TimeRange

Case study sull'implementazione di un *valore*

Compatibilità: dotNET 2.0+

Anno 2016/2017

Indice generale

1 Introduzione

In questo tutorial utilizzo un semplice problema di programmazione come mezzo per affrontare le problematiche connesse all'implementazione di un nuovo tipo di dato. L'obiettivo finale è quello di realizzare un tipo analogo a `TimeSpan`, il quale incapsula un intervallo di tempo.

A partire dalla definizione del problema, procederò per gradi, utilizzando dapprima un approccio procedurale, per arrivare progressivamente a un corretta soluzione OO. Metterò in evidenza i limiti delle soluzioni adottate di volta in volta, mostrando come superarli e dimostrando che l'implementazione di `TimeSpan` (o tipo analogo) rappresenta il risultato ideale.

(Nota a margine: non realizzerò un'implementazione identica a quella di `TimeSpan`, ma una molto più semplice, che non prende in considerazione le gestione dell'*overflow* di valori interi)

1.1 Problema: maratona olimpica

Si desiderano elaborare i tempi dei partecipanti dalla gara della maratona olimpica. Questi sono memorizzati in un file CSV:

```
Tadesse Abraham, Svizzera, 2:11:42
Ghirmay Ghebreslassie, Eritrea, 2:11:4
Feyisa Lilesa, Etiopia, 2:09:54
Eliud Kipchoge, Kenya, 2:8:44
Munyo Mutai, Uganda, 2:11:49
Galen Rupp, USA, 2:10:05
Alphonce Simbu, Tanzania, 2:11:15
Jared Ward, USA, 2:11:30
...
```

Occorre eseguire le seguenti operazioni:

1. Visualizzare l'elenco in ordine di classifica (in base ai tempi).
2. Visualizzare la differenza tra il tempo migliore e quello peggiore.

Vincoli

Non è possibile utilizzare il tipo `TimeSpan`.

(Nota bene: per semplicità non si fa riferimento ai requisiti relativi alla UI, alla posizione del file e alle azioni da intraprendere nel caso in cui il contenuto del file non sia valido.)

2 Un approccio procedurale

Davanti a un problema simile la prima questione è: come rappresentare i dati. Il modello scelto deve avere la funzione di favorire l'implementazione delle richieste del problema: visualizzazione delle informazioni sugli atleti ed elaborazioni sulla base dei loro tempi.

Di seguito fornisco alcune opzioni, delle quali le prime due sono volutamente inadeguate.

2.1 Opzione 1: nessun modello

È la scelta più semplice: ogni atleta è rappresentato da una stringa che memorizza la riga corrispondente del file.

```
class Program
{
    static void Main(string[] args)
    {
        string[] atleti = CaricaAtleti("Maratona.txt");
        //... elabora i dati
    }

    static string[] CaricaAtleti(string nomeFile)
    {
        return File.ReadAllLines(nomeFile);
    }
}
```

Il codice di caricamento è semplicissimo; d'altra parte, avere le informazioni nella forma:

"`Tadesse Abraham`, `Svizzera`, `2:11:42`"

rende complicata l'implementazione dei requisiti, poiché implica l'estrazione dei dati (nominativo, nazione e tempo) per ogni operazione da realizzare.

2.2 Opzione 2: utilizzare dei campi stringa

Ogni atleta è rappresentato da tre campi stringa; i dati degli atleti sono memorizzati in una matrice di tre colonne.

```
static string[,] CaricaAtleti(string nomeFile)
{
    string[] righe = File.ReadAllLines(nomeFile);
    string[,] atleti = new string[righe.Length, 3];
    for (int i = 0; i < righe.Length; i++)
    {
        string[] campi = righe[i].Split(',');
        atleti[i, 0] = campi[0].Trim();
        atleti[i, 1] = campi[1].Trim();
        atleti[i, 2] = campi[2].Trim();
    }
    return atleti;
}
```

I dati di un atleta sono memorizzati separatamente, ma la loro elaborazione non risulta molto più semplice rispetto a prima: ogni dato è individuato attraverso una coppia di coordinate nella matrice. Inoltre, i tempi sono memorizzati come stringhe e dunque non è possibile utilizzarli direttamente nelle operazioni necessarie a soddisfare i requisiti del problema.

2.3 Opzione 3: *record*

L'uso di un *record* consente di rappresentare il singolo atleta in modo significativo:

```
class Atleta
{
    public string Nominativo;
    public string Nazione;
    public string Tempo;
}
```

```
static Atleta[] CaricaAtleti(string nomeFile)
{
    string[] righe = File.ReadAllLines(nomeFile);
    Atleta[] atleti = new Atleta[righe.Length];
    for (int i = 0; i < righe.Length; i++)
    {
        string[] campi = righe[i].Split(',');
        atleti[i].Nominativo = campi[0].Trim();
        atleti[i].Nazione = campi[1].Trim();
        atleti[i].Tempo = campi[2].Trim();
    }
    return atleti;
}
```

Il tipo `Atleta` incapsula il concetto di *atleta* e consente di accedere facilmente ai suoi attributi: `Nominativo`, `Nazione` e `Tempo`. Per quest'ultimo, però, è tuttora utilizzato il tipo `string`; ciò non consente di eseguire direttamente operazioni sui tempi. Ad esempio, il tempo di:

`"Tadesse Abraham, Svizzera, 2:11:42"`

è maggiore del tempo di:

`"Eliud Kipchoge, Kenya, 2:8:44"`

ma così non risulterebbe da un confronto tra stringhe, poiché la prima è minore della seconda.

Le opzioni considerate finora sollevano l'identica questione: *la rappresentazione dei dati mediante dei tipi adeguati è fondamentale*, poiché sono i tipi a stabilire i vincoli e le operazioni ammissibili su di essi. C# fornisce i tipi primitivi (`int`, `char`, `string`, `double`, etc), i quali rappresentano interi, caratteri, stringhe, etc. In alcune situazioni sono sufficienti, in altre è necessario definire nuovi tipi:

- perché non esiste un tipo in grado di rappresentare il dato (vedi, ad esempio, i singoli atleti);
- perché i tipi utilizzabili non forniscono le operazione richieste (vedi i tempi degli atleti).

