* e il loro ruolo fondamentale nella rappresentazione dei dati.
- **Moduli:**
- **OOP: oggetti I**, nel quale esamino il modo in cui sono implementati gli *oggetti*, senza però approfondire le regole del linguaggio e, soprattutto, la funzione che svolgono nella programmazione.

1.1 Rappresentare dati e implementare funzioni: *record* e *moduli*

Per poter comprendere la funzione svolta dagli *oggetti*, occorre partire da due aspetti chiave della programmazione. Il primo riguarda la possibilità di rappresentare concetti che vadano oltre il singolo valore, sia esso stringa, numerico, etc. Questa è il ruolo dei *record*. Il secondo riguarda la capacità di incapsulare funzioni che vadano oltre il singolo procedimento, o metodo. Questa possibilità è fornita dai *moduli*.

I due ruoli non si sovrappongono. Non è possibile implementare funzioni mediante un *record*, poiché non definisce dei metodi. Non è possibile rappresentare dati mediante un *modulo*, nonostante, per svolgere la propria funzione, possa usare delle variabili.

Nella programmazione, però, esistono concetti che abbracciano entrambi gli aspetti. Considera ad esempio il concetto di *lista*:

una lista è una collezione dinamica e indicizzabile di elementi.

Il concetto di *lista* richiama la necessità di memorizzare gli elementi (rappresentare i dati) e di eseguire delle operazioni, come inserimento, rimozione, etc (procedimenti).

L'implementazione è piuttosto semplice; è sufficiente un *record* che definisca i dati necessari a gestire lo stato della lista (supponi una lista di stringhe):

```
class ListaStringhe
{
    public string[] Items; // memorizza gli elementi
    public int Count;      // memorizza il numero di elementi
}
```

e dei metodi per eseguire le operazioni (per semplicità, supponi una lista di capacità massima di 100 elementi):

```

class Program
{
    static ListaStringhe lista;
    static void Main(string[] args)
    {
        ...
    }

    static void Create()    // crea la lista (inizializza campi)
    {
        lista = new ListaStringhe
        {
            items = new string[100],
            count = 0
        };
    }

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

    static string GetItem(int index) { ... }    // restituisce un elemento dato l'indice
    static void SetItem(int index, string item) { ... }    // setta un elemento dato
                                                         // l'indice
    static void RemoveAt(int index) { ... }    // rimuove un elemento all'indice specificato
}

```

Il codice prodotto, però, permette di gestire una lista dinamica, ma non implementa il concetto di *lista*. Non consente, cioè, di creare più liste, o di passare una lista a un metodo, etc. In sostanza, il concetto di *lista*, come molti altri concetti, va oltre i dati e i metodi necessari a implementarla.

Nel tutorial **OOP: oggetti I** ho spiegato che con un'accurata progettazione di *record* e *moduli* è possibile implementare concetti come *lista*, ma è una soluzione che presenta dei difetti. Il modello di programmazione *object oriented* li risolve in modo elegante ed efficace.

2 Introduzione gli oggetti (classi)

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

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

Dunque, gli *oggetti* uniscono le caratteristiche di *record* (stato → dati) e *moduli*, (comportamento → metodi).

2.1 Terminologia: *oggetti* (classi) e *istanze di oggetti*

Con il termine *oggetto* (in corsivo) intendo un tipo di dato, come `string`, `int`, `double`, etc. Una variabile *oggetto* viene chiamata *istanza* dell'*oggetto*. Dunque, nel seguente codice:

```
StringList elencoNomi = new StringList();  
...
```

`elencoNomi` è un'*istanza* del tipo `StringList`.¹

Nella terminologia, comunque, si usa il termine **classe** e non *oggetto*; si usa il termine "oggetto" per identificare un'*istanza* di *classe*. (Dunque: `StringList` è una *classe*, mentre `elencoNomi` è un oggetto, o istanza di *classe*.)

L'importante è comprendere i concetti e non confonderli. Nel testo uso il corsivo per dare a *oggetto* il significato di tipo, e dunque di *classe*, mentre uso il carattere normale per dare il significato di *istanza*.

2.2 Sintassi

La definizione di un *oggetto* segue la sintassi (qui semplificata):

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

Ricapitolando:

- Un *oggetto* è definito mediante la parola chiave `class`, che introduce un nuovo tipo, rappresentato dal nome dell'*oggetto*.
- Un *oggetto* definisce zero o più campi e zero o più metodi. Campi e metodi vengono definiti **membri** dell'*oggetto*. (L'ordine delle definizioni non ha importanza.)

