Record

Rappresentare i dati mediante i record

Anno 2018/2019

Indice generale

1 Introduzione....4

1.1 “Rappresentare” i dati....4

2 I limiti dei tipi predefiniti: un esempio....5

2.1 Rappresentare e organizzare i dati....5

2.2 Caricamento dei piloti....6

2.3 Visualizzazione....6

2.4 Ordinamento....6

2.5 Piloti di una determinata scuderia....7

2.5.1 Visualizzazione dei piloti di una scuderia....8

2.6 Metodo Main(): verifica delle funzioni implementate....8

2.7 Analisi del programma....9

3 Introduzione ai *record*....10

3.1 Terminologia: *record* e *istanze* di *record*....10

3.2 Struttura dei *record*....11

3.2.1 Convenzione sulla nomenclatura di record e campi....11

3.2.2 Valore iniziale dei campi (valore predefinito)....11

3.2.3 Inizializzare i campi durante la dichiarazione....11

3.3 Dichiarare e creare variabili *record*....12

3.4 Inizializzare variabili *record* (contestualmente alla dichiarazione)....12

3.5 Accesso e valorizzazione dei campi....12

3.6 Assegnazione tra variabili *record*....13

4 Usare i *record*: un esempio....14

4.1 Implementare la lista dei piloti....14

4.2 Caricamento piloti....14

4.3 Visualizzazione....15

4.4 Ordinamento....16

4.5 Piloti di una determinata scuderia....16

4.5.1 Visualizzazione dei piloti di una scuderia....16

4.6 Metodo Main()....17

4.7 Conclusioni....17

5 Dati contestuali → *record*.....18

5.1 Rappresentare “oggetti”.....18

5.1.1 Rappresentare gli oggetti mediante dei *record*.....19

6 Modelli di dati.....21

6.1 Progettare un modello di record: un esempio.....21

6.1.1 Realizzare un “modello semplice”.....22

6.1.2 Creare un “modello completo” dei dati: Squadra ↔ Calciatore.....22

6.1.3 Gestire i dati nel rispetto del modello.....23

6.1.4 Uso del modello completo nelle funzioni del programma.....24

6.2 Conclusioni.....26

7 Assegnazione e confronto tra variabili *record*.....27

7.1 Assegnazione.....27

7.2 *Value versus reference* (riferimenti e valori).....28

7.3 Passaggio di parametri.....29

7.4 Confronto per uguaglianza.....29

8 Frequently Asked Questions.....30

8.1 Perché un *record* viene definito *tipo strutturato eterogeneo*?.....30

8.2 Qual è il valore predefinito di una variabile (globale) *record*?.....30

8.3 Il tipo di un campo può essere a sua volta un *record*?.....30

8.4 Qual è il valore predefinito del campo di un *record*?.....30

8.5 I campi di un *record* possono essere *statici*?.....30

8.6 I campi di un *record* devono essere *pubblici*?.....31

8.7 Un record può essere restituito da un metodo?.....31

8.8 Il nome di una variabile può essere uguale al nome del *record* che ne definisce il tipo?.....32

8.9 Modificare i campi di un parametro *record* si riflette sull’argomento corrispondente?.....32

1 Introduzione

Il tutorial fornisce un'introduzione ai *record*, e dunque alla possibilità di definire nuovi *tipi* in grado di rappresentare accuratamente i dati del problema.

Dopo aver riepilogato l'attuale modello di rappresentazione dei dati, lo userò in un semplice esercizio di programmazione e ne mostrerò i limiti. Successivamente introdurrò i *record* e mostrerò come consentano di superare questi limiti e di semplificare lo sviluppo del programma.

1.1 "Rappresentare" i dati

Di fronte a un problema di programmazione, tra le prime operazioni da compiere ci sono:

- Stabilire come *rappresentare* i dati.
- Stabilire la loro organizzazione.

Cerchiamo di capirne il senso prendendo come esempio un semplice problema:

> Dati i mm di pioggia caduti giornalmente nella prima settimana del mese, trovare il valore medio giornaliero e il numero di giorni nei quali ha piovuto sopra la media.

I dati del problema sono:

- I mm di pioggia caduti giornalmente nella prima settimana.
- Il valore medio (in mm di pioggia).
- Il numero di giorni sopra la media.

Stabilire come rappresentare i dati significa *scegliere i tipi di dati da utilizzare*. In questo caso la scelta appropriata è usare il tipo `double` per i mm di pioggia e il tipo `int` per il numero di giorni sopra la media.

Nota bene: non ho considerato quante e quali variabili servano, né come debbano essere elaborati i dati; questo verrà dopo. Per il momento ho stabilito che ovunque, nel programma, i mm di pioggia devono essere memorizzati in variabili di tipo `double`.

L'organizzazione riguarda la *scelta delle strutture dati appropriate per memorizzare i dati*. In questo caso occorre stabilire come memorizzare i sette valori che rappresentano i mm di pioggia caduti la prima settimana. La scelta più ovvia è quella di usare un vettore o una lista di `double`.

Entrambe le operazioni richiedono un'analisi del problema in relazione ai tipi forniti dal linguaggio (`int`, `string`, `double`, vettori, liste, etc), in modo da scegliere i tipi adatti.

Come vedremo, però, in molti problemi di programmazione l'uso dei soli tipi predefiniti pone dei severi limiti alla qualità e alla semplicità del programma.

2 I limiti dei tipi predefiniti: un esempio

Considera il seguente problema:

> Realizza un programma che gestisca i piloti iscritti al campionato di MotoGP. Il programma deve prevedere
>
> 1. Caricamento dei piloti: nominativo, scuderia, punteggio.
> 2. Visualizzazione dei piloti in ordine di classifica.
> 3. Data una scuderia, visualizzare i piloti che vi appartengono.
>
> **Vincoli**
>
> Occorre separare l'elaborazione dei dati dalla visualizzazione dei risultati.
>
> (Separare l'ordinamento dalla visualizzazione della classifica. Separare la ricerca dei piloti di una scuderia dalla loro visualizzazione.)

La soluzione proposta di seguito non rappresenta un programma vero e proprio, poiché si limita a implementare le funzioni richieste senza curarsi di gestire l'interazione con l'utente.

2.1 Rappresentare e organizzare i dati

Il programma deve gestire un elenco di piloti; ogni pilota è caratterizzato da *nominativo* (`string`), *scuderia* (`string`), *punteggio* (`double`): occorrono dunque tre liste.