2.4 Rappresentare il tempo degli atleti: intervallo di tempo

Elaborare il tempo degli atleti significa poter conoscere ore, minuti e secondi, confrontare due tempi, ottenere l'intervallo che intercorre tra due tempi. Un approccio è quello di rappresentare un singolo tempo mediante un *record*:

```
class TimeRange
{
    public int Ore;
    public int Minuti;
    public int Secondi;
}
...
class Atleta
{
    public string Nominativo;
    public string Nazione;
    public TimeRange Tempo;
}
```

2.4.1 Implementare le operazioni sui tempi

`TimeRange` si limita a memorizzare i dati; al codice applicativo resta la responsabilità di implementare le operazioni necessarie:

```
static TimeRange Parse(string tempo)  // ottiene un tempo da una stringa
{
    string[] campi = tempo.Split(':');
    var t = new TimeRange();
    t.Ore = int.Parse(campi[0]);
    t.Minuti = int.Parse(campi[1]);
    t.Secondi = int.Parse(campi[2]);
    return tr;
}

static string ToString(TimeRange t)  // converte un tempo in stringa
{
    string segno = (t.Ore < 0 || t.Minuti < 0 || t.Secondi < 0) ? "-" : "";
    return string.Format("{0}{1}:{2}:{3}", segno,
             Math.Abs(t.Ore), Math.Abs(t.Minuti), Math.Abs(t.Secondi));
}

static int TempoTotale(TimeRange t)   // ottiene i secondi totali del tempo
{
    return t.Ore * 3600 + t.Minuti * 60 + t.Secondi;
}

static TimeRange Intervallo(TimeRange t1, TimeRange t2) // ottiene l'intervallo di tempo
{                                                       // che intercorre tra due tempi
    var tr = new TimeRange();
    int scarto = TempoTotale(t1) - TempoTotale(t2);
    tr.Ore = scarto / 3600 % 3600;
    tr.Minuti = scarto / 60 % 60;
```

```
    tr.Secondi = scarto % 60;
    return tr;
}

static int Confronta(TimeRange t1, TimeRange t2) // confronta due tempi e restituisce:
{                                                 // -1, 0, 1 in base al fatto che il primo
    int sect1 = TempoTotale(t1);                 // sia minore, uguale, maggiore del secondo
    int sect2 = TempoTotale(t2);

    if (sect1 > sect2)
        return 1;
    if (sect1 < sect2)
        return -1;
    return 0;
}
```

2.5 Implementazione dei requisiti

Sulla scorta dei metodi precedenti, si possono implementare i requisiti:

```
static void Main(string[] args)
{
    Atleta[] atleti = CaricaAtleti("Maratona.txt");
    Ordina(atleti);
    Visualizza(atleti);
    Atleta primo = atleti[0];
    Atleta ultimo = atleti[atleti.Length - 1];
    TimeRange scarto = Intervallo(ultimo.Tempo, primo.Tempo);
    Console.WriteLine("\nTempo fra ultimo e primo = {0}", ToString(scarto));
}

static Atleta[] CaricaAtleti(string nomeFile)
{
    string[] righe = File.ReadAllLines(nomeFile);
    Atleta[] atleti = new Atleta[righe.Length];
    for (int i = 0; i < righe.Length; i++)
    {
        string[] campi = righe[i].Split(',');
        atleti[i] = new Atleta();
        atleti[i].Nominativo = campi[0].Trim();
        atleti[i].Nazione = campi[1].Trim();
        atleti[i].Tempo = Parse(campi[2]);
    }
    return atleti;
}


static void Visualizza(Atleta[] atleti)
{
    Console.WriteLine("{0,-25} {1,-15} {2}\n", "Nominativo", "Nazione", "Tempo (h/m/s)");
    foreach (var a in atleti)
    {
```

```
        Console.WriteLine("{0,-25} {1,-15} {2}", a.Nominativo, a.Nazione,
                                                 ToString(a.Tempo));
    }
}
```

```
static void Ordina(Atleta[] atleti)
{
    for (int i = 0; i < atleti.Length-1; i++)
    {
        for (int j = i+1; j < atleti.Length; j++)
        {
            if (Confronta(atleti[i].Tempo, atleti[j].Tempo) > 0)
            {
                var temp = atleti[i];
                atleti[i] = atleti[j];
                atleti[j] = temp;
            }
        }
    }
}
```

2.6 Conclusioni

I metodi `Parse()`, `ToString()`, `Confronta()`, `Intervallo()` forniscono un supporto all'elaborazione dei tempi degli atleti. Si tratta di una soluzione funzionale, ma non ottimale. Infatti:

1. Non consente di processare i tempi come accade con gli altri valori. Vorremmo poter scrivere:

   ```
   if (atleti[i].Tempo > atleti[j].Tempo)
   ```

   Oppure:

   ```
   TimeRange scarto = ultimo - primo;
   ```

   Ma non possiamo farlo, poiché `TimeRange` è un semplice *record* e non definisce alcuna operazione.

2. Si tratta di una soluzione difficile da riutilizzare in altri scenari; occorre copiare `TimeRange` e i metodi che lo elaborano nei nuovi progetti. Non esiste, cioè, un componente da poter facilmente riutilizzare.

3. Si tratta di una soluzione funzionale al problema corrente, ma probabilmente insufficiente in altri scenari, nei quali potrebbe essere necessario considerare anche i giorni, oppure sommare i tempi. In questo caso saremmo costretti ad aggiungere le operazioni necessarie, col risultato di avere soluzioni simili, ma non identiche, al problema di rappresentare lo stesso concetto: un intervallo di tempo.

3 Un approccio object oriented

La programmazione *object oriented* presuppone che i tipi incorporino sia dati che le operazioni da eseguire su di essi. L'obiettivo è realizzare un componente (l'*oggetto*) che definisca tutto ciò che serve a implementare il concetto "intervallo di tempo". Ciò consentirebbe di:

- semplificare la realizzazione di qualunque codice debba gestire dei tempi;
- riutilizzare lo stesso concetto in applicazioni diverse senza dover scrivere del codice ad hoc.