¹ In realtà la situazione non è esattamente questa, ma restano due affermazioni accettabili.

- 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 nel codice esterno all'*oggetto*.

Dalle precedenti regole si trae un'implicazione importante: 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 progettare i tipi di dati in modo coerente, sulla base dei requisiti da soddisfare.

Progettare un *record* non richiede particolari attenzioni, se non la scelta accurata dei campi e del loro tipo. Implementare un *oggetto* è un compito più difficile, poiché deve soddisfare dei vincoli e rispettare determinati principi di progettazione: *incapsulamento*, *invarianza*, *coesione*.

2.3 Progettare oggetti

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 o a grandezza naturale; dal disegno degli interni; dal definire le sue caratteristiche: motorizzazione, potenza, velocità, consumi, etc. In sostanza, si progetta il veicolo dal punto di vista di chi lo dovrà utilizzare: il cliente.

Nel progettare un *oggetto* si dovrebbe adottare un processo analogo: stabilire innanzitutto il concetto che l'*oggetto* deve rappresentare e i requisiti, dal punto di vista dell'utilizzatore, che deve soddisfare. Dopodiché si può pensare al modo in cui l'*oggetto* debba essere implementato per soddisfare i requisiti suddetti.

Questo approccio è utile anche a conseguire un risultato fondamentale: tenere separato ciò che l'*oggetto* "fa", dal "come" è in grado di farlo. In sostanza: separare 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.

3 Oggetti che incapsulano strutture dati

Le strutture dati – *lista*, *pila*, *coda*, *dizionario*, etc – sono esempi ideali 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 struttura dati nella quale gli elementi sono accessibili mediante la loro posizione, o indice. 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.
- Numero di elementi memorizzati.
- Inserimento (in coda) 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.

Ogni operazione deve essere implementata mediante un metodo pubblico.

3.2 Interfaccia pubblica della classe `StringList`

L'insieme dei membri pubblici, accessibili al codice esterno, rappresenta l'**interfaccia pubblica** della classe. Di fatto, è questa che definisce la funzione dell'oggetto; dunque: *qualunque membro dell'oggetto che non è 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 l'importanza il modo in cui questi metodi sono implementati, né come e dove siano memorizzati gli elementi.

3.3 Implementazione (privata)

I membri privati e il codice dei metodi contribuiscono all'**implementazione privata** della classe. Segue l'implementazione completa della classe (i membri privati sono evidenziati):

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

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

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

    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 logica 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.

3.3.1 Refactoring: l'importanza di un'implementazione privata

In `StringList` il metodo `Add()` garantisce che il vettore contenente gli elementi abbia la capacità necessaria a memorizzare il nuovo elemento, eventualmente aumentando la capacità del vettore:

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

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

```
public void Add(string item)
{
    if (count == items.Length)
        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 definito privato*! Se fosse pubblico, eliminarlo significherebbe produrre un **braking change**, cioè una modifica all'interfaccia pubblica, potenzialmente in grado di invalidare il codice che usa la classe.

3.4 Costruttori

I costruttori sono metodi speciali che vengono eseguiti nella creazione degli oggetti. Il loro scopo è stabilire lo stato iniziale dell'oggetto; in linea di massima non dovrebbero eseguire altri compiti.

3.4.1 Costruttore predefinito

Un costruttore che non accetta parametri è detto **costruttore predefinito**. Una classe priva di costruttori "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
{
    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.

3.4.2 Costruttore con parametri

Se per un *oggetto* non esiste uno stato predefinito valido, oppure si decide che debba essere il codice esterno a impostare lo stato iniziale, occorre definire un costruttore con parametri:

```
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:

```
StringList elenco1 = new StringList(50) // OK!
StringList elenco2 = new StringList() // ERRORE: non esiste il costruttore StringList()!
```

3.4.3 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 della lista.

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

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

```

    }

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

```

3.4.4 Chiamare un costruttore da un altro: parola chiave *this*.

All'interno della classe un costruttore non può essere invocato esplicitamente, con un'eccezione: un costruttore può chiamarne implicitamente un altro mediante la parola chiave `this`. Ciò è utile per evitare duplicazioni di codice.

