Sugli oggetti

Approfondimento sui "*reference types*"

Linguaggio C#

Anno 2017/2018

Indice generale

1 Introduzione

In questo tutorial approfondisco il modello di memorizzazione dei *reference types*, affrontando alcuni scenari e le relative implicazioni. Mi baserò su semplici esempi di codice, e schematizzerò il risultato prodotto in memoria allo scopo di chiarire alcuni concetti:

- Ciclo di vita degli oggetti (e record).
  - Differenza tra campi statici e non-statici (d'istanza)
- Differenza tra metodi statici e non-statici nell'accesso ai campi di classe.
- *Shared references*.

In alcuni degli esempi presentati utilizzerò le seguenti definizioni:

```
public class Persona
{
    public string Nome;
    public int Età;
}

public class ElencoPersone
{
    private List<Persona> elenco = new List<Persona>();
    public void Add(Persona p)
    {
        elenco.Add(p);
    }

    public List<Persona> GetPersone()
    {
        return elenco;
    }

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

    public Persona this[int index]
    {
        get { return elenco[index]; }
        set { elenco[index] = value; }
    }
}
```

La classe `ElencoPersone` gestisce un elenco di record di tipo `Persona`, fornendo le operazioni di inserimento e accesso indicizzato. La classe fornisce inoltre l'accesso alla lista interna (`elenco`) utilizzata per memorizzare i record (metodo `GetPersone()`).

2 Ciclo di vita di variabili e oggetti

Il termine *ciclo di vita* fa riferimento alla durata, o esistenza, di un oggetto; cioè:

- A partire da quando l'oggetto viene creato (viene allocata una zona di memoria che ne memorizza i dati).
- Fino a quando l'oggetto non è più utilizzabile (la zona di memoria utilizzata dall'oggetto è pronta per essere "riciclata" ed essere resa nuovamente disponibile).

Prima di procedere con degli esempi, occorre premettere che il *ciclo di vita*:

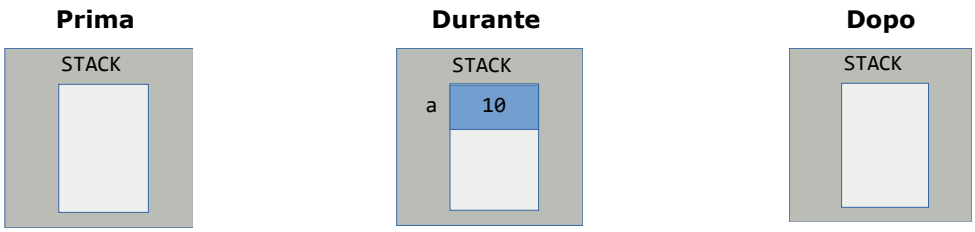
- È sempre riferito al programma in esecuzione (si parla di oggetti e non di classi).
- Non è influenzato dalla visibilità delle variabili (`public` / `private`).
- È influenzato dal luogo di dichiarazione delle variabili: locali ↔ globali.
- Dipende dal fatto che una variabile sia statica o no.
- Infine: implica una distinzione tra variabili e oggetti.

2.1 Ciclo di vita delle variabili locali

Il ciclo di vita di una variabile locale inizia con l'esecuzione del metodo nel quale è dichiarata e termina con la fine del metodo. Considera il seguente codice; in un metodo viene dichiarata e inizializzata una variabile intera:

```
class Program
{
    static void Main(string[] args)
    {
        Demo();
    }

    static void Demo()
    {
        // viene riservato spazio per "a" nella memoria stack
        int a = 10;
        Console.WriteLine(a);
    } // "a" non esiste più
}
```

Di seguito viene schematizzata la situazione in memoria, prima, durante e dopo l'esecuzione di `Demo()`:



Le variabili locali, di qualsiasi metodo, vengono allocate nella memoria stack, che viene (appunto) gestita come una pila.