3.1 *Oggetto* TimeRange: dati + operazioni

Se ci si limita a considerare il "mantra" della OOP, *combinare dati e operazioni nello stesso contenitore*, la trasformazione da *record* a *oggetto* è semplice: basta incorporare i metodi di elaborazione dei tempi nel tipo `TimeRange`:

```
class TimeRange
{
    public int Ore;
    public int Minuti;
    public int Secondi;

    //static TimeRange Parse (string tempo) {...}
    public void Parse(string tempo)
    {
        string[] campi = tempo.Split(':');
        Ore = int.Parse(campi[0]);
        Minuti = int.Parse(campi[1]);
        Secondi = int.Parse(campi[2]);
    }

    //static int Confronta(TimeRange t1, TimeRange t2) {...}
    public int Confronta(TimeRange t)
    {
        int sect1 = TempoTotale(this);  //-> calcola i secondi di "se stesso"
        int sect2 = TempoTotale(t);

        if (sect1 > sect2)
            return 1;
        if (sect1 < sect2)
            return -1;
        return 0;
    }

    //static TimeRange Intervallo(TimeRange t1, TimeRange t2) {...}
    public TimeRange Intervallo(TimeRange t)
    {
        int scarto = TempoTotale(this) - TempoTotale(t);
        var tr = new TimeRange();
        tr.Ore = scarto / 3600 % 3600;
        tr.Minuti = scarto / 60 % 60;
```

```
        tr.Secondi = scarto % 60;
        return tr;
    }

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

    public static int TempoTotale(TimeRange t)  // invariato rispetto a prima
    {
        return t.Ore * 3600 + t.Minuti * 60 + t.Secondi;
    }
}
```

Segue il codice applicativo, dopo le modifiche:

```
static void Main(string[] args)
{
    Atleta[] atleti = CaricaAtleti3("Maratona.txt");
    Ordina(atleti);
    Visualizza(atleti);
    var primo = atleti[0];
    var ultimo = atleti[atleti.Length - 1];

    var scarto = Intervallo(ultimo.Tempo, primo.Tempo);
    var scarto = ultimo.Tempo.Intervallo(primo.Tempo);

    Console.WriteLine("\nTempo fra primo e ultimo = {0}", ToString(scarto));
    Console.WriteLine("\nTempo fra primo e ultimo = {0}", scarto.ToString());
}

static Atleta[] CaricaAtleti(string nomeFile)
{
    string[] righe = File.ReadAllLines(nomeFile);
    Atleta[] atleti = new Atleta[righe.Length];
    for (int i = 0; i < righe.Length; i++)
    {
        string[] campi = righe[i].Split(',');
        atleti[i] = new Atleta();
        atleti[i].Nominativo = campi[0].Trim();
        atleti[i].Nazione = campi[1].Trim();
        atleti[i].Tempo = Parse(campi[2]);
        var tr = new TimeRange();
        tr.Parse(campi[2]);
        atleti[i].Tempo = tr;
    }
    return atleti;
}
```

```
static void Ordina(Atleta[] atleti)
{
    for (int i = 0; i < atleti.Length-1; i++)
    {
        for (int j = i+1; j < atleti.Length; j++)
        {
            if (Confronta(atleti[i].Tempo, atleti[j].Tempo) > 0)
            if (atleti[i].Tempo.Confronta(atleti[j].Tempo) > 0)
            {
                var temp = atleti[i];
                atleti[i] = atleti[j];
                atleti[j] = temp;
            }
        }
    }
}
```

3.1.1 Analisi della classe: "funzionamento" del codice object oriented

I metodi `Parse()`, `Confronta()`, `Intervallo()` e `ToString()` sono, nella sostanza, rimasti uguali agli originali; ma sono diversi nella forma e questo ne spiega la diversa modalità di impiego.

Consideriamo `Intervallo()` nelle due versioni:

```
static TimeRange Intervallo(TimeRange t1, TimeRange t2)
{
    int scarto = TempoTotale(t1) – TempoTotale(t2);
    ...
}
```

```
TimeRange Intervallo(TimeRange t)
{
    int scarto = TempoTotale(this) - TempoTotale(t);
    ...
}
```

E nelle due chiamate:

```
...
var scarto = Intervallo(ultimo.Tempo, primo.Tempo);
...
```

```
...
var scarto = ultimo.Tempo.Intervallo(primo.Tempo);
...
```

In pratica, si tratta di "manquillage" sintattico; il compilatore traduce automaticamente la seconda versione (quella *object oriented*) nella prima, operando le seguenti aggiunte (in grigio):

```
static TimeRange Intervallo(TimeRange this, TimeRange t)
{
    int scarto = TempoTotale(this) - TempoTotale(t);
    ...
}
```

Il compilatore introduce un parametro nascosto nella firma del metodo; questo è accessibile mediante la parola chiave `this` e memorizza l'oggetto attraverso il quale viene invocato il metodo. Coerentemente, il compilatore traduce anche l'istruzione di chiamata, passando come primo argomento l'oggetto attraverso il quale viene invocato il metodo:

```
...
var scarto = ultimo.Tempo.Intervallo(ultimo.Tempo, primo.Tempo); // ultimo.Tempo -> this
...
```

Accesso ai dati dell'oggetto

Si consideri il metodo `Parse()`, a sinistra come è scritto, a destra come viene tradotto dal compilatore (segue un esempio di chiamata del metodo):

Come è scritto dal programmatore

```
void Parse(string tempo)
{
    string[] campi = tempo.Split(':');
    Ore = int.Parse(campi[0]);
    Minuti = int.Parse(campi[1]);
    Secondi = int.Parse(campi[2]);
}
```

```
var tr = new TimeRange();
tr.Parse("2:12:4");
```

Traduzione del compilatore

```
static void Parse(TimeRange this, string tempo)
{
    string[] campi = tempo.Split(':');
    this.Ore = int.Parse(campi[0]);
    this.Minuti = int.Parse(campi[1]);
    this.Secondi = int.Parse(campi[2]);
}
```

```
var tr = new TimeRange();
Parse(tr, "2:12:4");
```

Nel metodo è possibile utilizzare direttamente i campi e gli altri metodi dell'oggetto; il compilatore aggiunge automaticamente il riferimento al parametro nascosto che rappresenta l'oggetto. Se lo si desidera, mediante `this` è possibile referenziare esplicitamente l'oggetto (e dunque accedere esplicitamente al parametro nascosto):

```
public void Parse(string tempo)
{
    string[] campi = tempo.Split(':');
    this.Ore = int.Parse(campi[0]);
    ...
}
```

È ciò che viene fatto in `Intervallo()`, per calcolare i secondi totali dell'oggetto e sottrarli ai secondi totali del parametro `t`:

```
public TimeRange Intervallo(TimeRange t)
{
    int scarto = TempoTotale(this) - TempoTotale(t);
    ...
}
```

3.2 Conclusioni

Aver spostato la logica di elaborazione dei tempi dentro `TimeRange` ha tolto questa responsabilità al codice applicativo, anche se non ha cambiato significativamente i metodi `Main()`, `Ordina()`, `CaricaAtleti()` e `Visualizza()`. È stato comunque raggiunto un importante risultato: la classe `TimeRange` incorpora tutto ciò che serve a gestire un intervallo di tempo e può essere facilmente riutilizzata in altri scenari.

