ESPERTO IN UN CLICK

Michael Ferrari

C#

LE BASI PER TUTTI

IMPARA A PROGRAMMARE
PER IL MONDO .NET

area51
Publishing

Michael Ferrari

C#

Le basi per tutti

Curatore della collana: Mirco Baragiani Cover: Valerio Monego Redazione e sviluppo ebook: Enrico De Benedictis

Segui **Area51 Publishing** su:

 **Google Plus**

 **Facebook**

 **Twitter**

 **Pinterest**

www.area51editore.com

www.area51editore.com/blog

Introduzione

Il linguaggio C# nasce ufficialmente nel 2001 a opera dell'ingegnere danese Anders Hejlsberg, capo di un team di sviluppo Microsoft all'interno del progetto (dot).NET. Il linguaggio, orientato agli oggetti, è nato con lo scopo di prendere il meglio dai linguaggi a oggetti all'epoca in voga ovvero: C++, Java e Delphi.

Il risultato di questo lavoro ha portato alla realizzazione di un linguaggio più compatto e leggibile rispetto al C++ ma con meno elementi verbosi di sintassi rispetto al linguaggio Java.

Tutti i cardini del progetto .NET di Microsoft sono incapsulati all'interno del linguaggio C# che di fatto è nato come la piattaforma preferenziale nello sviluppo di applicazioni per il Framework .NET ed è quindi semplice e immediato trovare una correlazione puntuale tra astrazioni, classi, interfacce, delegati ed eccezioni nel linguaggio in fatto di gestione del framework.

Il linguaggio C#, a differenza dei suoi "cugini", è allo stesso tempo un linguaggio interpretato e compilato e queste due caratteristiche possono essere delegate a due distinte fasi senza avere compromessi in termini di resa finale. Durante la fase di testing la macchina virtuale presenterà tutte le caratteristiche del codice e solo in un secondo tempo si potrà decidere se compilare il codice e quindi rilasciare il software sotto forma di pacchetto nativo binario.

Caratteristiche del linguaggio

Queste sono in riassunto tutte le caratteristiche che rendono C# un linguaggio unico nel suo genere (alcune saranno ampliamente approfondite in questo ebook):

- Abbraccia vari paradigmi di programmazione, come la maggior parte dei nuovi linguaggi. Certamente l'aspetto object oriented è quello più utilizzato, ma si può usare C# anche per programmare usando l'approccio funzionale. Tutto questo grazie appunto all'evoluzione avutasi nel tempo che ha notevolmente arricchito il linguaggio di costrutti ed espressività. Anche i paradigmi riflessivo e concorrente sono supportati.
- Nasce come linguaggio a tipizzazione statica, ovvero una variabile nasce e muore rimanendo sempre dello stesso tipo. Tuttavia, a partire dalla versione 4.0, grazie all'introduzione del DLR è possibile anche dichiarare una variabile in maniera dinamica, aumentando ancora di più la duttilità del linguaggio. Nel caso in cui si scelga la tipizzazione statica essa è anche forte, ovvero sono permesse su di esse solo operazioni "legali" e sicura, quindi non ammette pericolose conversioni implicite che possono portare a perdita di informazione. Grazie alla sua nuova natura dinamica C# ammette ora anche una parziale inferenza di tipo.
- Nell'ambito della programmazione a oggetti permette solo l'ereditarietà singola da classi ma è possibile ereditare da quante interfacce si vuole. Se siete dei fan dell'ereditarietà multipla, con C# rimarrete delusi. Ovviamente questo non preclude nulla in termini di potenzialità ed espressività del linguaggio e d'altronde è innegabile che l'ereditarietà singola sia l'approccio a oggi di maggior successo.
- Offre un valido supporto per l'internazionalizzazione.
- Supporta nativamente la generazione di documentazione XML e anche scrittura di servizi Web XML Based.
- Può facilmente cooperare con altri linguaggi .Net compliant, un concetto pilastro per .Net.

- Tramite il progetto Mono gira anche su piattaforme Linux e Mac.
- Tramite l’eccellente suite Xamarin può essere adoperato anche per creare applicazioni Android.
- Dispone pienamente di tutte le features proprie di .Net, quindi un efficiente meccanismo di garbage collection, gestione delle eccezioni avanzata, dalla versione 4.0 in avanti è possibile usare anche il design by contract e sfruttare le CPU multi core.

Differenze con Java

Sebbene C# sia ritenuto simile a Java, esistono alcune importanti differenze fra i due linguaggi. Quella maggiormente visibile è certamente la presenza in C# di molti costrutti (alcuni sono reminiscenze del C++) che in Java sono stati deliberatamente vietati. I sostenitori del C# affermano che essi rendono il codice più chiaro; i sostenitori del Java affermano che proprio la presenza di un gran numero di parole-chiave e di regole da ricordare rende il linguaggio più difficile da imparare. Quello che in Java è chiamato package, in C# viene chiamato “namespace” o “spazio di nomi”. Un ulteriore livello di organizzazione in C# è costituito dagli “assembly”, che possono contenere al proprio interno diversi spazi di nomi.

Capitolo 1

Variabili e tipi di dato

Le somiglianze con Java si notano anche quando si parla di variabili. La dichiarazione e l'inizializzazione sono infatti le stesse. Un'altra somiglianza importante è l'uso del punto e virgola " ; " che ha la stessa funzione come in Java.

Tra le differenze più importanti spiccano i tipi di dato. In C#, infatti, ogni tipo presente in Java acquisisce alcune varianti che però non sono molto utilizzate e quindi al momento non indispensabili. Se sei curioso puoi visitare il sito: http://msdn.microsoft.com/it-it/library/ms228360(v=vs.90).aspx

C# è un linguaggio type-safe. Tutte le variabili devono essere dichiarate, e occorre specificarne il tipo.

Abbiamo due tipi di variabili: *valore* e *riferimento*.

Quando una variabile contiene un tipo valore, contiene direttamente un oggetto con il valore. Quando una variabile contiene un tipo riferimento, in realtà contiene un qualcosa che si riferisce a un oggetto. In questo caso una seconda variabile può contenere lo stesso riferimento.

Tipi valore

Vediamo i tipi valore predefiniti in C# e i corrispondenti tipi di sistema nel framework .Net.

Tipo	***Valori ammissibili***
sbyte	da -128 a 127
short	da -32768 a 32767
int	da -2147483648 a 2147483647
long	da -9223372036854775808 a 9223372036854775807
byte	da 0 a 255
ushort	da 0 a 65535
	da 0 a 4294967295

uint	
ulong	da 0 a 18446744073709551615
float	da ±1.5 x 10-45 a ±3.4 x 1038 con 7 cifre significative
double	da ±5.0 x 10-324 a ±1.7 x 10308 con 15 o 16 cifre significative
decimal	da ±1.0 x 10-28 a ±7.9 x 1028 con 28 o 29 cifre significative
char	caretteri unicode (16 bit)
bool	true or false

Vediamo degli esempi di dichiarazione di variabili: int var1 = 5;

int var2 = var1;

var2 = 15;

Notazione a Cammello Alto

Diversamente da Java, la notazione utilizzata in C# ha una piccola variante. I nomi dei metodi (presto li conoscerai) iniziano per lettera maiuscola quindi avrai, ad esempio, MioMetodo e non mioMetodo come in Java.