In `StringList` i due costruttori condividono praticamente lo stesso codice; è possibile rimediare facendo sì che il *costruttore predefinito* chiami l'altro, specificando 10 come capacità iniziale:

```

class StringList
{
    string[] items;
    int count;
    public StringList():this(10) { } // invoca 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*.

3.4.5 Definire dei parametri con valori predefiniti

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

```

class StringList
{
    string[] items;
    int count;

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

```

Di fronte a questo codice, il linguaggio definisce automaticamente due costruttori identici a quelli forniti nell'esempio precedente, facendo sì che il costruttore predefinito chiami l'altro.

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 dall'insieme dei metodi.

L'**invariante di classe** stabilisce che l'oggetto, a partire dalla sua creazione, deve sempre trovarsi in uno stato valido. L'invariante deve essere stabilita in fase di progettazione e pone dei vincoli da rispettare sul funzionamento dei metodi pubblici: **precondizioni** e **postcondizioni**

In pratica, *precondizioni* e *postcondizioni* rappresentano un "contratto" fra il codice esterno e la classe. Il codice esterno è tenuto a fornire argomenti validi, il codice della classe a fornire risultati corretti sulla base di quegli argomenti.

4.1 Postcondizioni

Una *postcondizione* stabilisce i vincoli sullo *stato* che un metodo pubblico deve rispettare. Se un metodo restituisce un valore, la *postcondizione* stabilisce eventuali vincoli su di esso.

Ad esempio, le *postcondizioni* del metodo `Add()` sono:

- Il numero di valori della lista è aumentato di 1.
- Il nuovo valore diventa 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.

4.1.1 Non soddisfare le postcondizioni

Considera la seguente versione di `Add()`, nella quale ho volutamente introdotto un bug:

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

L'esecuzione del metodo non produce alcun errore, ma le *postcondizioni* non sono soddisfatte: ad ogni inserimento, l'elemento viene memorizzato nella prima posizione. Il risultato è che, pur senza produrre alcun errore, la lista memorizzerà un solo elemento, l'ultimo inserito.

4.2 Precondizioni

Le *precondizioni* sono vincoli diretti al codice esterno: ha la responsabilità di fornire degli argomenti validi ai metodi eseguiti. Questi ultimi, d'altra parte, devono verificare che le *precondizioni* siano soddisfatte, in caso contrario devono sollevare un'eccezione.

Appare evidente che *precondizioni* e *postcondizioni* sono collegate. Ogni metodo deve produrre risultati corretti (*postcondizioni*), ma non può farlo se riceve dati non validi (*precondizioni*). Questo, naturalmente, riguarda anche i costruttori.

4.3 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.

4.3.1 Creazione della lista: verifica della capacità iniziale

Il costruttore accetta come parametro la capacità iniziale della lista. Dato che un valore negativo è inaccettabile, occorre stabilire la giusta precondizione da soddisfare: `capacity > 0` o `capacity >= 0`. La prima garantisce sicuramente l'invariante di classe; apparentemente anche la seconda, ma in realtà 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 inconsistenti tra loro. La soluzione consiste nell'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 preconditione nel costruttore: `capacity >= 0`

Se `capacity` non rispetta la preconditione (`>=0`), il costruttore deve sollevare un'eccezione del tipo appropriato:

```
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.

4.3.2 Accesso all'elemento: `GetItem()` e `SetItem()`

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

```
StringList nomi = new StringList(); //-> capacità: 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 riferenzia un elemento esistente, ma non inizializzato, del vettore; nel secondo caso viene prodotta un'eccezione, perché l'elemento di indice 10 non esiste affatto. In realtà entrambi gli elementi richiesti non esistono nella lista!

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));
}
```