4 Adottare i principi di progettazione OO

Il tipo `TimeRange` è funzionale ai requisiti dell'applicazione nella quale è stato realizzato, ma se vogliamo renderlo utilizzabile in un ampia gamma di scenari dobbiamo adottare criteri di progettazione più stringenti. Vi sono diverse aree nelle quali intervenire:

- **Validità**: occorre garantire che gli oggetti abbiano sempre dei valori coerenti e che le loro operazioni restituiscano sempre dei valori validi.
- **Efficienza**: le *perfomance*, occupazione di memoria e velocità di esecuzione delle operazioni, sono importanti.
- **Incapsulamento**: occorre separare la funzione di `TimeRange` dal modo in cui questa funzione è implementata.
- **Usabilità**: un intervallo di tempo è essenzialmente un *valore*, come un intero o una stringa; ci si aspetta di poterlo manipolare allo stesso modo degli altri tipi primitivi.

Di seguito modificherò progressivamente il progetto di `TimeRange` in modo che aderisca a questi criteri.

4.1 Stabilire la modalità di creazione dei tempi

Attualmente, il meccanismo di creazione di un nuovo tempo si svolge in due fasi:

```
TimeRange t = new TimeRange();      // crea l'oggetto
t.Parse("1:2:3");                   // imposta il suo valore (1 ora, 2 minuti, 3 secondi)
```

È opportuno dare la possibilità di specificare il valore durante la creazione dell'oggetto. A questo scopo definisco due costruttori; il primo sostituisce il metodo `Parse()`, il secondo consente di specificare l'intervallo direttamente in ore, minuti e secondi:

```
class TimeRange
{
    ...
    public TimeRange(string tempo)
    {
        string[] campi = tempo.Split(':');
        Ore = int.Parse(campi[0]);
        Minuti = int.Parse(campi[1]);
        Secondi = int.Parse(campi[2]);
    }
    public TimeRange(int ore, int minuti, int secondi)
    {
        Ore = ore;
        Minuti = minuti;
        Secondi = secondi;
    }
    ...
    public TimeRange Intervallo(TimeRange t)
    {
        int scarto = TempoTotale(this) - TempoTotale(t);
        return new TimeRange(scarto / 3600 % 3600, scarto / 60 % 60, scarto % 60);
```

```
    }
}
```

Nota bene: non è più possibile creare un tempo senza specificarne il valore; per questo motivo ho dovuto modificare il metodo `Intervallo()`, che ora usa uno dei nuovi costruttori, calcolando ore, minuti e secondi sulla base del tempo totale in secondi. Per il codice applicativo cambia poco:

```
static Atleta[] CaricaAtleti(string nomeFile)
{
        ...
        var tr = new TimeRange();
        tr.Parse(campi[2]);
        atleti[i].Tempo = tr;

        atleti[i].Tempo = new TimeRange(campi[2]);
        ...
}
```

4.2 Rappresentare un intervallo di tempo in secondi

`TimeRange` memorizza il tempo in ore, minuti e secondi, ma lo elabora calcolando ogni volta il tempo totale corrispondente (metodo `TempoTotale()`). Tanto vale calcolare subito questo dato:

```
class TimeRange
{
    ...
    public int TempoTotale;
    public TimeRange(string tempo)
    {
        ...
        TempoTotale = Ore * 3600 + Minuti * 60 + Secondi;
    }

    public TimeRange(int ore, int minuti, int secondi)
    {
        ...
        TempoTotale = Ore * 3600 + Minuti * 60 + Secondi;
    }

    public TimeRange(int tempoTotale)
    {
        TempoTotale =  tempoTotale
    }

    public int Confronta(TimeRange t)
    {
        if (TempoTotale > t.TempoTotale)
            return 1;
        if (TempoTotale < t.TempoTotale)
            return -1;
        return 0;
    }
```

```
    public TimeRange Intervallo(TimeRange t)
    {
        return new TimeRange(TempoTotale - t.TempoTotale);
        return new TimeRange(scarto / 3600 % 3600, scarto / 60 % 60, scarto % 60);
    }

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

    public static int TempoTotale(TimeRange t)
    {
        return t.Ore * 3600 + t.Minuti * 60 + t.Secondi;
    }
}
```

Nota bene: ho aggiunto un nuovo costruttore, che accetta direttamente il tempo totale. Questo, oltre a fornire un'ulteriore modalità di creazione di un tempo, semplifica il metodo `Intervallo()`.

Apparentemente, l'introduzione del nuovo campo non ha alcun impatto sul codice applicativo, ma non è così: la possibilità di modificare i campi `Ore`, `Minuti` e `Secondi` produce un comportamento incoerente dell'oggetto.

4.3 Ottimizzare e rendere coerente l'implementazione

Si consideri il seguente codice:

```
var t1 = new TimeRange(1, 2, 3);    //-> 1:2:3
var t2 = new TimeRange(2, 2, 3);    //-> 2:2:3

t1.Ore = 2;                         //-> t1 diventa: 2:2:3
Console.WriteLine(t1.ToString());   //-> 2:2:3  (sembra ok)

var t3 = t2.Intervallo(t1);         //-> (t2-t1) mi aspetto che t3 sia: 0:0:0
Console.WriteLine(t3.ToString());   //-> invece è: 1:0:0
```

L'attuale implementazione di `TimeRange` presenta dei problemi:

- Lo *stato interno* (i dati memorizzati da `TimeRange`) non viene gestito in modo coerente. Il tempo è memorizzato sia in `Ore`, `Minuti` e `Secondi`, che in `TempoTotale`. Ciò provoca un comportamento inconsistente: il metodo `ToString()` usa `Ore`, `Minuti` e `Secondi`, mentre `Intervallo()` e `Confronta()` usano `TempoTotale`.
- È ridondante e, dunque, inefficiente. Per memorizzare un singolo dato (il tempo) vengono utilizzate quattro variabili, per un totale di 16 byte.