Nominativi	Scuderie	Punteggi
Hamilton, Lewis	Mercedes	281
Bottas, Valtteri	Mercedes	171
Vettel, Sebastian	Ferrari	241
...	...	...

A queste dovranno essere applicate le funzioni richieste dal problema: caricamento, ordinamento, visualizzazione, filtro (punto 3). Dato che le liste sono al centro del programma, è opportuno definirle come variabili globali, rendendole accessibili a tutti i metodi:

```
class Program
{
    static List<string> pilotiNome;
    static List<string> pilotiScuderia;
    static List<double> pilotiPunteggio;
    ...
}
```

2.2 Caricamento dei piloti

Questa funzione presuppone l'input da parte dell'utente; per semplicità implemento il procedimento aggiungendo i piloti direttamente nel codice:

```
static void CaricaPiloti()
{
    pilotiNome = new List<string>();
    pilotiScuderia = new List<string>();
    pilotiPunteggio = new List<double>();

    pilotiNome.Add("Hamilton, Lewis");
    pilotiScuderia.Add("Mercedes");
    pilotiPunteggio.Add(281);

    pilotiNome.Add("Bottas, Valtteri");
    pilotiScuderia.Add("Mercedes");
    pilotiPunteggio.Add(171);

    pilotiNome.Add("Vettel, Sebastian");
    pilotiScuderia.Add("Ferrari");
    pilotiPunteggio.Add(241);
    //... gli altri piloti
}
```

2.3 Visualizzazione

Questa funzione non richiede particolari spiegazioni:

```
static void VisualizzaPiloti()
{
    Console.WriteLine("{0,-20}{1,-15}{2,5}", "Nominativo", "Scuderia", "Punti");
    for (int i = 0; i < pilotiNome.Count; i++)
    {
        Console.WriteLine("{0,-20}{1,-15}{2,5}", pilotiNome[i], pilotiScuderia[i],
                                                 pilotiPunteggio[i]);
    }
}
```

2.4 Ordinamento

Per ottenere un ordine di classifica, l'ordinamento deve essere effettuato confrontando i punteggi. Quando due punteggi non sono ordinati tra loro, occorre scambiare anche i nominativi e le scuderie corrispondenti:

```
static void OrdinaPerClassifica()
{
    for (int i = 0; i < pilotiNome.Count-1; i++)
    {
```

```
        for (int j = i+1; j < pilotiNome.Count; j++)
        {
            if (pilotiPunteggio[i] < pilotiPunteggio[j])  //verifica ordine di classifica
            {
                string tmpNome = pilotiNome[i];
                pilotiNome[i] = pilotiNome[j];
                pilotiNome[j] = tmpNome;

                string tmpScuderia = pilotiScuderia[i];
                pilotiScuderia[i] = pilotiScuderia[j];
                pilotiScuderia[j] = tmpScuderia;

                double tmpPunteggio = pilotiPunteggio[i];
                pilotiPunteggio[i] = pilotiPunteggio[j];
                pilotiPunteggio[j] = tmpPunteggio;
            }
        }
    }
}
```

2.5 Piloti di una determinata scuderia

Questa funzione chiede di visualizzare i piloti di una scuderia. Ad esempio, nel caso della "Ferrari" l'output potrebbe essere:

```
Piloti della Ferrari:
Vettel, Sebastian    241 pt
Raikkonen, Kimi      174 pt
```

La sua implementazione sarebbe semplice, se non esistesse un vincolo che impone di separare le funzioni di elaborazione da quelle di visualizzazione. Posto questo vincolo, è necessario implementare un metodo che, ricevuta la scuderia, restituisca i piloti che vi gareggiano.

```
static void PilotiScuderia(string scuderia, out List<string> nomi,
                                            out List<double> punteggi)
{
    nomi = new List<string>();
    punteggi = new List<double>();
    for (int i = 0; i < pilotiScuderia.Count; i++)
    {
        if (pilotiScuderia[i] == scuderia)
        {
            nomi.Add(pilotiNome[i]);
            punteggi.Add(pilotiPunteggio[i]);
        }
    }
}
```

2.5.1 Visualizzazione dei piloti di una scuderia

Il metodo riceve la scuderia e i piloti che vi gareggiano (solo nomi e punteggi):

```
static void VisualizzaPilotiScuderia(string scuderia, List<string> nomi,
                                                       List<double> punteggi)
{
    Console.WriteLine("Piloti della {0}:", scuderia);
    for (int i = 0; i < nomi.Count; i++)
    {
        Console.WriteLine("{0,-20}{1} pt", nomi[i], punteggi[i]);
    }
}
```

2.6 Metodo Main(): verifica delle funzioni implementate

```
static void Main(string[] args)
{
    CaricaPiloti();
    OrdinaPerClassifica();
    VisualizzaPiloti();

    string scuderia = "Ferrari";  //sostituisce l'input dell'utente
    List<string> nomi;
    List<double> punteggi;
    PilotiScuderia(scuderia, out nomi, out punteggi);
    VisualizzaPilotiScuderia(scuderia, nomi, punteggi);
}
```

Il programma funziona, come è dimostrato dall'output (ho inserito soltanto sei piloti):

```
Classifica generale
Nominativo          Scuderia       Punti
Hamilton, Lewis     Mercedes         281
Vettel, Sebastian   Ferrari          241
Raikkonen, Kimi     Ferrari          174
Bottas, Valtteri    Mercedes         171
Verstappen, Max     Red Bull         148
Ricciardo, Daniel   Red Null         126


Piloti della Ferrari:
Vettel, Sebastian   241 pt
Raikkonen, Kimi     174 pt
```

2.7 Analisi del programma

Analizzando il codice emerge una questione che riguarda la rappresentazione dei dati. Il testo del problema fa riferimento ai piloti: caricarli, ordinarli, visualizzarli, filtrarli. Dunque, il problema ruota attorno al concetto di *pilota*, ma nel programma il *pilota* non è rappresentato.

Il programma gestisce i singoli dati – *nominativi*, *scuderie*, *punteggi* – ma non tratta questi dati come un'unita di informazione, anche se sarebbe estremamente utile, poiché in molti casi i dati vengono elaborati contestualmente, eseguendo su di essi la stessa operazione.

In conclusione, l'uso dei soli tipi predefiniti consente di rappresentare i dati, ma *non di "catturare" la relazione che hanno tra loro come appartenenti a un'unità di informazione di livello superiore*: il *pilota*.

3 Introduzione ai *record*

Il linguaggio C# (come qualsiasi altro linguaggio) consente di definire nuovi tipi allo scopo di rappresentare entità che i tipi predefiniti non possono rappresentare. Il *pilota* ne è un esempio.