È importante comprendere che `a` viene creata e distrutta ad ogni esecuzione del metodo. Per questo motivo non preserva il valore che aveva dopo la precedente esecuzione.

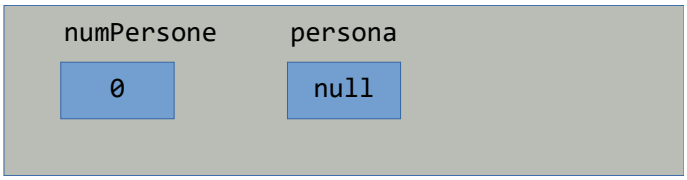
Naturalmente, una variabile locale dichiarata nel metodo `Main()` "vive" per tutta la durata del programma; resta comunque inaccessibile al codice esterno a `Main()`.

2.2 Ciclo di vita delle variabili statiche

Considera il seguente programma, nel quale sono dichiarate due variabili statiche:

```
class Program
{
    static Persona persona;
    static int numPersone;

    static void Main(string[] args)
    {
        Console.WriteLine(numPersone); // -> 0
        Console.ReadKey();
    }
}
```

L'esecuzione del programma produce in memoria la seguente situazione:



Immediatamente dopo l'avvio del programma viene allocato uno spazio di memoria sufficiente a memorizzare le due variabili. Entrambe vengono inizializzate a zero; nel caso di `persona` ciò produce un riferimento nullo.

In sostanza, il ciclo di vita di entrambe le variabili coincide con quello dell'intero programma, o, in altri termini, le variabili sono utilizzabili per tutta la durata del programma.

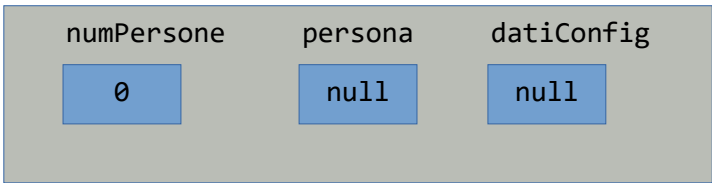
Nota bene: in memoria esiste una variabile di tipo `Persona`, ma non esiste alcun oggetto.

2.2.1 Variabili statiche dichiarate in moduli (o classi)

Il fatto che una variabile statica sia dichiarata altrove rispetto a `Program` non influenza il suo ciclo di vita, che corrisponde sempre a quello del programma. Considera il seguente modulo:

```
public static class Config
{
    public static Dictionary<string, string> datiConfig;
    ...
}
```

Dopo queste modifiche, l'esecuzione del programma produce la seguente situazione:



Niente distingue il ciclo di vita di `datiConfig` rispetto a quello delle due variabili in `Program`; semplicemente: il suo utilizzo in un modulo diverso da quello nel quale è dichiarata è regolato dai modificatori `public` / `private` e richiede il nome del modulo come prefisso.

2.3 Ciclo di vita degli oggetti

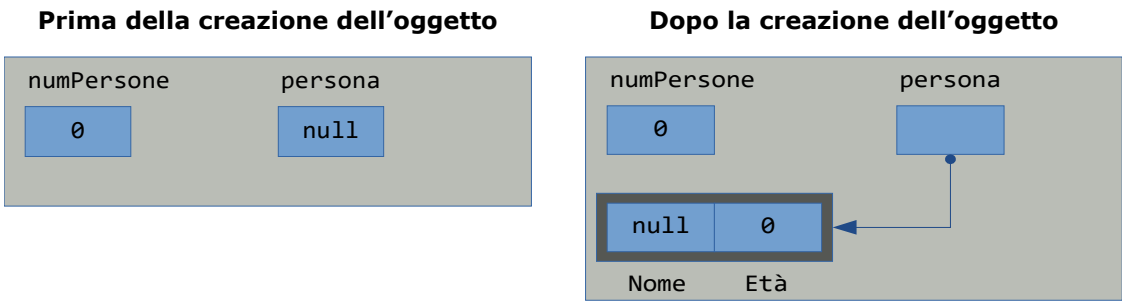
Il ciclo di vita di un oggetto comincia nel momento della sua creazione, che avviene con l'operatore `new`.