4.3.1 Definire in modo chiaro lo stato dell'oggetto

Un (intervallo di) tempo viene elaborato attraverso un solo valore: il tempo totale trascorso. Ore, minuti e secondi sono calcolabili a partire da esso e dunque possono essere implementati

mediante proprietà *getonly*:

```
class TimeRange
{
    public int TempoTotale;
    public TimeRange(string tempo)
    {
        string[] campi = tempo.Split(':');
        int ore = int.Parse(campi[0]);
        int minuti = int.Parse(campi[1]);
        int secondi = int.Parse(campi[2]);
        TempoTotale = ore * 3600 + minuti * 60 + secondi;
    }
    public TimeRange(int ore, int minuti, int secondi)
    {
        TempoTotale = ore * 3600 + minuti * 60 + secondi;
    }

    public TimeRange(int tempoTotale)
    {
        this.TempoTotale = tempoTotale;
    }

    public int Ore
    {
        get { return TempoTotale / 3600 % 3600; }
    }

    public int Minuti
    {
        get { return TempoTotale / 60 % 60; }
    }

    public int Secondi
    {
        get { return TempoTotale % 60; }
    }
}
```

Adesso il comportamento dell'oggetto dipende dal solo campo `TempoTotale` e dunque è sempre coerente. Contestualmente, il codice applicativo non può più modificare i singoli componenti del tempo. Dal punto di vista dell'efficienza c'è un calo di performance nell'accesso a `Ore`, `Minuti`, `Secondi` e `ToString()`. I metodi `Intervallo()` e `Confronta()` mantengono le stessa velocità di esecuzione, mentre l'occupazione di memoria si è ridotta a un quarto ed è determinata dal solo campo `TempoTotale`.

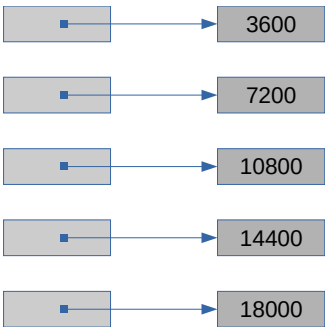
4.3.2 Utilizzare una struttura

Poiché un intervallo di tempo ha le dimensioni di un intero, rappresentarlo mediante una classe non è efficiente. Si consideri il seguente codice, che crea un vettore di 5 tempi:

```
TimeRange[] tempi = new TimeRange[5];
for (int i = 0; i < tempi.Length; i++)
```

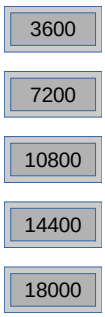
```
{
    tempi[i] = new TimeRange(i, 0, 0);
}
```

Segue una rappresentazione dell'occupazione di memoria del vettore:



Ogni elemento del vettore non contiene un tempo ma un riferimento ad esso. Ciò, non solo aumenta la quantità di memoria necessaria, ma mette sotto pressione il sistema di gestione della memoria (*garbage collector*), il quale deve gestire 6 oggetti distinti (il vettore più i cinque `TimeRange`), invece del solo vettore, come ci si aspetterebbe[1].

Definire `TimeRange` come struttura consente una più efficiente gestione della memoria. Nell'esempio precedente viene allocato in memoria il solo vettore, che occupa 20 byte soltanto:



Quanto detto per `TimeRange` vale in generale per qualsiasi tipo di dato rappresenti un oggetto di piccole dimensioni. `char`, `int`, `float`, `double`, `DateTime`, `TimeSpan`, `Point`, `Size`, `Rectangle`, per citare alcuni tipi, sono tutti implementati mediante strutture.

Tempo predefinito

L'uso di una `struct` implica automaticamente il concetto di *valore predefinito*, e cioè il valore di una variabile all'atto della sua dichiarazione. Per qualsiasi tipo di dato, questo corrisponde a un oggetto con i campi azzerati. Il seguente codice mostra tre tempi che memorizzano il valore predefinito:

```
static TimeRange t1;   //->0:0:0
static void Main(string[] args)
{
    TimeRange t2 = default(TimeRange);  //->0:0:0
    TimeRange t3 = new TimeRange();     //->0:0:0
    ...
}
```

1 L'uso di una classe implica un ulteriore allocazione di memoria per dei campi nascosti gestiti interamente.

4.4 Incapsulare completamente TimeRange

`TimeRange` incapsula il concetto di intervallo di tempo, che non significa soltanto rappresentare un tempo (come fa `Atleta` con gli atleti), ma anche fornire un insieme di operazioni che siano utilizzabili indipendentemente dalla sua implementazione. Ciò è stato dimostrato in 4.2 e 4.3: la modalità di memorizzazione del tempo è stata modificata senza produrre un impatto sull'uso degli oggetti. Il termine *incapsulamento* assume dunque un significato stringente: creare una "barriera" tra le operazioni di un oggetto e la sua implementazione; le prime sono utilizzabili senza che sia necessario conoscere la seconda.

Ebbene, `TimeRange` non è completamente incapsulato, poiché il campo `TempoTotale` è accessibile per il codice esterno.

4.4.1 "Proteggere" lo stato dell'oggetto

Esiste una concezione diffusa sull'incapsulamento, che lo vede come strumento per "nascondere" l'implementazione allo scopo di "proteggerla" da un uso non valido. Ad esempio, la classe `List<>` definisce il campo privato `_size`, che memorizza il numero degli elementi presenti nella lista ed è restituito dalla proprietà pubblica `Count`. Se `_size` fosse pubblico, il codice esterno potrebbe modificarne direttamente il valore, con il rischio di compromettere il funzionamento della lista. In questo caso, l'idea di incapsulare `_size` (e qualsiasi altro campo di `List<>`) appare una decisione scontata, ma che dire di `TimeRange` e del suo unico campo `TempoTotale`? Qualunque valore assegnato a `TempoTotale` è valido per definizione, e poiché tutto il resto si basa unicamente su questo campo, non esiste il pericolo di compromettere il funzionamento dell'oggetto. Dunque, si potrebbe pensare che definire `TempoTotale` pubblico non violi affatto il principio di incapsulamento. Ma non è così.