Un pilota è caratterizzato da tre dati: nominativo, scuderia e punteggio; lo si può rappresentare mediante il seguente costrutto:

```
class Pilota
{
    public string Nominativo;
    public string Scuderia;
    public double Punteggio;
}
```

Questa definizione introduce un nuovo tipo di dato – il `Pilota`, appunto. Convenzionalmente, questo costrutto viene chiamato *record* (registrazione).

Un record è un tipo di dato che definisce campi, o variabili, anche di tipo diverso tra loro.

È definito ***tipo di dato strutturato***, perché composto da più parti – i campi – ed ***eterogeneo***, poiché le sue parti possono essere di tipo diverso. (In contrapposizione agli *array*, chiamati tipi *strutturati omogenei*, poiché i loro elementi sono dello stesso tipo.)

3.1 Terminologia: *record* e *istanze* di *record*

Prima di proseguire è opportuno fare un po' di chiarezza sulla terminologia. Con il termine *record* (in corsivo) intendo un tipo di dato, come `Pilota`. Analogamente a ogni altro tipo di dato – `string`, `int`, `double`, etc – i *record* servono per creare i dati nel programma, i quali sono memorizzati in variabili, chiamate anche *istanze*. Dunque, nel seguente codice:

```
class Pilota
{
    public string Nominativo;
    public string Scuderia;
    public double Punteggio;
}
...
string scuderia = "Ferrari"
Pilota p1 = new Pilota();
Pilota p2 = new Pilota();
...
```

`scuderia` è un'*istanza* del tipo `string`, mentre `p1` e `p2` sono due *istanze* del tipo `Pilota`.[1]

Informalmente si usa dire che `scuderia` è una stringa, mentre `p1` e `p2` sono due record, nel senso di due istanze di *record*. (Per evitare ambiguità, nel testo uso il corsivo per dare a *record* il significato di *tipo*, mentre uso il carattere normale per dare il significato di *istanza*).

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

3.2 Struttura dei *record*

La definizione dei *record* segue la sintassi (semplificata):

```
class <nome record>
{
    public <tipo> <campo>;opz
    public <tipo> <campo>;opz
    ...
}
```

Ricapitolando:

- Un *record* è definito mediante la parola chiave `class`, che introduce un nuovo tipo, rappresentato dal nome del *record*.
- Un *record* definisce zero o più ***campi***, o variabili. (Benché inutile, è possibile definire un *record* vuoto.)
- La definizione di un campo deve essere preceduta dalla parola chiave `public`. (In sua assenza non viene segnalato un errore, ma il campo è inutilizzabile.)

3.2.1 Convenzione sulla nomenclatura di record e campi

Ogni linguaggio di programmazione ha le sue convenzioni, che non sono regole formali, ma norme adottate dalla comunità di programmatori. Per quanto riguarda i *record*: il nome del *record* e il nome dei campi deve cominciare con la maiuscola

3.2.2 Valore iniziale dei campi (valore predefinito)

Come accade per le variabili statiche, se a un campo non viene assegnato un valore, mantiene il *valore predefinito*, che dipende dal tipo del campo: `null` per le stringhe, 0 per tipi numerici, `false` per i `bool`, etc.

3.2.3 Inizializzare i campi durante la dichiarazione

È possibile fornire un valore iniziale ai campi durante la dichiarazione. Raramente è utile, eccetto quando il campo è un vettore o una lista. Segue la definizione del *record* `Studente`, che dichiara e crea un campo *array*:

```
class Studente
{
    public string Nominativo;
    public double[] voti = new double[10];
}
```

3.3 Dichiarare e creare variabili *record*

La dichiarazione di una variabile *record* e la sua creazione sono due fasi distinte, anche se possono coesistere nella stessa istruzione. La seguente istruzione dichiara la variabile `p`:

```
Pilota p;
```

Dopo la dichiarazione, `p` non contiene alcun valore e dunque è inutilizzabile. Per questo occorre eseguire la creazione:

```
p = new Pilota();
```

Adesso i tre campi del record `p` sono *inizializzati*, hanno cioè un valore predefinito: `Nominativo` e `Scuderia` contengono `null`, mentre `Punteggio` contiene zero.

Dichiarazione e creazione possono essere sono scritte nella stessa istruzione:

```
Pilota p = new Pilota();
```

3.4 Inizializzare variabili *record* (contestualmente alla dichiarazione)

Le variabili possono essere inizializzate con valori specifici durante la dichiarazione, questo vale anche per le variabili *record*. Poiché contengono più campi, l'inizializzazione ha una sintassi un po' complicata. Segue un esempio di inizializzazione di una variabile `Pilota`:

```
Pilota p = new Pilota()   // parentesi tonde facoltative
{
    Nominativo = "Vettel, Sebastian",
    Scuderia = "Ferrari",
    Punteggio = 241
};
```

Nota bene:

- I campi vengono inizializzati dentro un *blocco* (coppia di `{}`).
- Le inizializzazioni – `<campo> = <valore>` – sono separate dalla virgola.
- La coppia di parentesi tonde `()` dopo `new Pilota` (in grigio) è facoltativa.

Se uno o più campi vengono omessi, mantengono il loro valore predefinito. (La suddivisione in più righe non è obbligatoria, ma è utile a rende più leggibile il codice.)

3.5 Accesso e valorizzazione dei campi

Data una variabile *record* già creata, l'accesso ai campi avviene nella forma: `variabile.campo`. Per valorizzare i campi si può dunque scrivere:

```
Pilota p = new Pilota();
p.Nominativo = "Vettel, Sebastian";
p.Scuderia = "Ferrari";
```

```
p.Punteggio = 241;
```

Per utilizzare un campo, ad esempio per assegnarlo a una variabile, si scrive:

```
string scuderia = p.Scuderia;
```

3.6 Assegnazione tra variabili *record*

Si può assegnare una variabile *record* a un'altra come si fa per qualsiasi altro tipo di variabile. Dopo l'assegnazione, le due variabili condividono lo stesso contenuto:

```
//"p" contiene i dati di Sebastian Vettel
Pilota p2 = p;
// anche "p2" contiene i dati di Sebastian Vettel
```

Allo stesso modo si può passare un argomento *record* a un parametro dello stesso tipo.

(Nota a margine, dire che due variabili condividono lo stesso contenuto appare ambiguo, ma è il risultato prodotto dall'assegnazione tra due variabili *record*. Vedi 7 per un approfondimento.)