If – else e le relazioni logiche

L'istruzione principale che consente di gestire il controllo del flusso di un programma è l' "if" (salto condizionato).

Questa particolare istruzione permette di dare una scelta al programma, cosicché al verificarsi di una certa condizione, esegua determinate istruzioni.

Prima di procedere, però, dobbiamo fare un piccolo passo indietro e definire meglio un particolare tipo di variabile: *bool*.

Il contenuto di questa variabile può assumere valore "true" o "false". Una qualsiasi espressione logica assume un valore di tipo bool in quanto può essere vera o falsa. Di seguito sono presentate le tabelle logiche (dette anche di verità) più usate.

/	/	AND	OR	XOR
P1	P2	P1 & P2	P1 | P2	P1 ^ P2
T	T	T	T	F
T	F	F	T	T
F	T	F	T	T
F	F	F	F	F

<	minore
<=	minore uguale
>	maggiore
>=	maggiore uguale
!=	diverso
==	uguale

(Questa è usata per la relazione tra numeri.)

Date due variabili bool, ne possiamo esprimere una terza come combinazione di queste tramite una funzione sopra descritta: `bool a = true;`

```
bool b = false;

bool c = a & b;
```

In questo caso la variabile “c” varrà “false”, in quanto è il risultato dell’espressione logica AND tra “true” e “false”.

Funzioni particolari

Esiste una variante di AND e OR che può essere scritta raddoppiando il simbolo della funzione (&& per AND e || per OR). Questo permette di

arrivare alla soluzione visionando solo il primo degli operandi. Infatti nel caso dell'AND se il primo operando è "false" il risultato sarà "false" a priori. Nel caso dell'OR se il primo operando è "true" ne risulterà un valore "true". Solo in questi casi il controllo del secondo operando viene evitato, altrimenti si comportano normalmente.

Il costrutto if

Il costrutto "if" è definito nel seguente modo: **`if (condizione)`**

```
{
      istruzione 1;
      istruzione 2;
      ...
      istruzione n;
}
```

Se la condizione è "true" (rappresentata da una bool) il programma eseguirà le istruzioni contenute all'interno del corpo dell'if, determinato dalle parentesi graffe.

Un esempio:

```
if (anni >= 18)
Console.WriteLine("sei maggiorenne");
```

Quindi, se la variabile anni è maggiore uguale di 18, verrà stampato su console "sei maggiorenne".

Nota bene che quando il corpo dell'if è formato da un'unica istruzione, solo in questo caso, si possono omettere le parentesi graffe (sconsigliato).

Un'istruzione di selezione analoga è il costrutto if — else. Questo presenta un'estensione di quanto visto prima: **`if (condizione)`**

```
    {
      istruzione 1;
    }
      else
    {
      istruzione 2;
}
```

Se la condizione all'interno dell'if è "true" allora esegue "l'istruzione 1", altrimenti esegue l'istruzione contenuta nell'else.

Un esempio:

```
if (anni >= 18)

Console.WriteLine("sei maggiorenne");

else

Console.WriteLine("sei minorenne");
```

Una particolarità di questo costrutto è che può essere usato in combinazione per verificare un insieme di condizioni: "if a cascata".

```
if (condizione 1)

{

istruzione 1;

} else if (condizione 2)

{

istruzione 2;

} else

{

istruzione 3;
```

```
}
```

Come suggerisce l’esempio, i controlli vengono effettuati uno dopo l’altro. Se il primo if non viene verificato si passa al secondo, se anche in questo si ha esito negativo si passa al terzo e così via. In caso nessuna condizione fosse vera, verrà eseguito il codice presente nell’ultimo else.

Istruzione switch

Un modo più elegante di scrivere l’if a cascata è il costrutto switch: **`switch (variabile)`**

```
{
      case 1: istruzione 1; istruzione 2; break;
      case 2: istruzione 1; istruzione 2; break;
      ...
      case n: istruzione 1; istruzione 2; break;
      default: istruzione 1; break;
}
```

Questo può essere usato solo con variabili di tipo intero. Il suo valore viene confrontato con quello dei singoli casi e, se corrisponde a uno di questi, esegue le operazioni corrispondenti. L’istruzione break determina la fine delle operazioni da eseguire e ne esplicita l’uscita dal costrutto. Se nessuno dei casi dovesse corrispondere al valore della variabile verranno eseguite le operazioni definite in “default”. Le istruzioni “break” e “default” non sono obbligatorie, me per la stesura di un codice corretto è sempre bene usarli.

Qualora si omettesse l’istruzione “break” alla fine di ogni caso, una volta eseguite le istruzioni relative, verrebbero eseguite anche quelle dei casi sottostanti e ciò potrebbe determinare un uso errato di questo costrutto. “default” ti aiuta a definire casi di poco interesse o addirittura a risolvere errori derivanti da una programmazione non corretta. Se ad esempio ci si aspettava il valore uguale a 1 o 2, ma il programma ottiene un valore diverso, ad esempio 5, entra nel caso di default che potrebbe avvertire l’utente che il valore inserito non è

accettato.

Cicli do-while e while in C#

Per proseguire la nostra trattazione delle strutture di controllo introdurremo dei nuovi costrutti.

Ci viene facile pensare che il computer (o il nostro dispositivo mobile) esegua le operazioni del nostro programma esattamente nell'ordine con cui esse vengono scritte, una per volta, fino alla fine del codice. Abbiamo già visto che possiamo modificare il flusso del programma tramite le strutture di controllo. Oltre al costrutto *if-then-else*, che abbiamo già studiato, ne esistono altri che ci permettono svariate operazioni. Ad esempio, pensiamo ad uno scenario in cui dobbiamo ripetere una determinata azione molte volte, fino a quando una condizione viene verificata: sarebbe molto scomodo scrivere il codice dell'azione tante volte quanto serve. Spesso non conosciamo in partenze il numero esatto delle iterazioni da compiere, in questo caso l'approccio proposto risulterebbe impossibile.

Per essere più chiari introduciamo un semplice esempio. Supponiamo di volere stampare a video i primi 10 numeri interi. Con il bagaglio di conoscenze che abbiamo finora non possiamo fare altro che scrivere per 10 volte il codice relativo alla stampa di un numero. Penserete subito che la cosa non è poi così difficile o stancante, ma se dovessimo stampare i primi 1000 numeri? O i primi 10000? Per fortuna non dobbiamo copiare ed incollare la stessa linea di codice per tante volte, possiamo ricorrere ad un ciclo!

Il primo tipo di ciclo che impareremo ad usare è il *do-while*: Cominciamo subito con un pratico esempio, proviamo a stampare a video i primi 1000 numeri interi. Il codice da scrivere è il seguente: `int i = 1;`

```
do
{
      Console.WriteLine("{0}", i);
      i = i+1;
} while (i <= 1000);
```

Vediamo come funziona questo semplice esempio analizzandolo riga per riga:
`int i = 1;`

Come già sappiamo questa riga non fa altro che creare una nuova variabile di tipo intero con il valore iniziale 1. Useremo questa variabile come iteratore nel ciclo.