4.4.2 Incapsulamento: non dipendere dall'implementazione

Incapsulare un oggetto non serve (soltanto) a proteggerne lo stato interno, ma a evitare che il codice esterno possa dipendere dalla sua implementazione. Se questo accade si crea un accoppiamento tra il codice che usa l'oggetto e i dettagli che determinano il suo funzionamento; dopodiché, l'eventuale modifica dell'implementazione dell'oggetto richiederà una modifica del codice che lo usa o, ancora peggio, ne cambierà il funzionamento in modo imprevedibile.

Ad esempio, ipotizziamo un'applicazione che debba ottenere il numero totale di secondi che intercorrono tra due tempi. Un approccio potrebbe essere il seguente:

```
TimeRange tStart = StartClock();
...
TimeRange tStop = StopClock();
TimeRange tInterval = tStart.Intervallo(tStop);
int elapsedSeconds = tInterval.Ore * 3600 + tInterval.Minuti * 60 + tInterval.Secondi;
...
```

Ma, conoscendo l'implementazione di `TimeRange`, si può optare per un'alternativa più semplice ed efficiente:

```
TimeRange tStart = StartClock();
...
TimeRange tStop = StopClock();
TimeRange tInterval = tStart.Intervallo(tStop);
int elapsedSeconds = tInterval.TempoTotale;
```

```
...
```

Però, mentre la prima opzione è valida per definizione, la seconda si basa sull'assunzione che `TempoTotale` memorizzi il numero totale di secondi. Si tratta di assunzione che dipende dall'implementazione e potrebbe non essere più valida se questa venisse cambiata, ad esempio aggiungendo la gestione dei millisecondi. In quest'ultimo caso, `TempoTotale` aumenterebbe di un fattore 1000 e tutto il codice applicativo che si basa sull'assunzione suddetta smetterebbe di funzionare correttamente. Rendendo `TempoTotale` inaccessibile si evita che il codice esterno dipenda da esso:

```
struct TimeRange
{
    private int tempoTotale;
    ...
}
```

Evitare dipendenze attraverso una corretta documentazione delle operazioni

Esiste ancora una sottile dipendenza collegata a `tempoTotale` ed è rappresentata dal costruttore:

```
struct TimeRange
{
    private int tempoTotale;

    public TimeRange(int tempoTotale)
    {
        this.tempoTotale = tempoTotale;
    }
    ...
}
```

Questo è utile per creare un nuovo oggetto a partire da un altro o dal risultato di un'operazione; resta il fatto che, essendo pubblico, può essere impiegato dal codice esterno. Ad esempio, il seguente codice:

```
var t = new TimeRange(100); // -> 0:1:40
```

si basa sull'assunto che il valore specificato rappresenti il numero totale di secondi. Ancora una volta, se pure indirettamente, si lega l'uso dell'oggetto alla sua implementazione. Esistono due soluzioni:

1. Rendere privato il costruttore, relegandolo a un ruolo di "servizio".
2. Stabilire esplicitamente la natura del parametro.

Nel secondo caso occorre modificare il nome del parametro:

```
struct TimeRange
{
    private int tempoTotale;

    public TimeRange(int secondiTotali)
    {
        this.tempoTotale = secondiTotali;
    }
    ...
```

```
}
```

Nota bene: non si tratta semplicemente di cambiare un nome, ma di mettere un vincolo sul costruttore: creare un tempo sulla base dei secondi totali. Se in futuro dovessimo decidere di gestire anche i millisecondi, il costruttore dovrebbe essere modificato di conseguenza:

```
struct TimeRange
{
    private int tempoTotale;

    public TimeRange(int secondiTotali)
    {
        this.tempoTotale = secondiTotali * 1000;
    }
    ...
}
```

Detto questo, stabilisco di rendere il costruttore privato.[2]

2 In **TimeSpan** esiste un costruttore analogo che è pubblico.

5 “Standardizzare” TimeRange

Dopo le precedenti modifiche, il tipo `TimeRange` è efficiente e ben incapsulato; resta il fatto che è nato per soddisfare i requisiti di una specifica applicazione. Se vogliamo renderlo utilizzabile ovunque sia necessario elaborare il concetto di (intervallo di) tempo, è opportuno aumentare le sue funzionalità e “allineare” la sua interfaccia pubblica a quella degli altri tipi predefiniti. L'idea è che le applicazioni possano elaborare valori `TimeRange` nello stesso modo in cui elaborano interi, stringhe, etc.

5.1 Definire gli operatori standard

Quando si progetta un nuovo tipo è utile implementare tutti gli operatori coerenti con il concetto rappresentato. Per quanto riguarda `TimeRange`, sono senz'altro significativi i seguenti operatori: `==`, `!=`, `>`, `<`, `>=`, `<=`, `+` e `-`.

```
public struct TimeRange
{
    private int tempoTotale;
    ...
    public static bool operator == (TimeRange left, TimeRange right)
    {
        return left.TempoTotale == right.TempoTotale;
    }

    public static bool operator != (TimeRange left, TimeRange right)
    {
        return left.TempoTotale != right.TempoTotale;
    }

    public static bool operator > (TimeRange left, TimeRange right)
    {
        return left.TempoTotale > right.TempoTotale;
    }

    public static bool operator < (TimeRange left, TimeRange right)
    {
        return left.TempoTotale < right.TempoTotale;
    }

    public static bool operator >= (TimeRange left, TimeRange right)
    {
        return left.TempoTotale > right.TempoTotale;
    }

    public static bool operator <= (TimeRange left, TimeRange right)
    {
        return left.TempoTotale <= right.TempoTotale;
    }

    public static TimeRange operator + (TimeRange left, TimeRange right)
    {
```

```
        return new TimeRange(left.TempoTotale + right.TempoTotale);
    }

    public static TimeRange operator - (TimeRange left, TimeRange right)
    {
        return new TimeRange(left.TempoTotale - right.TempoTotale);
    }
}
```

Nota bene: coerentemente con i tipi primitivi, gli operatori `+` e `-` producono un nuovo valore, che equivale al risultato dell'operazione.

(Nota a margine: se `TimeRange` fosse stato una classe, l'implementazione degli operatori di confronto sarebbe stata più complicata.)