4 Usare i *record*: un esempio

Di seguito mostro come modificare il programma di gestione dei piloti in modo che utilizzi il *record* `Pilota`.

4.1 Implementare la lista dei piloti

Ora che esiste `Pilota`, è sufficiente una sola lista:

```
class Program
{
    static List<Pilota> piloti;

    static List<string> pilotiNome;
    static List<string> pilotiScuderia;
    static List<double> pilotiPunteggio;
    ...
}
```

4.2 Caricamento piloti

La funzione di caricamento implica la creazione, la valorizzazione e l'inserimento nella lista `piloti` di un record per ogni pilota iscritto al campionato:

```
static void CaricaPiloti()
{
    piloti = new List<Pilota>();

    Pilota p = new Pilota();
    p.Nominativo = "Hamilton, Lewis";
    p.Scuderia = "Mercedes";
    p.Punteggio = 281;
    piloti.Add(p);

    p = new Pilota();
    p.Nominativo = "Bottas, Valtteri";
    p.Scuderia = "Mercedes";
    p.Punteggio = 171;
    piloti.Add(p);

    p = new Pilota();
    p.Nominativo = "Vettel, Sebastian";
    p.Scuderia = "Ferrari";
    p.Punteggio = 241;
    piloti.Add(p);
    //... gli altri piloti
}
```

Alternativamente, è possibile creare e inizializzare i record in una sola istruzione:

```
static void CaricaPiloti()
{
    piloti = new List<Pilota>();

    Pilota p = new Pilota
    {
        Nominativo = "Hamilton, Lewis",
        Scuderia = "Mercedes",
        Punteggio = 241
    };

    piloti.Add(p);
    ...
}
```

Infine, si può fare a meno della variabile, creando e inizializzando il record mentre lo si passa al metodo `Add()`:

```
static void CaricaPiloti()
{
    piloti = new List<Pilota>();

    piloti.Add(new Pilota
    {
        Nominativo = "Hamilton, Lewis",
        Scuderia = "Mercedes",
        Punteggio = 241
    });

    ...
}
```

4.3 Visualizzazione

Poiché ora esiste una sola lista, è opportuno, e più semplice, usare un `foreach`:

```
static void VisualizzaPiloti()
{
    Console.WriteLine("Classifica generale");
    Console.WriteLine("{0,-20}{1,-15}{2,5}", "Nominativo", "Scuderia", "Punti");
    foreach (Pilota p in piloti)
    {
        Console.WriteLine("{0,-20}{1,-15}{2,5}", p.Nominativo, p.Scuderia, p.Punteggio);
    }
}
```

Nota bene: la variabile *iteratore* (`p`, in questo caso), assume il valore di ogni elemento della lista, dal primo al all'ultimo.

4.4 Ordinamento

Il metodo di ordinamento guadagna molto dall'uso del *record* `Pilota`; infatti, lo scambio tra due piloti si riduce alle classiche tre istruzioni:

```
static void OrdinaPerClassifica()
{
    for (int i = 0; i < piloti.Count-1; i++)
    {
        for (int j = i+1; j < piloti.Count; j++)
        {
            if (piloti[i].Punteggio < piloti[j].Punteggio)
            {
                Pilota tmp = piloti[i];
                piloti[i] = piloti[j];
                piloti[j] = tmp;
            }
        }
    }
}
```

4.5 Piloti di una determinata scuderia

Nella versione originale, il metodo `PilotiScuderia()` restituiva due liste, `nomi` e `punteggi`, mediante *parametri out*. La nuova versione, più semplice, restituisce una lista di piloti:

```
static List<Pilota> PilotiScuderia(string scuderia)
{
    List<Pilota> ris = new List<Pilota>();
    foreach (Pilota p in piloti)
    {
        if (p.Scuderia == scuderia)
            ris.Add(p);
    }
    return ris;
}
```

4.5.1 Visualizzazione dei piloti di una scuderia

Anche in questo caso il codice si semplifica:

```
static void VisualizzaPilotiScuderia(string scuderia, List<Pilota> piloti)
{
    Console.WriteLine("\nPiloti della {0}:", scuderia);
    foreach (Pilota p in piloti)
    {
        Console.WriteLine("{0,-20}{1} pt", p.Nominativo, p.Punteggio);
    }
}
```

4.6 Metodo Main()

In `Main()` si semplifica la chiamata ai metodi `PilotiScuderia()`: e `VisualizzaPilotiScuderia()`:

```
static void Main(string[] args)
{
    CaricaPiloti();
    OrdinaPerClassifica();

    VisualizzaPiloti();

    string scuderia = "Ferrari";
    List<Pilota> pilotiInScuderia = PilotiScuderia(scuderia);
    VisualizzaPilotiScuderia(scuderia, pilotiInScuderia);

}
```

4.7 Conclusioni

La corretta rappresentazione dei dati è la premessa fondamentale per lo sviluppo del programma. Scegliere i giusti tipi predefiniti è molto importante, ma spesso insufficiente. In molti problemi di programmazione è necessario rappresentare delle entità - *pilota*, *cliente*, *studente*, *dipendente*, etc – che sono unità di informazione più grandi del semplice dato. In alcuni programmi esistono gruppi di dati che mostrano una relazione reciproca, poiché vengono sempre elaborati contestualmente. In tutti questi casi è opportuno superare il limite dei tipi predefiniti e definire nuovi tipi in grado di fornire una rappresentazione accurata dei dati. Ciò semplifica lo sviluppo del codice, rendendolo inoltre più semplice da comprendere e manutenere (modificare).

5 Dati contestuali → *record*

Nell'esempio precedente, l'opportunità di definire un *record* – `Pilota` – emerge chiaramente dalla semplice analisi del problema, come emerge la struttura del *record*, comprendente i campi *nominativo*, *nazione* e *punteggio*. In altre situazioni, d'altra parte, può accadere di cominciare a sviluppare il programma con un determinato modello dei dati, per poi rendersi conto che alcuni gruppi di dati sono correlati tra loro ed elaborati contestualmente. In questo caso occorre valutare attentamente l'opportunità di "aggregare" questi dati mediante un *record*.

Segue un esempio basato sul gioco "colpisci l'invasore".