> ### Tipo string
>
> Il tipo `string` merita una menzione speciale, poiché si tratta sì di un *reference type*, ma consente di utilizzare una sintassi tipica dei *value type* (`int`, `double`, `bool`, etc) Infatti, si può scrivere:
>
> ```
> string s = "ciao!";
> ```
>
> Il linguaggio traduce l'istruzione precedente in:
>
> ```
> string s = new String(new char[] { 'c', 'i', 'a', 'o', '!' });
> ```

Di seguito ho aggiunto un'istruzione che crea un oggetto e lo assegna alla variabile `persona`:

```
class Program
{
    static Persona persona;
    static int numPersone;
    static void Main(string[] args)
    {
        persona = new Persona();
        Console.ReadKey();
    }
}
```

A pagina successiva viene mostrata la situazione in memoria, prima e dopo l'esecuzione dell'istruzione evidenziata:



2.3.1 Termine del ciclo di vita di un oggetto

In sintesi, un oggetto non è più utilizzabile (e dunque è al termine del suo ciclo di vita) *quando non esiste più nessuna variabile che lo referenzia*. Questa situazione può verificarsi in scenari diversi. Di seguito ne mostro alcuni.

Ciclo di vita di un oggetto referenziato da una variabile statica

Nel seguente codice, il metodo `CreaPersona()` crea un oggetto e lo assegna alla variabile statica `persona`.

```
class Program
{
    static Persona persona;
    static int numPersone;
    static void Main(string[] args)
    {
        //qui non esiste ancora alcun oggetto
        CreaPersona();
        //l'oggetto è stato creato
        Console.ReadKey();
    } //-> l'oggetto (come qualsiasi altra cosa) muore qui!

    static void CreaPersona()
    {
        persona = new Persona { Nome = "Pippo", Età = 34 };
    }
}
```

Dunque: l'oggetto "nasce" nell'istruzione creazione. Poiché è stato assegnato a una variabile statica, "muore" quando questa termina il proprio ciclo di vita.

Ma quello appena mostrato non è un destino immutabile:

```
static void Main(string[] args)
{
    //qui non esiste ancora alcun oggetto
    CreaPersona();
    //l'oggetto è stato creato
    persona = null;  //-> "persona" continua a vivere, ma l'oggetto muore qui!
    Console.ReadKey();
}
```

Assegnare `null` alla variabile fa sì che l'oggetto non sia più referenziato e dunque non sia più utilizzabile: l'oggetto muore prima che termini il ciclo di vita di `persona`.

Ciclo di vita di un oggetto referenziato da una variabile locale

L'oggetto muore al termine del metodo, poiché qui termina il ciclo di vita della variabile locale:

```
static void Main(string[] args)
{
    //qui non esiste ancora alcun oggetto
    CreaPersona();  //l'oggetto nasce e muore qui
    //qui non esiste più alcun oggetto
    Console.ReadKey();
}
static void CreaPersona()
{
    var p = new Persona { Nome = "Pippo", Età = 34 };
    Console.WriteLine(p.Nome);
} //termina il ciclo di vita di "p" e dunque anche dell'oggetto
```

Ciclo di vita di un oggetto referenziato da più variabili

Partiamo dall'esempio precedente, modificando il codice:

```
static void Main(string[] args)
{
    //qui non esiste ancora alcun oggetto
    var p = CreaPersona();  //l'oggetto nasce qui (ma non muore)
    Console.ReadKey();
}  //!l'oggetto muore qui

static Persona CreaPersona()
{
    var p = new Persona { Nome = "Pippo", Età = 34 };
    return p;
} //termina il ciclo di vita di "p", ma non dell'oggetto!
```

L'oggetto viene creato in `CreaPersona()` e il suo *reference* restituito. In `Main()` l'assegnazione a `p` "tiene" vivo l'oggetto fino a quando anche questa variabile termina il proprio ciclo di vita.