Dopo questa modifica diventa possibile scrivere codice come il seguente:

```
var t1 = new TimeRange(1, 2, 3); // -> 1:2:3
var t2 = new TimeRange(2, 2, 3); // -> 2:2:3
var b1 = t1 == t2;  // -> false
var b2 = t1 < t2;   // -> true
var b3 = t1 >= t2;  // false
var t3 = t1 + t2;   // -> 3:4:6
var t4 = t1 - t2;   // -> -1:0:0
```

Nel codice applicativo è possibile (ma non obbligatorio) modificare il metodo di ordinamento:

```
static void Ordina(Atleta[] atleti)
{
    for (int i = 0; i < atleti.Length-1; i++)
    {
        for (int j = i+1; j < atleti.Length; j++)
        {
            if (atleti[i].Tempo > atleti[j].Tempo)
            {
                var temp = atleti[i];
                atleti[i] = atleti[j];
                atleti[j] = temp;
            }
        }
    }
}
```

5.2 Eseguire il parsing da stringa

I tipi predefiniti sono allineati a uno standard per quanto riguarda la conversione da e per stringa. Definiscono un metodo statico `Parse()` che restituisce un nuovo valore a partire da una stringa, e un metodo di istanza `ToString()` che produce una rappresentazione stringa del valore. Inoltre, il metodo `Parse()` solleva l'eccezione `FormatException` se l'argomento stringa non memorizza un valore valido.

Compatibilmente a questo modello, definisco il metodo statico `Parse()` ed elimino il costruttore che svolge la funzione omologa:

```
public struct TimeRange
{
    private int tempoTotale;

    public TimeRange(string tempo)
    {
        ...
    }

    public static TimeRange Parse(string tempo)
    {
        try
        {
            if (tempo.Contains(' '))
                throw new Exception();

            string[] fields = tempo.Split(':');
            if (fields.Length > 3)
                throw new Exception();

            int secondi = 0;
            int minuti = 0;
            int ore = int.Parse(fields[0]);
            if (fields.Length > 0)
                minuti = int.Parse(fields[1]);
            if (fields.Length > 1)
                secondi = int.Parse(fields[2]);
            return new TimeRange(ore, minuti, secondi);
        }
        catch
        {
            throw new FormatException("TimeRange non valido: " + tempo);
        }
    }
    ...
}
```

Nel codice applicativo occorre modificare il metodo di caricamento degli atleti:

```
static Atleta[] CaricaAtleti(string nomeFile)
{
        ...
        atleti[i].Tempo = new TimeRange(campi[2]);
        atleti[i].Tempo = TimeRange.Parse(campi[2]);
    }
    return atleti;
}
```

5.3 Informazioni sul valore

`TimeRange` consente di ottenere ore, minuti e secondi di un tempo; ma esistono altre informazioni rilevanti che si possono ricavare: le ore, i minuti e i secondi totali. Questi sono valori reali, perché possono memorizzare anche frazioni di tempo. Prima di implementare queste proprietà, definisco delle costanti simboliche che memorizzano dei rapporti utili nei vari calcoli:

```
public struct TimeRange
{
    const int TEMPO_PER_ORE = 3600;
    const int TEMPO_PER_MINUTI = 60;

    private int tempoTotale;
    ...
    public double SecondiTotali
    {
        get { return tempoTotale; }
    }

    public double MinutiTotali
    {
        get { return (double)tempoTotale / TEMPO_PER_MINUTI; }
    }

    public double OreTotali
    {
        get { return (double)tempoTotale / TEMPO_PER_ORE; }
    }
    ...
}
```

Nota bene: non sarebbe necessario definire `double` anche `SecondiTotali`, ma facendolo evito di legare il codice all'implementazione.[3]

5.3.1 Informazioni sul tipo

Come gli altri tipi numerici, è utile che `TimeRange` definisca i valori massimo e minimo:

```
public struct TimeRange
{
    ...
    public static readonly TimeRange TempoMassimo = new TimeRange(int.MaxValue);
    public static readonly TimeRange TempoMinimo = new TimeRange(int.MinValue);
    ...
}
```

Nota bene: i campi sono *readonly*, altrimenti nulla impedirebbe al codice esterno di sostituire il valore originale con un altro.

3 In **TimeSpan** la proprietà analoga si chiama **Ticks** e restituisce un valore integrale corrispondente al tempo totale.

5.4 Creazione di tempi sulla base di valori totali

I seguenti metodi statici realizzano la funzione inversa delle proprietà che restituiscono il tempo totale in ore, minuti o secondi:

```
public struct TimeRange
{
    ...
    public static TimeRange PerOre(double ore)
    {
        return new TimeRange((int) (ore * TEMPO_PER_ORE));
    }

    public static TimeRange PerMinuti(double minuti)
    {
        return new TimeRange((int)(minuti * TEMPO_PER_MINUTI));
    }

    public static TimeRange PerSecondi(double secondi)
    {
        return new TimeRange((int)secondi);
    }
    ...
}
```

Nota bene: prima viene eseguito il calcolo e soltanto dopo viene applicata la conversione a intero; in caso contrario sarebbe persa l'eventuale parte frazionaria del valore specificato.

Adesso è possibile scrivere il seguente codice:

```
var t1 = TimeRange.PerOre(1.5);   // -> 1:30:0
var t2 = TimeRange.PerMinuti(90); // -> 1:30:0
var b1 = t1 == t2;                // -> true
var b2 = t1 < t2;                 // -> false
double ore = t1.OreTotali;        // -> 1.5
```

6 Considerazioni finali

Sono opportune alcune considerazioni sul risultato raggiunto.

6.1 Utilità e qualità di TimeRange

`TimeRange` è estremamente utile, poiché, attraverso una semplice interfaccia permette a qualsiasi applicazione di elaborare in modo consistente lo stesso concetto. Per quanto riguarda la sua qualità, per giudicarla non si può prescindere dal suo ambito di impiego. Per sua natura, `TimeRange` appartiene al dominio *fondazionale*, e cioè a fa parte di quei componenti (moduli e oggetti) sui quali si basa tutto il software, indipendentemente da piattaforma, SO e tipo di applicazione. Appare evidente che un componente simile potrebbe essere utilizzato in migliaia di applicazioni e referenziato in milioni di righe di codice; ne segue che, in uno scenario reale, dovrebbe essere progettato come estrema attenzione. L'attuale versione rappresenta soltanto un semplice esempio di programmazione OO e come tale presenta alcuni difetti.