5.1 Rappresentare "oggetti"

Considera il testo del problema "colpisci l'invasore" (ne riporto soltanto l'inizio):

> Si vuole realizzare un gioco nel quale una navicella spara un proiettile allo scopo di colpire un bersaglio che attraversa orizzontalmente lo schermo:
>
> 

In sostanza, ci sono da gestire tre oggetti nello schermo, dei quali uno, la navicella, è sempre visibile, mentre gli altri no. Dopo una breve analisi, emerge la seguente rappresentazione dei dati necessari a gestire i tre oggetti:

```
class Program
{
    static int navicellaX;
    static int navicellaY;
    static int navicellaDirezione
    static string navicella;

    static int proiettileX;
    static int proiettileY;
    static bool proiettileVisibile;
    static string proiettile;

    static int bersaglioX;
    static int bersaglioY;
    static bool bersaglioVisibile;
    static string bersaglio;
    ...
}
```

navicella

proiettile

bersaglio

Anche solo considerando i nomi delle variabili, emerge l'esistenza di tre gruppi distinti di variabili contestuali.

Con il termine *contestuale* intendo dire che una variabile ha senso d'esistere soltanto se considerata insieme alle altre dello stesso gruppo. In altri termini, le variabili di un gruppo hanno una precisa relazione tra loro e spesso vengono elaborate nello stesso frammento di codice (contestualmente, appunto).

È la tipica situazione nella quale i *record* consentono una migliore rappresentazione dei dati del problema.

5.1.1 Rappresentare gli oggetti mediante dei record

Occorre definire un nuovo tipo per ogni oggetto del gioco:

```
class Navicella
{
    public string scafo;
    public int X;
    public int Y;
    public int Direzione;
}

class Proiettile
{
    public string scafo;
    public int X;
    public int Y;
    public bool Visibile;
}

class Bersaglio
{
    public string scafo;
    public int X;
    public int Y;
    public bool Visibile;
}
```

Sulla base dei nuovi tipi, sono sufficienti tre variabili per memorizzare tutti i dati:

```
class Program
{
    static Navicella navicella;
    static Bersaglio bersaglio;
    static Proiettile proiettile;
    ...
}
```

Naturalmente, l'obiettivo non è tanto quello di "risparmiare" sul numero di variabili, ma definire dei tipi in grado di rappresentare i concetti del problema, semplificarne l'elaborazione e facilitare lo sviluppo del programma.

Quest'ultimo punto è particolarmente interessante. Supponi di voler potenziare il programma, in modo che sia possibile sparare più proiettili contemporaneamente. Per quanto riguarda le variabili necessarie, la modifica da effettuare è estremamente semplice:

```
class Program
{
    static Navicella navicella;
    static Bersaglio bersaglio;
    static Proiettile proiettile;
    static List<Proiettile> proiettili;
    ...
}
```

Al contrario, usando un modello privo di *record*, sarebbe necessario definire quattro liste, una per ogni dato di un proiettile:

```
List<int> proiettiliX;
List<int> proiettiliY;
List<bool> proiettiliVisibile;
List<string> proiettili;
...
```

E ogni processo di elaborazione dei proiettili dovrebbe tenere conto di questa separazione dei dati di un singolo proiettile.

6 Modelli di dati

Nel programma di gestione del campionato MotoGP esiste un unico concetto da elaborare – il *pilota* – e dunque è sufficiente definire un *record* che lo rappresenta; ma non esiste alcun limite al numero di nuovi tipi che è possibile implementare.

In alcuni casi i nuovi tipi non hanno (o non è necessario che abbiano) un legame diretto tra loro, come nel caso di `Navicella`, `Bersaglio` e `Proiettile`. Ad esempio, si può immaginare una nuova versione del programma MotoGP che gestisca anche le informazioni relative alle gare da svolgere. In questo caso avremmo due *record*:

```
class Pilota
{
    public string Nominativo;
    public string Scuderia;
    public double Punteggio;
}
```

```
class Gara
{
    public DateTime Data;
    public string Circuito;
    public string Nazione;
}
```

In molti scenari, comunque, i *record* hanno, o possono avere, una relazione diretta tra loro. In pratica: il tipo di uno o più campi di un *record* è a sua volta un *record*. È l'analisi del problema che deve far emergere, se esiste, questa relazione, avendo l'obiettivo fondamentale di rappresentare il più accuratamente possibile i dati e le relazioni che hanno tra loro.

L'analisi ha dunque lo scopo di creare un ***modello dei dati***; infatti non si tratta più soltanto di scegliere quali tipi predefiniti usare, ma di stabilire con quale "forma" rappresentare i dati e le loro relazioni.

6.1 Progettare un modello di record: un esempio

Considera il seguente problema:

> Realizza un programma per la gestione del campionato di calcio di serie A. Occorre gestire le informazioni sui calciatori (nominativo e gol segnati) e le squadre (nome e punti in classifica).
>
> Implementa le seguenti funzioni:
>
> 1. Caricamento dei dati.
> 2. Visualizzazione delle statistiche:
>   a) elenco squadre iscritte al campionato (nome, punteggio e gol totali segnati);
>   b) elenco giocatori, suddivisi per squadra (nome e gol, segnati);
>   c) dato il nominativo di un calciatore, visualizzare tutte le informazioni disponibili, comprese quelle sulla squadra;
>   d) ...

Qui mi pongo l'obiettivo di stabilire come rappresentare i dati del problema (dunque non fornirò una soluzione al problema).

6.1.1 Realizzare un "modello semplice"

Esistono due concetti: *squadra* e *calciatore*. Possono essere rappresentati mediante due *record*:

```
class Calciatore
{
    public string Nominativo;
    public string NomeSquadra;
    public double Gol;
}
```

```
class Squadra
{
    public string Nome;
    public double Punti;
}
```

Nel codice applicativo saranno definite due liste, una per i calciatori, l'altra per le squadre:

```
static List<Calciatore> calciatori;
static List<Squadra> squadre;
```

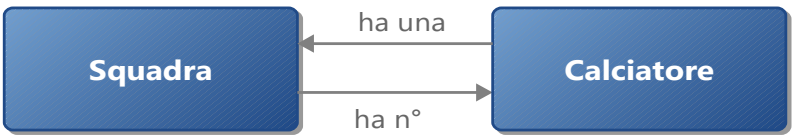
Si tratta di una soluzione valida, ma che non cattura la relazione tra i due concetti: *ogni calciatore appartiene a una squadra*. Naturalmente, nel programma questa relazione viene costruita utilizzando il campo `NomeSquadra` del *record* `Calciatore`. Ad esempio, si può rispondere al punto 2.a con il seguente codice:

```
static void VisualizzaInfoSquadre()
{
    foreach (var squadra in squadre)
    {
        double golSquadra = 0;
        foreach (var c in calciatori)
        {
            if (c.NomeSquadra == squadra.Nome)    //il calciatore appartiene alla squadra?
                golSquadra += c.Gol;

        }
        Console.WriteLine("{0} {1} {2}", squadra.Nome, squadra.Punti, golSquadra);
    }
}
```

6.1.2 Creare un "modello completo" dei dati: Squadra ↔ Calciatore

Creare un modello completo significa dare forma alle relazioni che i tipi di dati hanno tra loro. Nel problema in questione, le relazioni possono essere così schematizzate:



E si leggono: *un calciatore appartiene a una squadra*; *una squadra ha "n" calciatori*.