`do`

La keyword do introduce un ciclo *do-while*. Tutte le operazioni contenute all'interno del blocco racchiuso dalle parentesi graffe verranno ripetute fino a che la condizione espressa dal *while* sarà verificata. In questo caso la condizione è i <= 1000. Ciò significa che fino a che la variabile i ha un valore minore di 1000 verranno ripetute le operazioni all'interno delle parentesi graffe, ovvero:
`Console.WriteLine("{0}", i);`

`i = i + 1;`

Queste due operazioni non fanno altro che stampare a video il valore corrente della variabile i e successivamente incrementare di uno questo valore.

Avremo quindi che alla prima iterazione del ciclo la variabile i avrà il valore 1, che verrà stampato a video e successivamente incrementato. Alla seconda iterazione avremo che i ha valore 2, quindi verrà stampato questo valore e successivamente nuovamente incrementato. Questa operazione proseguirà fino a che la variabile i assumerà valore 1001. A questo punto, infatti, la condizione espressa dal *while* non sarà più verificata ed il ciclo si arresterà. Il programma potrà quindi continuare con le eventuali operazioni successive.

> **Un errore frequente**
>
> Uno degli errori più frequenti per i principianti è quello di non introdurre all'interno del blocco di un ciclo una operazione che possa far cambiare la condizione. In questo caso l'operazione in questione è l'incremento della variabile i. Se questa operazione non fosse presente la variabile i avrebbe sempre lo stesso valore (1), quindi, non solo verrebbe stampato sempre il numero 1, ma addirittura il ciclo non si arresterebbe mai, poiché la condizione i <= 1000 sarebbe sempre verificata!

Al lettore più attento non sarà sfuggito il fatto che con il costrutto *do-while* le

operazioni nel blocco verranno effettuate almeno una volta.

Spesso non è detto che le operazioni in questione debbano essere eseguite, in questi casi si può ricorrere al ciclo *while*.

I cicli *while* sono molto simili ai cicli *do-while*, l'unica differenza risiede nel fatto che la condizione viene verificata all'inizio del ciclo.

Se la condizione non è verificata il ciclo non ha luogo.

La sintassi è:

```
while (<Test>)
{
      <codice da ripetere>
}
```

Come esempio vediamo come calcolare la potenza di un numero.

Supponiamo di voler calcolare il valore di 2^10, possiamo scrivere il seguente codice:

```
int base = 2;
int esponente = 10;
int i = 0;
int risultato = 1;

while(i < esponente){
      risultato = risultato*base;
      i = i + 1;
}
```

Il codice a questo punto dovrebbe essere abbastanza chiaro. Si inizializzano quattro variabili di tipo intero: la base della potenza da calcolare, l'esponente, una variabile i che ci serve come contatore del ciclo per tenere traccia di quante iterazioni sono state eseguite ed una variabile che conterrà il risultato. Il ciclo viene ripetuto tante volte quanto vale l'esponente.

La riga `risultato = risultato*base;` non fa altro che aggiornare il valore

della variabile risultato. Alla fine della prima iterazione questa variabile avrà valore 2, alla fine della seconda 4

e così via.

Ciclo for in C#

Per terminare la nostra trattazione sui cicli, dopo avere introdotto i costrutti do-while e while passeremo ora allo studio del ciclo for.

Sostanzialmente questo nuovo costrutto ha la stessa funzione di quelli già presentati, ovvero reiterare l'esecuzione di una o più istruzioni. La differenza principale risiede nella definizione del numero delle volte che questo ciclo deve essere eseguito.

Il ciclo for viene utilizzato di solito quando il numero delle iterazioni è conosciuto a priori. Vediamo la sintassi: **`for(inizializzazione valore; condizione; incremento)`**

```
{
      <istruzione 1>;
      <istruzione 2>;
      …
      <istruzione n>;
}
```

In "inizializzazione valore" si stabilisce il valore iniziale della variabile che tiene conto del numero di iterazioni.

La condizione è analoga a quella vista nel ciclo do-while o in quello while, essa viene verificata e viene eseguito il blocco di istruzioni esclusivamente se essa ha valore true.

Infine in "incremento" si aumenta il valore della variabile inizializzata (salvo eccezioni, che vedremo in seguito).

Esempio Supponiamo di voler sommare i primi 50 numeri interi, potremmo scrivere il seguente codice: **`int somma = 0;`**

```
for(int i = 1; i <= 50; i++){
```

```
        somma += i;
}

Console.WriteLine("{0}", somma);
```

Analizziamo ora il codice appena proposto.

```
int somma = 0;
```

Inizializziamo una variabile intera, che chiamiamo somma, che al termine dell'esecuzione del ciclo dovrà contenere il risultato.

```
for(int i = 1; i <= 50; i++)
```

il significato del ciclo proposto è intuitivo. Inizializziamo la variabile contatore del ciclo i ad uno, che sarà il primo numero da sommare. La condizione da verificare è che l'indice di iterazione deve essere minore o uguale di 50. Questo perché vogliamo sommare i primi cinquanta numeri, useremo quindi la stessa variabile i per incrementare il valore della somma.

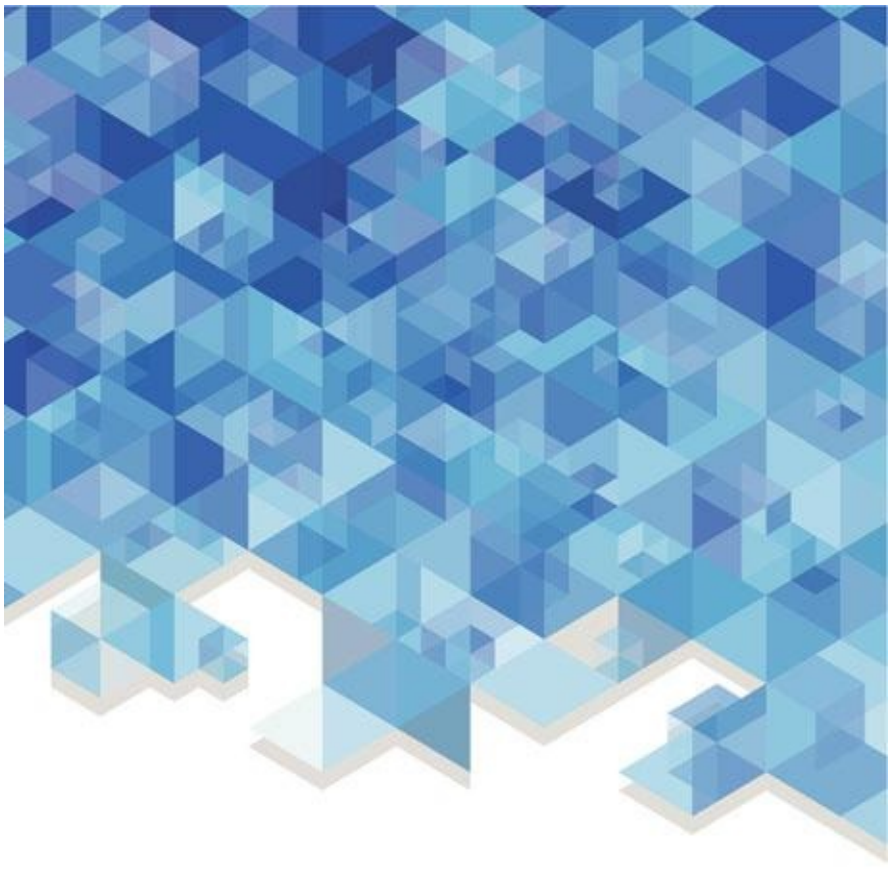
```
somma += i;
```

L'aggiornamento del risultato avviene tramite questa operazione. Come abbiamo visto in precedenza, l'istruzione appena scritta è equivalente a: `somma = somma + i;`

ovvero si aggiunge al valore corrente della variabile somma il valore della variabile i, che varia ad ogni iterazione del ciclo. Ai lettori più attenti non sarà sfuggito che all'iterazione i-esima del ciclo, la variabile somma conterrà il risultato dell'addizione dei primi i numeri interi.