6.1.1 Validità

Un corretto incapsulamento dovrebbe garantire che i valori siano sempre consistenti in relazione alle operazioni effettuate; se ciò non è possibile deve essere sollevato un errore, altrimenti viene compromesso il funzionamento del codice applicativo. `TimeRange` non offre garanzie in tal senso, come dimostra il seguente codice:

```
var t1 = new TimeRange(100000, 0, 0); //-> 1 milione di ore
var t2 = TimeRange.TempoMassimo;      //-> 2523:14:7
var t3 = new TimeRange(0, 0, 1);      //-> 1 secondo
var t4 = t2 + t3;
var t5 = TimeRange.PerSecondi(10000000000.0);  //-> 10 miliardi di secondi

Console.WriteLine(t1.ToString());  //-> -2246:28:16
Console.WriteLine(t4.ToString()); //-> -2523:14:8
Console.WriteLine(t5.ToString()); //-> -2523:14:8
```

Non vi sono controlli sull'intervallo di valori ammissibili, con il risultato che è possibile ottenere tempi imprevedibili. Vi sono varie tecniche per affrontare questo problema e `TimeSpan` le adotta.

6.1.2 Allineamento agli standard dei tipi primitivi

Al di là della nomenclatura in italiano, `TimeRange` ha un'interfaccia pubblica e un funzionamento coerenti con quella dei tipi primitivi, ma non completamente, e questo pregiudica la sua integrazione con il resto del *type system*.

Interfaccia IEquatable

Tutti i tipi per i quali è significativo il concetto di *value equality* (confronto per uguaglianza tra due valori) dovrebbero implementare l'interfaccia `IEquatable<>`. Ciò consente, ad esempio, di usarli come chiavi nei dizionari.

Interfaccia IComparable

Tutti tipi per i quali esiste un criterio di ordinamento dovrebbero implementare l'interfaccia `IComparable<>`. Ciò consente di ordinarli e di utilizzarli in algoritmi di ricerca binaria. `TimeRange`

definisce già l'operazione necessaria: il metodo `Confronta()`. Basta rinominarlo in `CompareTo()` e implementare l'interfaccia suddetta.

Parsing da stringa

L'attuale codice di *parsing* è funzionale, ma limitato. Dovrebbe essere possibile interpretare più formati. Inoltre, dovrebbe essere aggiunto anche il metodo `TryParse()`, che consente di eseguire il *parsing* senza sollevare eccezioni in caso di errore.

Interfaccia IFormattable (rappresentazione stringa), override del metodo ToString()

Analogamente al *parsing*, dovrebbe essere possibile rappresentare il tempo in vari formati; lo si può fare implementando l'interfaccia `IFormattable`, il cui metodo `ToString()` accetta una stringa di formato. Inoltre, l'attuale metodo `ToString()` (senza parametri) dovrebbe essere virtuale, in modo da sovrascrivere l'implementazione ereditata da `object`.

Entrambe le soluzioni consentono di specificare direttamente valori `TimeRange` nei metodi `Write()`, `WriteLine()` e `string.Format()`.

6.1.3 Risoluzione temporale e intervallo di variazione

`TimeRange` ha una risoluzione di un secondo e dunque non può essere impiegato in scenari che richiedono una gestione fine del tempo: programmi di *benchmark*, sistemi *realtime*, etc. In generale, una risoluzione di un millisecondo è sufficiente per la maggior parte degli scenari. (`TimeSpan` ha una risoluzione di 100 nanosecondi.) Al lato opposto, sarebbe utile poter utilizzare anche i giorni.

Aumentare la risoluzione significa automaticamente aumentare l'intervallo di variazione, in caso contrario non sarebbe possibile rappresentare valori massimi e minimi significativi per moltissime applicazioni. Per questo motivo sarebbe opportuno memorizzare il `tempoTotale` in un `long`.

6.2 "Immutabilità" di TimeRange

Gli oggetti appartenenti a un tipo immutabile non possono essere modificati dopo la creazione. Tutti i tipi primitivi sono immutabili, come lo sono `DateTime` e `TimeSpan`. La maggior parte dei tipi sono mutabili; alcuni esempi sono: `List<>`, `Stack<>`, `Dictionary<>` , `Size` e `Point`.

L'immutabilità è una caratteristica desiderabile, poiché semplifica la scrittura di codice privo di effetti collaterali, risultato particolarmente utile negli scenari *multi-threading*. Ma molti oggetti devono essere mutabili per natura, spesso per una questione di performance. Si pensi al tipo `List<>`; renderlo immutabile significherebbe far sì che ogni modifica alla lista produca una nuova lista contenente una copia degli elementi originali più la modifica; in molte situazioni si tratta di un costo inaccettabile.

Particolarmente interessanti sono quei tipi assimilabili a valori e che occupano poca memoria; in questo caso la scelta tra un'implementazione mutabile o immutabile non è affatto scontata. Si consideri il tipo `Point` (*namespace* `System.Drawing`). Incapsula il concetto di punto e memorizza le coordinate in due campi interi. È un tipo mutabile e dunque è possibile modificare le singole coordinate, come mostra il seguente codice:

```
Point p = new Point(10, 5); //-> X:10, Y:5
p.X = -10;
Console.WriteLine(p);       //-> {X=-10, y=5}
```

Se `Point` fosse immutabile, l'unico modo per modificare una sola coordinata sarebbe quello di

creare un nuovo punto, eseguendo due assegnazioni invece di una:

```
Point p = new Point(10, 5); //-> X:10, Y:5
p = new Point(-10, p.Y);  //-> crea un nuovo punto, impostando sia X che Y
...
```

Per questo motivo è stato deciso, dato l'impiego di `Point` e le esigenze di performance, di renderlo mutabile.

Che dire di `TimeRange`? Ha un solo campo, che è immutabile; non esistono dunque vantaggi in termini di performance in una implementazione mutabile, poiché qualsiasi operazione che produce un nuovo tempo implica comunque un'assegnazione al suo unico campo. Ciò detto, un'implementazione immutabile (come l'attuale) ha tutti i vantaggi:

- È più semplice da realizzare.
- È più semplice e sicura da utilizzare, soprattutto in scenari *multi-thread*.