Segue una nuova versione dei *record* `Squadra` e `Calciatore`:

```
class Calciatore
{
    public string Nominativo;
    public Squadra Squadra;
    public double Gol;
}

class Squadra
{
    public string Nome;
    public double Punti;
    public List<Calciatore> Calciatori;
}
```

Sulla base di questo modello è possibile memorizzare tutti i dati del campionato in un'unica lista:

```
static List<Squadra> squadre;
```

6.1.3 Gestire i dati nel rispetto del modello

Perché il modello sia utile è necessario che i dati siano memorizzati correttamente. Segue un metodo che carica dei dati predefiniti (come in 2.2).

```
static void CaricaDatiCampionato()
{
    Squadra squadra = new Squadra
    {
        Nome = "Juventus",
        Punti = 23,
        Calciatori = new List<Calciatore>()
    };

    //inserisce i calciatori nella squadra appena creata
    squadra.Calciatori.Add(
        new Calciatore
        {
            Nominativo = "Ronaldo, Cristiano",
            Gol = 5,
            Squadra = squadra   //->assegna al calciatore la squadra di appartenenza
        });

    squadra.Calciatori.Add(
        new Calciatore
        {
            Nominativo = "Bonucci, Leonardo",
            Gol = 0,
            Squadra = squadra   //->assegna al calciatore la squadra di appartenenza
        });
    //...  altri calciatori e altre squadre
}
```

Alcune considerazioni:

- Ho usato la tecnica descritta in 3.2.3, che crea un record e ne inizializza immediatamente i campi.
- Con questa tecnica ho creato ogni calciatore direttamente nell'istruzione che lo inserisce nella lista calciatori di una squadra.
- Durante la creazione di un calciatore, l'ho associato alla squadra di appartenenza mediante il campo `Squadra`.

(Nota bene: con due *record* che si referenziano l'un l'altro, il codice da scrivere può essere complicato.)

6.1.4 Uso del modello completo nelle funzioni del programma

Avere un modello che rappresenta i dati e le loro relazioni semplifica lo sviluppo del programma. A destra viene proposta la nuova soluzione al punto 2.a, confrontata con la precedente:

Uso del modello semplice

```
static void VisualizzaInfoSquadre()
{
    foreach (var squadra in squadre)
    {
        double golSquadra = 0;
        foreach (var c in calciatori)
        {
           if (c.Squadra == squadra.Nome)
                golSquadra += c.Gol;
        }
        Console.WriteLine(...);
    }
}
```

Uso del modello completo

```
static void VisualizzaInfoSquadre()
{
    foreach (var squadra in squadre)
    {
        double golSquadra = 0;
        foreach (var c in squadra.Calciatori)
        {
            golSquadra += c.Gol;
        }
        Console.WriteLine(...)
    }
}
```

Considera adesso il punto 2.d: *dato il nominativo di un calciatore, visualizza tutte le informazioni disponibili, comprese quelle sulla squadra*.

Il modello completo semplifica la separazione del processo di ricerca da quello di visualizzazione del risultato:

```
static void Main(string[] args)
{
    CaricaDatiCampionato();
    Calciatore c = CalciatoreByNominativo("Ronaldo, Cristiano");
    if (c != null)
        VisualizzaCalciatore(c);
    else
        Console.WriteLine("Calciatore non trovato");
}
```

```
static void VisualizzaCalciatore(Calciatore c)
{
    var squadra = c.Squadra;
    Console.WriteLine("{0} {1} {2} {3}", c.Nominativo, c.Gol, squadra.Nome, squadra.Punti);
}

static Calciatore CalciatoreByNominativo(string nominativo)
{
    foreach (var squadra in squadre)
    {
        foreach (var c in squadra.Calciatori)
        {
            if (c.Nominativo == nominativo)
                return c;
        }
    }
    return null;
}
```

Il modello semplice complica questa separazione: poiché `Calciatore` non referenzia la squadra di appartenenza ma soltanto il suo nome, non è possibile ottenere tutte le informazioni richieste restituendo semplicemente il record del calciatore cercato.

La soluzione della funzione di ricerca si complica parecchio:

```
static bool CalciatoreByNominativo(string nominativo, out Squadra squadra,
                                                       out Calciatore calciatore)
{
    calciatore = null;
    squadra = null;
    foreach (var c in calciatori)   // cerca il calciatore
    {
        if (c.Nominativo == nominativo)
        {
            calciatore = c;
            break;
        }
    }

    if (calciatore == null)      // se non lo trova, inutile cercare la squadra
        return false;

    foreach (var s in squadre)   // cerca la squadra
    {
        if (s.Nome == calciatore.Squadra)
        {
            squadra = s;
            break;
        }
    }
    return true;
}
```

Naturalmente, si complica anche l'uso:

```
static void Main(string[] args)
{
    CaricaDatiCampionato();
    Calciatore c;
    Squadra squadra;
    bool trovato = CalciatoreByNominativo("Ronaldo, Cristiano", out squadra, out c);
    if (trovato == true)
        VisualizzaCalciatore(c);
    else
        Console.WriteLine("Calciatore non trovato");
}
```

6.2 Conclusioni

Progettare un modello dei dati non è, concettualmente, un'operazione diversa dallo scegliere i tipi di dati predefiniti da usare. Considera i problemi usati come esempio.

In 2.1 ho stabilito quali tipi predefiniti usare per rappresentare le informazioni sui piloti: *nominativo* (`string`), *squadra* (`string`) e *punteggio* (`double`).

In 3 ho mostrato che *nominativo*, *squadra* e *punteggio* non sono dati isolati, ma hanno una relazione reciproca. Utilizzando un *record* è possibile dare forma a questa relazione mediante un nuovo tipo di dato: `Pilota`.

Infine, in 6.1 ho mostrato, con i tipi `Squadra` e `Calciatore`, che anche i *record* possono avere una relazione tra loro, e che dare forma a questa relazione semplifica lo sviluppo del programma.