A ciclo ultimato mostreremo il risultato della computazione con il solito: `Console.WriteLine("{0}", somma);`

Capitolo 2



Array

I tipi di dato visti finora hanno in comune il fatto che essi memorizzano un solo valore, sia esso un intero, un decimale, una stringa *etc*. In molti casi si ha la necessità di memorizzare diversi valori dello stesso tipo. Partiamo con un esempio. Supponiamo di voler memorizzare i nomi delle nazioni europee, potremmo procedere utilizzato tante variabili di tipo stringa, scrivendo un codice simile al seguente: **`string stato1 = "Francia"`**

```
string stato2 = "Germania"

string stato3 = "Italia"

string stato4 = "Spagna"
```

Appare chiaro sin da subito che questo approccio risulta particolarmente scomodo poiché si ha a che fare con un gran numero di variabili non legate tra loro. L'unica elemento di unione risiede nell'idea del programmatore. Ci troveremmo davanti a noiosi sforzi legati al fatto che questo approccio ci porta alla scrittura di codice differente ogniqualvolta dobbiamo processare una o più di queste variabili. Si pensi che fatica compiere la stessa operazione su tutte le variabili senza poter ricorrere ad un ciclo!

Per fortuna non siamo costretti a scrivere codice del genere. Possiamo utilizzare gli array. Un array è una lista indicizzata di variabili dello stesso tipo, memorizzata in un'unica variabile. Si parla di liste indicizzate perché effettivamente si accede agli elementi di un array tramite un indice, che identifica la posizione della variabile sulla quale si agisce. L'indice può assumere valori che variano da 0 (il primo elemento della lista), a n-1, dove n è il numero di elementi contenuti nell'array. Per accedere ad un determinato elemento usando l'indice di posizione si ricorre alla notazione con parentesi quadre: **`nomeArray[<indice>]`**

Dichiarazione di array

Creare un nuovo array è semplice. Proviamo a crearne uno per modificare il codice scritto.

```
string[] stati;
```

In questo modo abbiamo creato una variabile array di stringhe (ricordiamo che le variabili contenute nell'array devono essere tutte dello stesso tipo). Non vi è ragione di accedere agli elementi dell'array creato, semplicemente perchè esso non è ancora stato inizializzato. L'inizializzazione può essere effettuata facilmente, usando due metodi diversi.

Il primo metodo consiste nello specificare direttamente i valori contenuti nel nostro array. Per fare ciò si usano le parentesi graffe: `string[] stati = {"Francia", "Germania", "Italia", "Spagna"};`

Si noti che adesso l'elemento stati[0] sarà "Francia", stati[1] Germania e così via.

Il secondo metodo per inizializzare un array è ricorrere alla seguente sintassi:

```
string[] stati = new string[4];
```

In questo caso abbiamo creato un array di 4 stringhe.

I due metodi possono essere anche combinati assieme ottenendo espressioni del tipo:

```
string[] stati = new string[4]{"Francia", "Germania", "Italia",
"Spagna"};
```

Si noti che la lunghezza dell'array, dichiarata tra parentesi quadre, deve essere uguale al numero di elementi inizializzati tra le parentesi graffe.

Accedere agli elemento di un array

L'accesso agli elementi di un array, come spiegato, è semplice e intuitivo, grazie alla notazione con le parentesi quadre. I pregi di questa notazione sono immediatamente intuibili se si prova a scrivere un ciclo che accede a tutti gli elementi di un array.

A titolo esemplificativo supponiamo di voler stampare a video tutti i nomi delle

nazioni europee contenuti nell’array creato. Possiamo procedere nel seguente modo:

```
for ( int i = 0; i < stati.Length; i++ ){
	Console.WriteLine( stati[i] );
}
```

Lo spezzone di codice appena scritto non fa altro che usare la variabile i del ciclo come indice per scorrere gli elementi dell’array. Si noti che in stati.Length viene memorizzato il numero di elementi presenti nell’array. Non vi è bisogno di inizializzare questo intero, esso viene legato automaticamente ad ogni array in fase di inizializzazione.

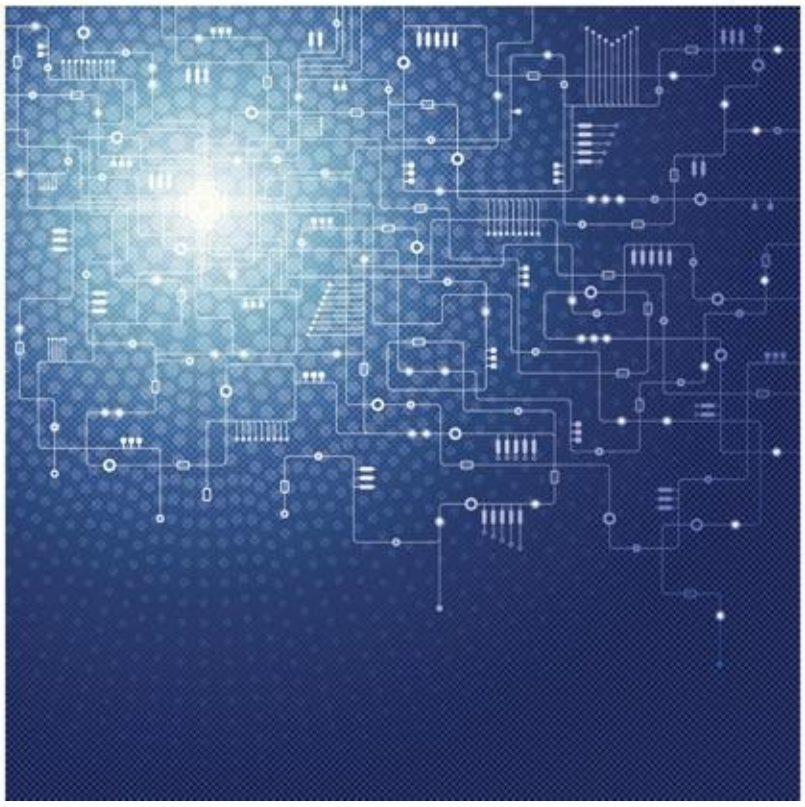
Foreach

Abbiamo a disposizione un altro metodo per scorrere gli elementi di un array: il costrutto foreach. La sintassi è semplice, possiamo scrivere:

```
foreach ( string stato in stati ){
	Console.WriteLine( stato );
}
```

Il risultato è esattamente lo stesso del for precedente. Si evita così l’utilizzo degli indici e si perde la possibilità di modificare gli elementi dell’array all’interno del ciclo.