Durante l'esecuzione del programma, esistono, se pur in momenti diversi, due variabili che referenziano l'oggetto. Queste nascono e muoiono in momenti diversi; l'oggetto comincia il proprio ciclo di vita con la `p` di `CreaPersona()` e lo termina con la `p` di `Main()`.

2.3.2 Ciclo di vita dei campi dell'oggetto

Il ciclo di vita dei campi di un oggetto coincide con quello dell'oggetto stesso. Attenzione, qui sto parlando di variabili e non di eventuali oggetti referenziati dalle stesse. Nel secondo caso vale la regola applicata a qualsiasi oggetto.

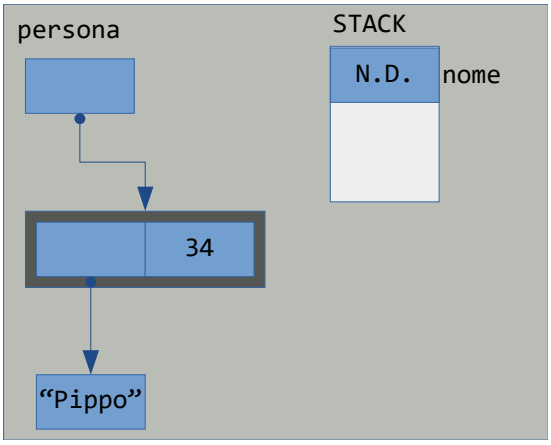
Nel seguente codice viene creato un oggetto `Persona`; dopodiché il campo `Nome` dell'oggetto viene assegnato a una variabile locale: dunque, due variabili referenziano lo stesso oggetto.

```
static Persona persona;
static int numPersone;
static void Main(string[] args)
{
    //qui non esiste ancora alcun oggetto
    persona = new Persona { Nome = "Pippo", Età = 34 }; //anche "Pippo" è un oggetto!

    string nome = persona.Nome; // nome e persona.Nome referenziano lo stesso oggetto
    persona = null;   // l'oggetto principale muore qui, ma "Pippo" continua a esistere
    Console.ReadKey();
}  //!l'oggetto "Pippo" muore qui
```

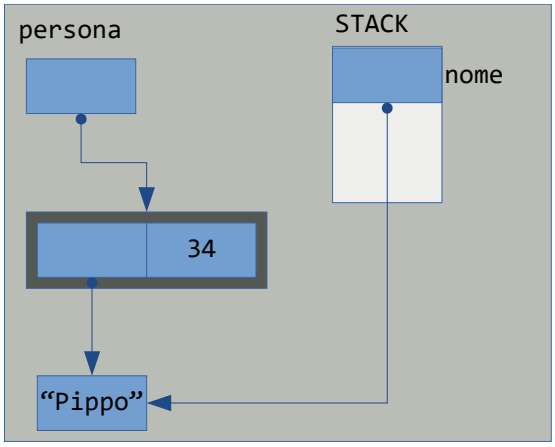
Accettata l'idea che anche le stringhe sono oggetti, il codice mostra che un oggetto referenziato dal campo di un altro oggetto può sopravvivere al proprio genitore. Infatti, la variabile locale `nome` referenzia lo stesso oggetto referenziato dal campo `Nome` (e cioè `"Pippo"`). Quando l'oggetto referenziato da `persona` muore (`persona = null`), anche il campo `Nome` muore con esso, ma non l'oggetto che referenziava, poiché questo è tenuto in vita dalla variabile locale `nome`.

Segue lo schema che mostra la situazione prima e dopo l'assegnazione alla variabile locale:

Prima dell'assegnazione



Dopo l'assegnazione



Anche in questo caso, come nell'esempio precedente, il ciclo di vita dell'oggetto non coincide con nessuna delle variabili che, in momenti diversi, lo referenziano.