In sostanza, nell'analizzare i dati di un problema è necessario adottare il giusto livello di astrazione. Se dei concetti non possono essere convenientemente rappresentati da tipi predefiniti, è opportuno definire nuovi tipi mediante *record*. Se tra i concetti rappresentati esiste una relazione, è opportuno che questa venga implementata nei tipi suddetti.

7 Assegnazione e confronto tra variabili *record*

Per un corretto uso delle variabili *record* è fondamentale comprendere il loro comportamento nelle due operazioni principali: assegnazione e confronto per uguaglianza. Questo dipende dal modo in cui sono gestite in memoria, lo stesso modo adottato per *array* e liste.

Qui non approfondirò il modello di memorizzazione dei *record*, mi limiterò a evidenziare il loro comportamento nelle due operazioni.

7.1 Assegnazione

Considera il seguente codice, nel quale una variabile intera viene assegnata a un'altra:

```
int a = 10;
int b = a; //-> a: 10   b: 10
a = 30;    //-> a: 30   b: 10
```

Di seguito mostro la situazione prima e dopo l'esecuzione dell'ultima istruzione:

Dunque: `a` e `b` sono oggetti distinti che possono avere lo stesso valore, esattamente come due fratelli gemelli sono identici, pur essendo persone distinte. La figura di destra dimostra che qualunque modifica ad `a` non influenza `b`, e viceversa.

Considera adesso il *record* `Posizione`:

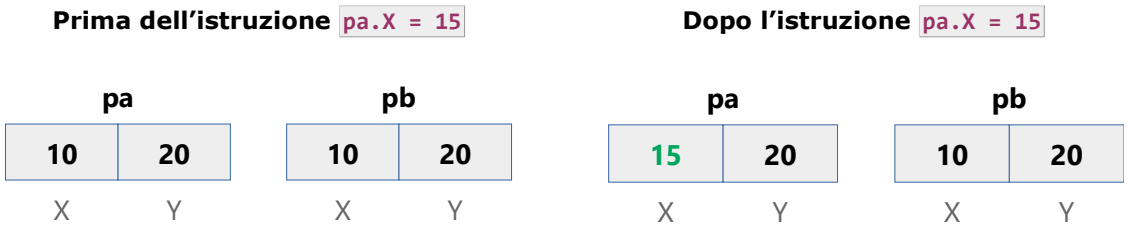
```
class Posizione
{
    public int X;
    public int Y;
}
```

Segue un frammento di codice che assegna una variabile *record* a un'altra:

```
Posizione pa = new Posizione { X = 10, Y = 20 };
Posizione pb = pa;  //-> pa.X: 10   pa.Y: 20       pb.X: 10    pb.Y: 20
pa.X = 15;          //-> pa.X: 15   pa.Y: 20       pb.X: ?     pb.Y: 20
```

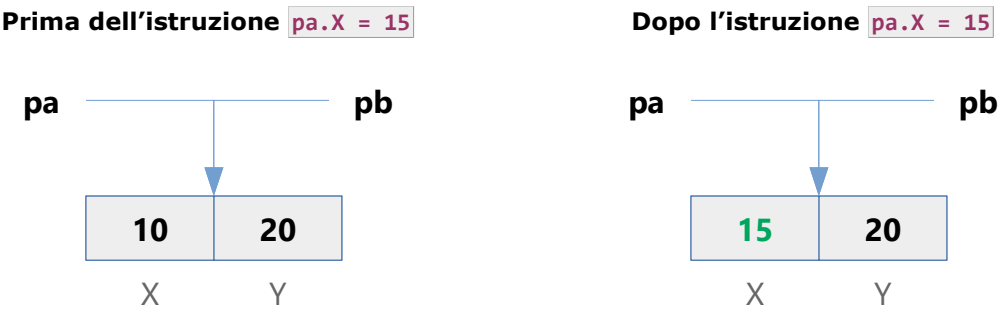
Ci si aspetta che, dopo l'istruzione `pb = pa`, esistano due variabili *record* dal contenuto identico. Inoltre, ci si aspetta che dopo la modifica del campo `pa.X` le due variabili non siano più uguali.

In conclusione, ci si aspetta la seguente situazione:



Ma non è affatto così! In realtà accade questo:

```
Posizione pa = new Posizione { X = 10, Y = 20 };
Posizione pb = pa;  //-> pa e pb referenziano lo stesso oggetto!
pa.X = 15;          //-> pa.X: 15   pa.Y: 20       pb.X: 15     pb.Y: 20
```



Le figure mostrano che dopo l'assegnazione: `pb = pa`, entrambe le variabili fanno riferimento allo stesso oggetto! Per questo motivo qualunque modifica sui campi di `pa` si riflette sui campi di `pb` e viceversa, poiché in realtà non esistono affatto due oggetti distinti!

7.2 *Value versus reference* (riferimenti e valori)

Appare evidente che il comportamento del tipo `int` (nonché `double`, `char`, `byte`, etc) differisce da quello dei *record* (nonché *array* e liste). Il motivo dipende dal modo in cui il linguaggio gestisce le due categorie di tipi, che sono definiti[2]:

- *Value type* (tipi valore): `int`, `double`, `char`, `byte`, etc.
- *Reference type* (tipi riferimento): *record*, *array*, liste, etc.

Nei *tipi valore*, due variabili sono sempre oggetti distinti. Possono avere un contenuto identico, (lo stesso valore, appunto), ma non potranno mai condividerlo. Ne consegue che qualsiasi modifica fatta a una variabile di *tipo valore* non potrà mai influenzare il contenuto di un'altra variabile.

Nei *tipi riferimento*, le variabili non memorizzano il valore, ma un riferimento all'oggetto (alla area di memoria) che lo contiene. Ne consegue che più variabili possono referenziare lo stesso oggetto (la stessa area di memoria) e dunque condividere lo stesso valore.

2 Nota bene, non cito il tipo **string**, poiché si tratta di un tipo particolare. Appartiene alla categoria dei *reference type*, ma si comporta (apparentemente) come un *value type*.