Capitolo 3



Introduzione

La programmazione orientata agli oggetti è uno stile fondamentale di programmazione adottato dalla maggior parte dei linguaggi moderni. Questo permette un livello di astrazione superiore rispetto alla programmazione procedurale, rendendo il codice molto più facile da gestire, ad esempio per un'ulteriore estensione del programma o per la sua manutenzione.

Classi e oggetti

Le classi costituiscono l'elemento base della programmazione ad oggetti. La loro creazione permette la definizione di nuovi tipi di dato, con particolari caratteristiche e proprietà, tendendo ad astrarre, nel modo più semplice possibile, oggetti presenti nella realtà. Le classi, quindi, costituiscono le regole e i tratti somatici che ogni oggetto, di quel tipo, deve osservare.

Un oggetto non è altro che un'istanza di una determinata classe. L'istanza rappresenta un elemento distinto, allocato in memoria, con propri attributi appartenenti alla stessa natura di quelli definiti nella classe, ma che possono assumere valori propri e differire da istanza ad istanza. Per capire meglio il significato di classe ed oggetto si pensi al concetto di 'Persona'. Il colore degli occhi, la lunghezza dei capelli e l'altezza sono tutte caratteristiche comuni alle persone, ma ognuna di esse non le presenta necessariamente uguali a quelle di un'altra. Da qui si può dire che la 'classe persona' costituisce lo stampo di ogni 'oggetto persona', e che ogni istanza è un elemento concreto in memoria che può assumere valori propri.

Definire una classe

La sintassi per definire una classe è la seguente: **`[modificatori] class nome`**

```
{
    ...
```

```
}
```

Esempio:

```
public class punto
{
    ...
}
```

I modificatori delle classi sono due:

- *public* estende la visibilità di questa classe a tutte le altre. Il concetto di visibilità verrà trattato successivamente. Per il momento utilizza questo attributo in ogni caso.
- *internal* limita la visibilità della classe ai soli elementi appartenenti allo stesso assembly della classe.

Nelle classi dobbiamo distinguere due elementi principali: variabili e metodi.

Variabili

Le variabili (o variabili di istanza) attribuiscono a una classe le caratteristiche di cui vogliamo dotarla. Queste possono comprendere tipi primitivi, come int, double, bool, char o anche altri oggetti. La sintassi per dichiarare una variabile è la seguente: **`[modificatori] tipo nome`**

I modificatori delle classi sono quattro:

- *public* la variabile è visibile in tutte le altre classi.
- *private* limita la visibilità solo alla classe stessa.
- *protected* la visibilità è limitata alla classe e alle sue ereditarie (questo concetto verrà trattato in seguito).
- *internal* funziona allo stesso modo del modificare delle classi.

Ad esempio se si volesse creare una classe “Punto” nel piano cartesiano, doteremmo la classe in questione di due coordinate, e quindi di due variabili di tipo reale: **`private double x;`**

`private double y;`

In questo modo si è creato “lo stampo” per ogni oggetto Punto, che avrà queste due variabili come “caratteristica”.

Metodi

I metodi rappresentano le azioni che è possibile effettuare sugli oggetti, o che un singolo oggetto può fare. La sintassi per scrivere un metodo è la seguente: **`[modificatori] tipoRitorno nome (parametri)`**

```
{

	...
}
```

I modificatori sono identici a quelli delle variabili e mantengono le stesse funzioni. Il tipo di ritorno identifica il risultato che l’elaborazione del metodo restituisce. Nel caso l’elaborazione sia passiva, in altre parole non produca output, il tipo di ritorno sarà “void” (nulla). Il metodo può ricevere delle variabili che serviranno per poter completare l’elaborazione, queste sono dette parametri e sono opzionali. Ricordati che ogni metodo che restituisce un tipo (quindi non void) deve includere al suo interno l’istruzione “return”. Questa serve per mandare in output il frutto di un’elaborazione. Ecco un esempio: **`public int somma(int a, int b)`**

```
{

	int somma = a + b;
	return somma;
}
```

Esiste un metodo particolare chiamato “Costruttore”. Questo è la funzione (o metodo) che permette di definire come deve avvenire la creazione di un oggetto,

e quindi, come deve essere istanziato. Qui è bene scrivere tutto il codice che definirà i valori iniziali delle variabili d'istanza. La sua definizione è identica a quella di un qualsiasi metodo ma differisce per l'assenza del nome e per il tipo di ritorno che è la classe di appartenenza. Un esempio completo sulla classe "Punto": **`public class Punto`**

```
{
        private double x;
        private double y;
        public Punto()
        {
          x = 0.0;
          y = 0.0;
        }
}
```

Dichiarazione di un oggetto

Un oggetto è considerato come una qualsiasi variabile. Può essere dichiarato senza inizializzazione: **`TipoOggetto nome;`**

oppure con inizializzazione diretta: **`TipoOggetto nome = new TipoOggetto ([parametri]);`**

L'istruzione "new" corrisponde all'allocazione diretta di un oggetto in memoria e serve per creare un'istanza.

Puoi accedere alle variabili di un oggetto o invocare un suo metodo nel seguente modo: **`nomeOggetto.nomeVariabile`**

`nomeOggetto.nomeMetodo([parametri]);`

Ricordati che la visibilità dei metodi e delle variabili deve essere estesa alla classe in cui stai operando, quindi lavorando su un'altra classe non ti è possibile, ad esempio, richiamare l'accesso alla variabile (private) "x" di un' istanza dell'oggetto "Punto".

Passaggio di parametri in C#

Quando una funzione agisce su degli argomenti può utilizzarli per giungere ad un risultato o modificare gli argomenti stessi facendo si che siano loro i risultati dell'operazione.

Passaggio di parametri per valore

Il modo più semplice di passare un dato a una funzione è il passaggio per valore che garantisce alla funzione l'accesso al parametro in questione ma non gli consente di apportare ad esso modifiche definitive: Ecco un esempio di passaggio per valore: **`int x = 5;`**

```
x++;

Resetta(x);

x++;


…


public void Resetta (int i) {

    i=0;
}
```

La variabile intera x vale inizialmente x e, dopo un primo incremento, il suo valore diventa 6. A questo punto viene chiamato il metodo Resetta sul *valore* di x=6. L'intero i, nella funzione, contiene una copia del valore di x e vale quindi 6. Dopo l'istruzione i=0, il valore di i diventa 0 ma quello di x resta 6! Alla fine viene chiamata nuovamente l'istruzione x++ che rende x = 7 e non 1.

Il passaggio per valore può quindi essere visto come un passaggio per copia che non modifica il valore originario che viene passato. Si può usare questo metodo quando non si vuole modificare il valore del parametro ma lo si vuole utilizzare per elaborare un altro risultato, ad esempio per restituire il valore della radice

quadrata di x senza modificare x stessa.