3 Campi statici e di istanza

Innanzitutto una premessa: i campi statici possono essere definiti anche all'interno di classi e non soltanto di moduli (classi statiche). Ciò produce un paradosso: una classe definisce un campo (statico) che ha un ciclo di vita diverso dagli oggetti di quella classe.

In realtà il paradosso è soltanto apparente; infatti, l'appartenenza di un campo statico a una classe riguarda soltanto la sua visibilità, non il modo in cui viene gestito in memoria.

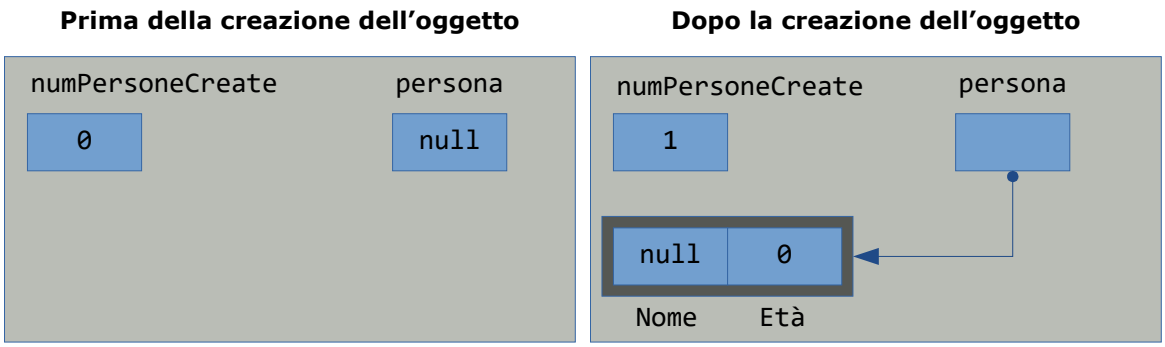
Considera la seguente modifica al tipo `Persona`:

```
public class Persona
{
    static int numPersoneCreate;

    public Persona()
    {
        numPersoneCreate++;
    }

    public string Nome;
    public int Età;
}
```

e il seguente codice in `Main()`:

```
static Persona persona;
static void Main(string[] args)
{
    // esiste già il campo "numPersoneCreate", ma non esiste ancora alcun oggetto.
    persona = new Persona(); // l'oggetto viene creato
    Console.ReadKey();
}
```

Ecco la situazione in memoria prima e dopo la creazione dell'oggetto `Persona`:



La gestione in memoria di `numPersoneCreate` è identica a quella di `persona`. L'appartenenza alla classe `Persona` influenza la sua accessibilità nel codice esterno, ma nient'altro.

3.1 Accesso ai campi statici all'interno della classe

All'interno della classe, l'uso dei campi statici segue le consuete regole: si utilizza il nome del campo. D'altra parte, è valido, anche se inutile, premettere il nome della classe:

```
public class Persona
{
    static int numPersoneCreate;
    public Persona()
    {
        Persona.numPersoneCreate++;  // Ok, anche se è inutile usare Persona.
    }

    public string Nome;
    public int Età;
}
```

Al contrario, non è corretto accedere al campo premettendo la parola `this`:

```
public class Persona
{
    static int numPersoneCreate;
    public Persona()
    {
        this.numPersoneCreate++;
    }

    public string Nome;
    public int Età;
}
```

Il perché è ovvio: `numPersoneCreate` *non è memorizzato dentro l'oggetto*. Accedere ad esso mediante la parola chiave che rappresenta un *reference* all'oggetto è semplicemente privo di senso.

4 Metodi statici e metodi di istanza

Una classe può definire sia metodi di istanza che metodi statici; tra i due tipi di metodi c'è una differenza fondamentale: *i metodi statici non possono accedere ai campi non-statici della classe*. Il perché risiede nel modo in cui il linguaggio gestisce gli oggetti e traduce i metodi di istanza.