4.3.3 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 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--;
}
```

4.4 Campi pubblici e *invariante di classe*

Un *oggetto* non deve dichiarare pubblici i campi, perché in questo modo rende accessibile la propria implementazione e dunque viola il principio di incapsulamento. Oltre a questo, appare evidente che i campi pubblici rendono impossibile garantire l'*invariante di classe*; infatti, il codice esterno può modificare arbitrariamente il loro valore senza che la classe possa imporre alcun vincolo al riguardo.

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

4.5 Conclusioni

Una buona metafora per comprendere il concetto di *invariante di classe* è quella del *dispositivo*. Un dispositivo (istanza) funziona sulla base di un progetto (classe); questo deve prevedere in quali *stati* il dispositivo può trovarsi e quali *stati*, invece, devono essere evitati perché non validi.

L'*invariante di classe*, appunto, fa riferimento alla necessità di progettare le classi in modo che gli oggetti (le istanze) si trovino in stati validi e producano risultati corretti. Poiché nel progettare una classe non si ha il controllo su come sarà usata, è necessario che i metodi pubblici verifichino che siano soddisfatte le condizioni per eseguire correttamente il procedimento richiesto. Se queste *precondizioni* non sono soddisfatte, il metodo deve produrre un'eccezione.

Le *postcondizioni* sono, di fatto, incorporate nei procedimenti da eseguire 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.²

2 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 (ma 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 dell'operazione eseguita.

Normalmente, i metodi incapsulano un procedimento, che può produrre un risultato. In alcuni casi, però, il metodo rappresenta 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()` esprimono l'idea di un procedimento eseguito sulla lista; il metodo `Count()`, d'altra parte, veicola l'idea di una semplice informazione che viene restituita. In questo caso, quindi, esprime un intento normalmente associato alle variabili: accedere e/o modificare un dato.

La *proprietà* esistono 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 d'uso delle variabili.

Considera il metodo `Count()`, la cui funzione è restituire il numero di elementi della lista; restituire cioè il valore del campo `count`. Una 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 Definizione di una proprietà

Segue la sintassi (semplificata):

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

opzionali

Nota bene:

- Diversamente dai metodi, una proprietà non definisce dei parametri.
- Può definire uno o due *accessor*, i quali corrispondono a dei metodi.
- 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 sono un costrutto flessibile che rappresenta 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`, che fanno da “wrapper” a un campo, sono definite anche proprietà con *backing field*; (letteralmente: “campo che sta dietro”), dove tale termine indica, appunto, il campo contenente l'informazione resa accessibile dalla proprietà.

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

In alcuni casi il valore da restituire non è memorizzato in un campo, ma viene prodotto attraverso un semplice 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 della proprietà viene derivato, non ottenuto direttamente da un campo.

5.5 Proprietà “scrivibile”: implementazione di *get* e *set accessor*

Nella classe `List<>` la proprietà `Capacity` è modificabile, e dunque definisce anche il *set accessor*. Prima di vedere come implementarla in `StringList`, considera l'uso della proprietà:

```
List<string> nomi = new List<string>(10);
Console.WriteLine(nomi.Capacity); // -> 10
...
nomi.Capacity = 20;
Console.WriteLine(nomi.Capacity); // -> 20 (items.Length è 20)
```

È come se il codice esterno utilizzasse una variabile, `Capacity`. In realtà la proprietà fornisce un meccanismo per restituire e modificare la dimensione del vettore contenente gli elementi. Il *set accessor* contiene il codice necessario ad aumentare la dimensione del vettore, preceduto dalla verifica della *precondizione*. ***

```
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” 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 Definizione di Count come proprietà automatica

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

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

Ovviamente, nei metodi della classe qualsiasi riferimento al campo `count` deve essere modificato per utilizzare `Count`.

Il funzionamento della proprietà automatica `Count` diventa più chiaro analizzando la traduzione operata dal linguaggio:

Proprietà

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

Traduzione linguaggio

```
int __count;
public int Get_Count()
{
    return __count;
}

private void Set_Count(int value)
{
    __count = value;
}
```

6 Proprietà indicizzate

Mentre le proprietà semplici sono largamente impiegate nella programmazione, le *proprietà indicizzate*, o *indicizzatori*, trovano il loro naturale impiego in *oggetti* che rappresentano collezioni.

Un *indicizzatore* fornisce un meccanismo di accesso che impiega l'identica sintassi usata con i vettori. In modo analogo alle proprietà semplici è un costrutto che unisce le funzioni di due metodi.

6.1 Definizione di una 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*.
- 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_this(<parametri>)
{
  {...}    get accessor
}
```

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

6.2 Implementazione di un indicizzatore in StringList

Attualmente `StringList` è indicizzabile attraverso i metodi `GetItem()` e `SetItem()`. Il loro uso è funzionale, ma poco naturale, poiché ci si aspetta di poter usare un oggetto indicizzabile 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_this(int index)
{
    ValidateIndex(index);
    return items[index];
}

public void Set_this(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_this(0, "Andrea");
for (int i = 0; i < nomi.Count; i++)
{
    Console.WriteLine(nomi.Get_this(i));
}
```

6.3 Conclusioni

Quello mostrato è un esempio tipico di *proprietà indicizzata*, dove la chiave per accedere a un elemento è, appunto, un indice intero. Ma non esistono vincoli sul tipo e numero 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.