Passaggio per riferimento

Quando viene passato un oggetto come parametro anziché un tipo di base, questo viene passato per riferimento. Viene passato al metodo un riferimento all'oggetto e non una sua copia in modo che i cambiamenti apportati all'oggetto si ripercuotano anche fuori dal metodo. È possibile ottenere questo risultato anche con le parole chiave *ref* e *out*.

```
int x = 5;

x++;

Resetta(ref x);

x++;


…


public void Resetta (ref int i) {

    i=0;
}
```

In questo caso il valore finale di x sarà davvero 1 perché i cambiamenti apportati all'interno del metodo resetta saranno permanenti.

La differenza tra ref e out sta nel fatto che quando si utilizza ref il valore della variabile dovrebbe essere già definito mentre quando si usa out si vuole che la variabile, inizialmente senza valore, venga inizializzata dalla funzione stessa.

Ereditarietà

L'ereditarietà è una sorta di "riciclaggio" del codice scritto in cui una classe

“assorbe” proprietà e metodi definiti in una classe preesistente e li modifica o li estende. Ciò permette il riuso del codice già scritto, velocizzando la fase di implementazione delle applicazioni e minimizzando il pericolo di incorrere in errori, dato che si utilizzano classi la cui correttezza dovrebbe essere verificata preliminarmente.

La classe dalla quale si ereditano i membri si chiama classe base o superclasse, mentre la nuova classe prende il nome di classe derivata o sottoclasse. La classe derivata a sua volta potrà essere classe base per ulteriori derivazioni.

L’utilizzo tipico dell’ereditarietà permette di creare classi derivate che possiedono un livello di specializzazione maggiore rispetto alla classe base, poiché, solitamente, la classe derivata aggiunge nuove funzionalità alla classe di partenza. In alcuni linguaggi di programmazione, come il C++, è possibile creare delle classi che estendono più classi base, in questo caso si parla di ereditarietà multipla. C# non permette questo tipo di operazione, si parla, quindi, di ereditarietà singola.

La relazione di specializzazione, che in C# è espressa tramite il concetto di ereditarietà, può essere rappresentata tramite l’espressione is-a (è un). Ad esempio possiamo dire che lo studente è una persona, ciò significa che uno studente possiede tutte le caratteristiche peculiari delle persone, ma ne possiede anche altre che lo specializzano ed appartengono alla sua categoria.

Creare una classe derivata in C# è semplice, basta scrivere dopo il nome della classe derivata due punti seguiti dal nome della classe base.

Ad esempio:

```
public class Studente : Persona
```

in questo caso la classe studente eredita le proprietà ed i metodi definiti nella superclasse Persona, in particolare può eccedere a tutti i membri pubblici e protetti.

Estendiamo il nostro esempio.

```
public class Persona
{
    string nome;
    int eta;
```

```
    public Persona(string Nome, int Eta)
    {
      nome = Nome;
      eta = Eta;
    }

    public string Nome
    {
      get { return nome; } set { nome = value; }
    }

    public int Eta
    {
      get { return eta; } set { eta = value; }
    }
}
```

Creiamo ora una classe derivata. Notiamo che il costruttore non viene ereditato. Nel nostro caso chiameremo direttamente il costruttore della superclasse.

```
public class Studente : Persona

    {
    private int matricola;

    public Studente (string Nome, int Eta, int Matricola)
    : base (Nome, Eta)
    {
      matricola = Matricola;
    }

    public int Matricola
    {
      get { return matricola;} set { matricola = value;}
    }
```

```
}
```

Nell’esempio appena proposto abbiamo visto una classe Studente che eredita le proprietà ed i metodi della classe Persona e aggiunge una nuova proprietà (matricola) che specializza il suo tipo.

“Esiste una semplice regola che stabilisce se debba essere usata l’ereditarietà?” La risposta è sì. La regola “*is-a*” afferma che ogni oggetto istanza di una classe derivata è un oggetto istanza della classe base. Ad esempio abbiamo visto che gli oggetti istanza della classe *Studente* sono oggetti della classe *Persona*. Non vale il contrario, non è detto che tutte le persone siano studenti.

Esiste un altro modo per esprimere la regola “*is-a*”, che prende il nome di *principio di sostituzione*. Esso afferma che ogni volta che un programma aspetta un oggetto di una determinata classe base, si può ricorrere all’utilizzo di un oggetto di una classe derivata. Questo oggetto, essendo in tutto e per tutto un oggetto della classe base non porterebbe ad alcuna violazione delle condizioni del programma.

Vediamo subito un esempio:

Possiamo dichiarare una variabile di tipo Persona scrivendo: `Persona p;`

possiamo inizializzare la variabile p indifferentemente con un oggetto della classe Persona, o, analogamente di una delle sue classi derivate (nel nostro caso Studente).

Possiamo quindi scrivere:

```
p= new Persona(“Michele”, 23);
```

oppure

```
p= new Studente(“Michele”, 23, 123456);
```

Entrambe le inizializzazioni sono corrette. La prima è banalmente corretta poiché si associa alla variabile p un oggetto istanza della classe Persona. La seconda è ugualmente corretta poiché un oggetto della classe Studente è un oggetto della classe Persona essendo la prima classe una sottoclasse della seconda.

Questo particolare comportamento delle variabili prende il nome di *polimorfismo*. Una qualsiasi variabile può fare riferimento a un oggetto di una determinata classe X oppure a un oggetto di qualsiasi sottoclasse della classe X.

Accessibilità

Quando realizziamo l’ereditarietà da una determinata classe dobbiamo porci sempre il problema dell’accessibilità dei membri della classe base. Se dichiariamo i membri di questa classe *private*, essi non saranno accessibili dalle sottoclassi, se invece li dichiariamo *public* essi saranno accessibili non solo alle sottoclassi, ma anche alle altre classi che non fanno parte della gerarchia di ereditarietà che stiamo sviluppando. Spesso è necessario che i membri siano accessibili alle sole sottoclassi. Per fare ciò ci viene in aiuto un nuovo tipo di limitatore di accessibilità: la parola chiave *protected*. Essa stabilisce che i membri di una classe base siano accessibili alla sola classe base e alle sue classi derivate.

Possiamo gestire non solo il livello di protezione dei membri, ma anche il loro comportamento nel caso di ereditarietà. Definendo un membro di una superclasse *virtual*, diamo la possibilità alle sottoclassi di sovrascriverlo, ed eventualmente, di modificare la sua implementazione, senza però eliminare il codice originale, che risulta ancora accessibile nella classe base.

Le classi base possono essere anche definite *abstract*. Le classi astratte non possono essere istanziate in maniera diretta, per utilizzarle bisogna definire una classe che li estenda. Le classi astratte possono avere anche dei membri astratti, che non hanno alcuna implementazione. L’onere di fornire una implementazione a questi membri è a carico di chi realizza una classe derivata dalla classe astratta

in questione.

Infine è possibile definire una classe base *sealed*. In tal caso non è possibile estendere questa classe. Il motivo principale per cui si utilizza il modificatore *sealed* è perché spesso si desidera che la semantica della classe base non venga modificata.

Interfacce

Per definire un insieme di requisiti che delle classi devono avere si ricorre al meccanismo delle interfacce.

Un'interfaccia è un insieme di metodi pubblici che sono definiti insieme con lo scopo di descrivere funzionalità specifiche. L'implementazione dei metodi definiti dalle interfacce è a carico di chi sviluppa una classe che implementa quella specifica interfaccia. Quando si decide di implementare un'interfaccia è necessario realizzare tutti i metodi definiti in essa.