Nel codice che segue, `Persona` definisce un metodo che restituisce l'anno di nascita. In `Main()` viene creato un oggetto ed eseguito il metodo suddetto:

```
public class Persona
{
    public string Nome;
    public int Età;

    public int AnnoNascita()
    {
        return DateTime.Now.Year - Età;
    }
}
...
static void Main(string[] args)
{
    var p = new Persona { Nome = "Filippo", Età = 25 };
    int anno = p.AnnoNascita();  // -> 1992
    Console.ReadKey();
}
```

Il linguaggio traduce il metodo `AnnoNascita()` in un metodo statico che accetta come primo parametro un oggetto di tipo `Persona`:

```
public class Persona
{
    public string Nome;
    public int Età;

    public int AnnoNascita()
    {
        return DateTime.Now.Year - Età;
    }

   // codice prodotto dal linguaggio
    public static int AnnoNascita(Persona this)
    {
        return DateTime.Now.Year - this.Età;
    }
}
```

Nota bene: `Età` viene tradotto in `this.Età`, dove `this` è il parametro che referenzia l'oggetto attraverso il quale viene invocato il metodo.

L'invocazione del metodo viene infatti tradotta nel seguente modo:

```
static void Main(string[] args)
{
    var p = new Persona { Nome = "Filippo", Età = 25 };

    int anno = p.AnnoNascita();
    int anno = Persona.AnnoNascita(p);

    Console.ReadKey();
}
```

Il parametro `this` definito in `AnnoNascita()` riceve l'oggetto `p`, che nel codice scritto dal programmatore è l'oggetto attraverso il quale viene chiamato il metodo.

Considera adesso l'ipotesi che il metodo `AnnoNascita()` sia statico:

```
public class Persona
{
    public string Nome;
    public int Età;

    public static int AnnoNascita()  // il metodo non  riceve alcun oggetto.
    {
        return DateTime.Now.Year - Età;  // Età a chi appartiene?
    }
}
```

Il linguaggio non produce alcuna trasformazione del metodo; dunque, l'uso di `Età` *implica l'accesso a un campo che non appartiene a un oggetto*, cosa chiaramente impossibile.

4.1 Uso di un metodo d'istanza all'interno dei metodi statici

Non può funzionare, e per lo stesso motivo visto in precedenza. Ipotizza di aggiungere a `Persona` un metodo statico che verifica se l'anno di nascita è precedente a un certo valore.

```
public class Persona
{
    public string Nome;
    public int Età;

    public int AnnoNascita()
    {
        return DateTime.Now.Year - Età;
    }

    public static bool SeMinoreDi(int anno)
    {
        return AnnoNascita() < anno;          // su quale oggetto viene eseguito il
    }                                         // metodo?
}
```

Un metodo d'istanza deve essere eseguito (appunto) su un'istanza, e cioè su un oggetto, *anche nel codice interno alla classe*. Come mai, allora, nei metodi non statici è possibile farlo senza dover specificare un oggetto? Perché lo fa il linguaggio per noi.

Ecco cosa accade se il metodo non è statico:

```
public class Persona
{
    public string Nome;
    public int Età;

    public int AnnoNascita()
    {
        return DateTime.Now.Year - Età;
    }

    // codice scritto dal programmatore
    public bool SeMinoreDi(int anno)
    {
        return AnnoNascita() < anno;
    }

    // traduzione del linguaggio!
    public bool SeMinoreDi(Persona this, int anno)
    {
        return this.AnnoNascita() < anno;
    }
}
```

5 Shared references

La distinzione tra variabile e oggetto ha molte implicazioni, tra le quali: più variabili possono referenziare lo stesso oggetto. Ciò, a sua volta, ha due implicazioni fondamentali:

- La assegnazioni sono assolutamente performanti, poiché in realtà non viene copiato alcun dato, ma soltanto un indirizzo di memoria.
- Parti diverse del programma possono condividere un riferimento allo stesso oggetto ed eseguire determinate operazioni su di esso.