7.3 Passaggio di parametri

Il passaggio di parametri (di input) a un metodo segue le stesse regole dell'assegnazione, come dimostra il seguente codice, nel quale un metodo modifica i campi di un record:

```
static void Main(string[] args)
{
    Posizione pa = new Posizione() { X = 10, Y = 20 };
    Trasla(pa, 5);  // il parametro "p" referenzia lo stesso oggetto di "pa"
    //->p.X: 15   p.Y: 25;
}


static void Trasla(Posizione p, int distanza)
{
    p.X = p.X + distanza;
    p.Y = p.Y + distanza;
}
```

Il metodo modifica il parametro `p` e, contestualmente, l'argomento `pa`; è così perché `p` e `pa` referenziano lo stesso oggetto.

7.4 Confronto per uguaglianza

Una volta compresa la differenza tra variabili di *tipo valore* e di *tipo riferimento* si capisce che il confronto tra due variabili dipende anche da quale categoria appartengono: l'operazione eseguita è esattamente la stessa; è la natura delle variabili che determina l'esito.

Mettiamo nuovamente a confronto interi e *record*:

Confronto tra interi

```
int a = 20;
int b = 20;

if (a == b)  //-> true
...
```

Confronto tra record

```
Posizione pa = new Posizione() { X = 10, Y = 20 };
Posizione pb = new Posizione() { X = 10, Y = 20 };

if (pa == pb) //-> false
...
```

A sinistra vengono confrontati i contenuti di `a` e `b`; poiché sono uguali, il confronto produce `true`. Anche a destra vengono confrontati i valori di `pa` e `pb`, ma le due variabili non contengono X e Y, ma i riferimenti agli oggetti contenenti i due campi. Dunque, gli oggetti referenziati da `pa` e `pb` sono uguali, *ma le due variabili sono diverse* (referenziano zone di memoria diverse).

L'esempio fa emergere una regola fondamentale: il confronto tra due variabile *record* non stabilisce l'uguaglianza tra i rispettivi campi, *stabilisce se referenziano lo stesso oggetto*!

Concludendo, se volessimo verificare l'uguaglianza tra i valori di `pa` e `pb` dovremmo scrivere:

```
if (pa.X == pb.X && pa.Y == pb.Y) //-> true
...
```

8 Frequently Asked Questions

Di seguito pongo alcune questioni sulla definizione e sull'uso dei *record*, facendolo sotto forma di F.A.Q. e cioè di domande comuni alle quali fornisco una risposta e una spiegazione.

8.1 Perché un *record* viene definito *tipo strutturato eterogeneo*?

Perché, innanzitutto, ha una *struttura*, è formato cioè da più parti, i *campi*. Questa struttura è *eterogenea*, perché i campi possono essere di tipo diverso tra loro. In confronto, gli *array* sono definiti *tipi strutturati omogenei*, perché sono dotati anch'essi di una *struttura* (gli elementi), nella quale, però, gli elementi sono tutti dello stesso tipo.

8.2 Qual è il valore predefinito di una variabile (globale) *record*?

È `null`, esattamente come per *array* e liste.

Supponi di aver definito il *record* `Atleta`; il codice seguente mostra lo stato iniziale di tre variabili globali:

```
class Program
{
    static int numAtleti;   //-> 0
    static Atleta[] atleti; //-> null
    static Atleta migliore; //-> null
}
```

8.3 Il tipo di un campo può essere a sua volta un *record*?

SI. Il campo di un *record* è una variabile a tutti gli effetti e dunque può essere di tipo qualsiasi.

8.4 Qual è il valore predefinito del campo di un *record*?

Vale la stessa regola applicata alle variabili globali (*static*): dipende dal tipo del campo:

```
class Atleta
{
    public string Nominativo; //-> null
    public double Punteggio;  //-> 0
}
```

8.5 I campi di un *record* possono essere *statici*?

In teoria **SI**, in pratica **NO**. I campi non devono essere *statici*, altrimenti non sarebbero in grado di assolvere alla loro funzione[3]. D'altra parte, il linguaggio non impone vincoli nella dichiarazione; è dunque responsabilità del programmatore evitare questo errore.

3 La spiegazione del perché va oltre lo scopo del *tutorial*.

Segue un esempio dove, erroneamente, dichiaro un campo statico:

```
class Atleta
{
    public string Nominativo;
    public static double Punteggio;  // formalmente corretto, ma sostanzialmente errato!
}
```

...

```
static void Main(string[] args)
{

    Atleta a = new Atleta();
    a.Nominativo = "Kent, Clark";
    a.Punteggio = 200;  // Errore! (il campo non compare nemmeno nell'intellisense!)
}
```

Di fatto: il campo `Punteggio` non appartiene alla variabile `a`.

8.6 I campi di un *record* devono essere *pubblici*?

In teoria **NO**, in pratica **SI**. I campi devono essere *pubblici*, altrimenti non sarebbero accessibili al codice. D'altra parte, il linguaggio non impone vincoli nella dichiarazione; è dunque responsabilità del programmatore evitare questo errore.

Ad esempio:

```
class Atleta
{
    public string Nominativo;
    double Punteggio;  // formalmente corretto, ma sostanzialmente errato!
}
```

...

```
static void Main(string[] args)
{

    Atleta a = new Atleta();
    a.Nominativo = "Kent, Clark";
    a.Punteggio = 200;  // Errore! (il campo non compare nemmeno nell'intellisense!)
}
```

Rispetto al punto 8.5, in questo caso il campo `Punteggio` è parte della variabile `a`, ma resta inaccessibile.

8.7 Un record può essere restituito da un metodo?

SI. Un metodo può restituire un valore di tipo qualsiasi e dunque anche un *record*.

8.8 Il nome di una variabile può essere uguale al nome del *record* che ne definisce il tipo?

SI. Si tratta di uno stile applicabile in generale, purché non si usino le parole chiave: `int`, `string`, `double`, etc.

```
static Navicella Navicella;         // OK
static Bersaglio Bersaglio;         // OK
static Proiettile Proiettile;       // OK
static string string;               // errore!
```

Poiché le variabili globali dovrebbero iniziare con la minuscola, questa regola è particolarmente utile nel dare i nomi a campi di un *record*:

```
class Calciatore
{
    public string Nominativo;
    public Squadra Squadra;    // OK
    ...
}
```

Il linguaggio non fa confusione al riguardo, perché è il contesto dell'istruzione a stabilire se un nome rappresenta un tipo o una variabile.

8.9 Modificare i campi di un parametro *record* si riflette sull'argomento corrispondente?

SI. Nel passare un argomento *record* a un metodo viene copiato il riferimento all'oggetto contenente i campi, non l'oggetto stesso. Dunque: qualsiasi modifica fatta mediante il parametro viene fatta all'oggetto originale.

Lo stesso non accade per i parametri di tipo valore: `int`, `double`, etc.