Non possono esistere oggetti istanza diretta di una interfaccia. Non è possibile istanziare un'interfaccia come si fa con le classi, inoltre non è possibile implementare i metodi all'interno dell'interfaccia stessa né tantomeno introdurre variabili.

La definizione di un'interfaccia è elementare, vediamo subito un esempio:

```
public interface Esempio

{

    void stampa(string s);

}
```

Abbiamo appena definito un'interfaccia di nome Esempio che espone un solo metodo, naturalmente pubblico. Si noti che l'implementazione del metodo non è presente. Come detto, essa dovrà essere fornita nelle classi che implementano questa interfaccia.

Per essere chiari vediamo un esempio:

```
class ImplementazioneEsempio : Esempio

{

    public void stampa(string s)
```

```
    {
      Console.WriteLine(“Ecco la stringa che vuoi stampare : {0}”,
s);
    }
}
```

Il codice dovrebbe essere chiaro. Abbiamo creato una classe ImplementazioneEsempio che implementa (si notino i due punti) l’interfaccia Esempio. Questa interfaccia espone un solo metodo, stampa, che è stato implementato all’interno della nostra classe.

Lo scopo delle interfacce è quello di definire i requisiti che una determinata classe deve avere. Ciò può essere molto utile se si sta sviluppando un servizio. Chi fornisce il servizio può richiedere che vengano fornite determinate funzionalità e per esprimere in dettaglio queste funzionalità può ricorrere alle interfacce.

Abbiamo in precedenza introdotto le classi astratte. Il lettore attento sicuramente avrà notato le similitudini tra questo tipo di classi e le interfacce. Sia le classi astratte sia le interfacce possono contenere metodi che devono essere implementati da una sottoclasse. Nessuna delle due può essere istanziata direttamente, anche se si può definire variabili di questo tipo assegnando poi a queste variabili degli oggetti di classi derivate da esse.

Delle domande sorgono spontanee. Quali sono le differenze? Quando scegliere un’interfaccia e quando una classe astratta?

La risposta è semplice. Bisogna osservare che le classi derivate possono estendere esclusivamente una classe base. Ciò significa che una classe derivata può estendere direttamente una sola classe astratta. Se si vuole estendere più classi astratte bisognerà ricorrere ad una catena di classi derivate. Le interfacce, invece, non essendo classi, non presentano questo problema. Una classe derivata può estendere un numero arbitrario di interfacce, basta scrivere un codice come questo: **`class ImplementazioneEsempio : Interfaccia1, Interfaccia2, Interfaccia3`**

```
{
    …
}
```

Sebbene sia una differenza sostanziale, essa non compromette la libertà di scelta. Come detto, tramite una catena di classi derivate è possibile ottenere gli stessi risultati.

Le classi astratte, a differenza delle interfacce, possono contenere l'implementazione di alcuni metodi, i quali possono essere virtual e quindi ridefinibili nelle sottoclassi. Le interfacce, invece, non possono fornire l'implementazione dei metodi. Essi devono essere necessariamente vuoti e con visibilità pubblica. La visibilità dei metodi delle classi astratte non è limitata, invece, a quella pubblica. Infine le interfacce, a differenza delle classi astratte, non possono contenere campi, costruttori, distruttori, membri statici o costanti.

La scelta dipende naturalmente dalle esigenze del programmatore, che dovrà basarsi sulle considerazioni appena espresse. Molti programmatori hanno l'abitudine di inserire all'interno delle classi astratte esclusivamente metodi astratti. Questa non è una buona pratica. Se è possibile bisogna spostare all'interno di una superclasse più metodi possibili ed è proprio la presenza di metodi implementati che spesso porta a scegliere la realizzazione di una classe astratta anziché una interfaccia.

Capitolo 4

File e persistenza in C#

Spesso si ha la necessità di memorizzare delle informazioni che siano accessibili anche dopo la terminazione di una applicazione. I dati salvati nelle variabili dei nostri programmi, infatti, se non memorizzati altrove, vengono persi quando terminiamo il programma stesso. Per ovviare a questo problema possiamo ricorrere ai file. Sebbene C# ci fornisca dei metodi molto semplici da utilizzare che ci permettono la scrittura e la lettura su file è doveroso introdurre preliminarmente alcuni concetti fondamentali.

Stream

Tutti i tipi di operazioni di input e output (I/O) in C# sono basati sull'utilizzo di stream. Gli stream non sono altro che delle rappresentazioni di dispositivi seriali, quali possono essere un flusso di rete, un indirizzo di memoria o semplicemente un file. Gli stream ci permettono di compiere un'astrazione del dispositivo che trattiamo. Grazie ad essi non dobbiamo preoccuparci del modo in cui i nostri dati fluiscono.

Esistono due tipi di stream:

- *output*; vengono utilizzati quando si vuole inviare dei dati ad una destinazione (file, schermo, rete etc).
- *input*; vengono utilizzati quando si vuole leggere dei dati da una sorgente (file, tastiera, rete etc).

Le classi utilizzate per rappresentare uno stream in C# sono molteplici, esse sono contenute nel namespace System.IO.

La classe File

La classe File è semplice da usare. Essa espone molti metodi statici che

permettono di compiere numerose operazioni sui file. Vi sono metodi per copiare file, crearli, cancellarli, spostarli *etc.* Qualsiasi operazione che svolgiamo normalmente su un file ha un relativo metodo in questa classe.

La classe FileInfo

Altra importante classe per la gestione dei file è FileInfo. A differenza della classe file essa non è statica. Un oggetto istanza di questa classe è creabile facilmente passando al suo costruttore il path del file: `FileInfo fileEsempio = new FileInfo(@"C:\FileDiEsempio.txt");`

Si noti l'utilizzo della chiocciola che fa si che la stringa venga interpretata come tale e la \ non venga considerata l'inizio di una sequenza di escape.

FileStream

Uno stream che punta a un file viene rappresentato tramite un'istanza della classe FileStream.

Vi sono diversi metodi per creare un oggetto di tipo FileStream. Il metodo più semplice consiste nel passare al costruttore due parametri: il path del file e la modalità di accesso o creazione. Per esempio, se volessimo aprire un file potremmo scrivere il seguente codice: `FileStream stream = File.Open(@"C:\FileDiEsempio.txt", FileMode.Open);`

Spesso si desidera specificare anche lo scopo dello stream. In questo caso dovremo passare al costruttore anche questa informazione. Gli scopi possibili sono elencati nella enumerazione FileAccess.

La classe FileStream agisce su byte e array di byte. Molto spesso bisogna operare su caratteri e stringhe, per questo motivo la classe FileStream è spesso sostituita dalle classi StreamReader e StreamWriter, che permettono, rispettivamente, di leggere o scrivere con facilità flussi di caratteri.

Bisogna sottolineare, però, che questa semplicità di utilizzo limita alcune operazioni. Se desideriamo, infatti, accedere ad una specifica posizione all'interno di un file dobbiamo necessariamente ricorrere ad un oggetto FileStream.

Abbiamo introdotto i concetti fondamentali che ci permetteranno di leggere da file e scrivere su di essi. Abbiamo visto in linea teorica cosa sono gli stream ed abbiamo studiato alcune caratteristiche fondamentali per la loro gestione. Passiamo adesso ad un pò di pratica che ci farà schiarire le idee.