La piena comprensione del secondo punto è fondamentale.

Considera il seguente codice, nel quale un metodo applica uno sconto a un elenco di prezzi memorizzato su un vettore:

```
static void ApplicaSconto(double[] prezzi, double sconto)
{
    for (int i = 0; i < prezzi.Length; i++)
    {
        prezzi[i] -= prezzi[i] * sconto;
    }
}
...

static void Main(string[] args)
{
    double[] prezziProdotti = { 100, 200, 150, 225 };

    ApplicaSconto(prezziProdotti, 0.1); //sconto 10%
    //-> 90 180 135 202,5
}
```

Il tutto si basa sul presupposto che al metodo `ApplicaSconto()` sia passato il *reference* del vettore `prezziProdotti`; il metodo potrà modificare il vettore attraverso la variabile locale `prezzi`.

Questo modello di gestione degli oggetti mostra qui i propri vantaggi. Nel passaggio del vettore da `Main()` a `ApplicaSconto()` non avviene in realtà nessuna copia dei dati, se non la copia dell'indirizzo tra `prezziProdotti` e `prezzi`.

Nel caso suddetto si dice che `ApplicaSconto()` produce un "effetto collaterale" (*side effect*), poiché modifica un oggetto che è stato creato fuori dal metodo.

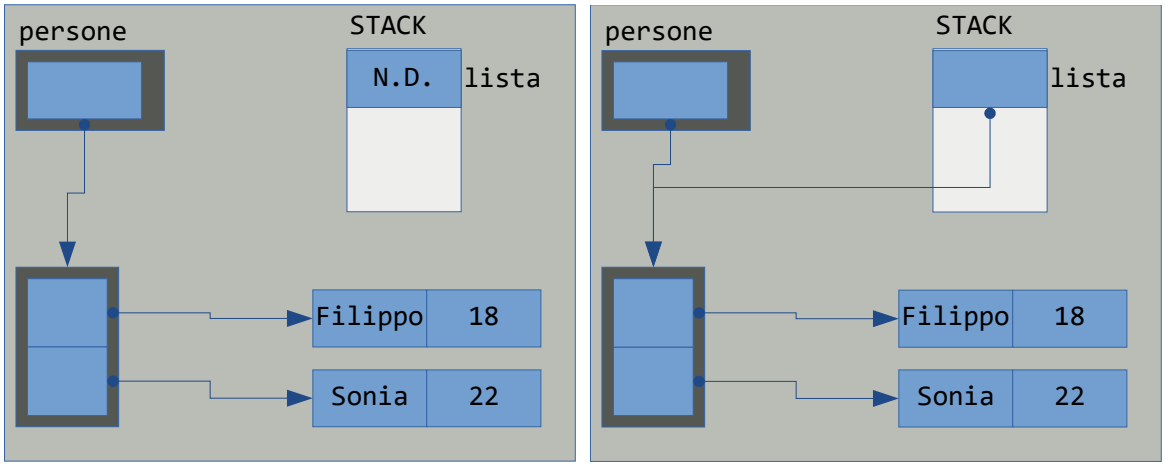
Non sempre, però, la possibilità di condividere il riferimento allo stesso oggetto è un vantaggio; infatti, non tutti gli effetti collaterali sono i benvenuti, poiché producono modifiche su oggetti gestiti altrove nel programma.

Considera la classe `ElencoPersone`. Questa definisce un metodo, `GetPersone()`, che restituisce il campo privato `elenco`. Segue un frammento di codice che usa la classe:

```
static ElencoPersone persone = new ElencoPersone();
static void Main(string[] args)
{
    persone.Add(new Persona { Nome = "Filippo", Età = 18 });
    persone.Add(new Persona { Nome = "Sonia", Età = 22 });

    var lista = persone.GetPersone(); //"lista" referenzia lo stesso oggetto
                                      //del campo privato "elenco".
    Console.ReadKey();
}
```

Lo schema seguente mostra la situazione in memoria prima e dopo l'esecuzione dell'istruzione evidenziata[1]:

Prima dell'esecuzione di GetPersone() **Dopo dell'esecuzione di GetPersone()**



Sia `lista` che il campo privato `elenco` referenziano l'oggetto `List<>` che memorizza le persone; non si tratta di una situazione desiderabile, poiché rende possibile modificare il `List<>`, "bypassando" l'interfaccia pubblica della classe `ElencoPersone`.

In sostanza, mediante `lista` è possibile produrre un effetto collaterale che modifica lo stato di un oggetto (`ElencoPersone`) senza utilizzare i metodi di quell'oggetto. In generale, una classe ben incapsulata non fornisce l'accesso alla propria rappresentazione interna, né dichiarando pubblici i campi, né restituendo direttamente quei campi (se sono dei *reference*).

Nello scenario suddetto, se decidessimo comunque di voler fornire una lista indicizzabile delle persone memorizzate dentro l'oggetto, potremmo implementare un metodo che restituisce un vettore contenente una copia dell'elenco.

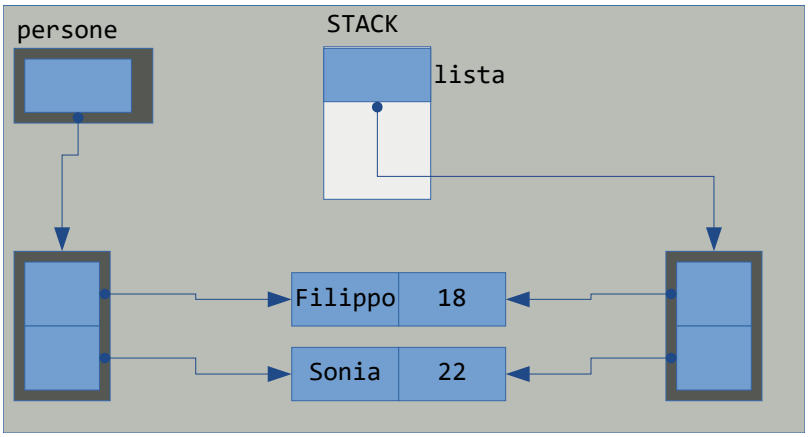
```
public ElencoPersone
{
    ...
    public Persona[] ToArray()
    {
        var lista = new Persona[Count];
```

1 Ho semplificato lo schema; infatti, anche le stringhe "Filippo" e "Sonia" sono oggetti, e in quanto tali dovrebbero essere rappresentati diversamente.

```
        for (int i = 0; i < Count; i++)
        {
            lista[i] = elenco[i];
        }
        return lista;
    }
}
```

```
static void Main(string[] args)
{
    persone.Add(new Persona { Nome = "Filippo", Età = 18 });
    persone.Add(new Persona { Nome = "Sonia", Età = 22 });

    var lista = persone.ToArray();   //"lista" è un vettore contenente una copia
                                     //dell'elenco
    Console.ReadKey();
}
```

Ecco la situazione in memoria dopo l'esecuzione del metodo `ToArray()`:



Nota bene:

- `lista` referenzia un altro oggetto (un vettore).
- Gli elementi del campo privato `elenco` e quelli del vettore `lista` condividono i riferimenti ai stessi dati.

L'ultimo punto è importante. Nel metodo `ToArray()` la copia del contenuto di `elenco` in `lista` rappresenta una copia dei riferimenti e non degli oggetti. Dopo l'operazione, dunque, `Main()` e `ElencoPersone` condividono i riferimenti agli oggetti contenenti i dati.

(In questo caso non si tratta di un problema, poiché `ElencoPersone` ha la funzione di incapsulare la gestione dei record, ma non l'uso dei record stessi.)