Scriviamo una semplice applicazione che scrive del testo su un file.

```
using System;
using System.IO;

namespace esempioscrittura
{
    class Class1
    {
      [STAThread]
      static void Main(string[] args)
      {
        try
        {

          // Il parametro passato al costruttore è il path del file
          StreamWriter sw = new
StreamWriter(@"C:\FileDiEsempio.txt");

          // Scriviamo del testo nel file
          sw.WriteLine("Sono un testo di esempio e sto finendo in
un file...");

          // Chiudiamo lo stream
          sw.Close();
        }
        catch(Exception e)
        {
          Console.WriteLine("Eccezione: " + e.Message);
        }
```

```
        }
    }
}
```

Come si può notare la scrittura su file è elementare. L'unico elemento che può destare preoccupazione è la presenza di un blocco try-catch. Esso serve a gestire le eccezioni che possono capitare in alcune occasioni. Avremo modo in seguito di parlare dettagliatamente di questo costrutto.

Per leggere da file il processo è analogo, naturalmente dovremo utilizzare uno StreamReader anzichè uno StreamWriter. Vediamo un esempio: stampiamo a video l'intero contenuto di un file: `StreamReader reader = new StreamReader(@"C:\FileDiEsempio.txt");`

```
Console.WriteLine(reader.ReadToEnd());

reader.Close();
```

Anche in questo caso bastano poche righe per ottenere il risultato cercato.

Random Access File

StreamReader e StreamWriter non danno la possibilità di scrivere in posizioni arbitrarie del file, per questo scopo è necessario utilizzare un FileStream. Gli oggetti di questa classe, infatti, mantengono al loro interno un puntatore che indica la posizione dove verrà effettuata la prossima lettura o scrittura; questa posizione è modificabile tramite il metodo Seek().

Il metodo Seek() riceve due parametri: il primo parametro è il numero di byte che indica di quanto il puntatore si deve spostare. Il secondo parametro rappresenta la posizione dalla quale bisogna contare. Esiste una enumerazione, SeekOrigin, che ha tre valori: Begin, Current, End; essi possono essere utilizzati come il secondo parametro del metodo Seek() e rappresentano rispettivamente la posizione iniziale, corrente e finale tra quelle che può assumere il puntatore.

I file acceduti selezionando la posizione del puntatore vengono chiamati Random Access File, perché permettono lo spostamento di quest'ultimo in corrispondenza di qualsiasi byte del file.

Vediamo degli esempi:

```
FileDiEsempio.Seek(10, SeekOrigin.Begin);
```

Il puntatore viene spostato di dieci byte a partire dalla posizione iniziale del file.

```
FileDiEsempio.Seek(5, SeekOrigin.Current);
```

Il puntatore viene spostato di cinque posizioni a partire dalla posizione corrente del puntatore.

```
FileDiEsempio.Seek(-8, SeekOrigin.End);
```

Il puntatore viene spostato di otto posizioni all'indietro a partire dall'utlima posizione del file.

Isolated Storage

Un concetto intuitivo

L'Isolated Storage è un concetto molto intuitivo, a ogni applicazione viene assegnata una certa quantità di memoria fisica accessibile esclusivamente da quella applicazione. Nessuna applicazione avrà i permessi necessari ad accedere al file system del dispositivo o a un Isolated Storage di un'altra applicazione.

Per iniziare a utilizzare l'Isolated Storage nelle nostre applicazioni dobbiamo importare i relativi namespace, usando il codice seguente: `using Sytem.IO;`

```
using System.IO.IsolatedStorage
```

Ora vi è la necessità di ottenere un'istanza dell'Isolated Storage per la nostra applicazione. Si può procedere in due modi: aggiungere oggetti all'IsolatedStorageSettings basandosi su un approccio a dizionario.

Vediamo un esempio. Possiamo ottenere una istanza dell'IsolatedStorageSettings con il seguente frammento di codice: `IsolatedStorageSettings settings =`

```
IsolatedStorageSettings.ApplicationSettings;
```

Tramite l'oggetto creato possiamo aggiungere le informazioni da salvare con semplice codice come il seguente: `settings.Add("Nome", "Michele");`

```
settings.Add("Nazione", "Italia");
```

E alla fine salvare il tutto tramite il metodo Save:

```
settings.Save();
```

Naturalmente vorremo anche procedere a recuperare i dati salvati, ciò si può fare con estrema semplicità:

```
string nome = settings["Nome"].toString();
```

Adesso la variabile nome conterrà la stringa "Michele", precedentemente salvata.

Salvare altri tipi di oggetto

Possiamo salvare non solo stringhe, ma anche altri tipi di oggetto. Supponiamo di avere un oggetto istanza della classe X memorizzato nell'Isolated Storage con chiave y, potremo recuperare questo oggetto tramite la seguente riga di codice: `X oggettox = settings["y"] as X;`

L'approccio sin qui proposto è valido se non si ha la necessità di avere un controllo specifico sulla memorizzazione dei dati. Se, per esempio, si volesse creare cartelle o dare una struttura particolare ai dati salvati, si dovrebbe ricorrere ai metodi tradizionali per la lettura e scrittura su file e cartelle.

Innanzitutto è necessario ottenere una istanza dell'Isolated Storage:

```
IsolatedStorageFile isf =
IsolatedStorageFile.GetUserStoreForApplication();
```

Ora si può procedere con molte operazioni, come creare una cartella:

```
isf.CreateDirectory("CartellaDiEsempio");
```

o scrivere su un file nella cartella appena creata:

```
StreamWriter sw = new StreamWriter(new
IsolatedStorageFileStream("CartellaDiEsempio\\FileDiEsempio.dat",
FileMode.CreateNew, isf));

sw.WriteLine("Stiamo scrivendo dentro il file appena creato");

sw.close();
```

Leggere i file è altrettanto facile, basta usare uno StreamReader come spiegato in precedenza.

Capitolo 5


if(parameters.contains(
hql += " and p.name = :name";
}
if(parameters.contains("age")){
hql += " and p.age = :age";
}
TypedQuery<Person> query = em.
if(parameters.contains("name")){
query.setParameter("name",
}
if(parameters.contains("age")){
Integer.

La classe Object

Tutte le classi che creiamo ereditano, in maniera diretta o indiretta, da un'unica classe base: la classe object. La classe object fornisce sette metodi molto usati. Poiché a causa dell'eredità tutti gli oggetti che creiamo sono delle istanze della classe object, tutti i nostri oggetti avranno a disposizione questi sette metodi.

Procediamo allo studio di questi metodi.

Equals

Il metodo Equals, come suggerisce il nome, non fa altro che confrontare due oggetti. Il valore restituito è un booleano; true se i due oggetti confrontati sono uguali, false altrimenti. Il metodo viene invocato su un oggetto e prende come argomento l'altro oggetto da confrontare. Bisogna sottolineare il fatto che se due oggetti di una determinata classe devono essere confrontati, il metodo Equals andrebbe modificato (tramite sovrascrittura, override) per fare in modo che vengano confrontati i contenuti degli oggetti. Il metodo Equals di default, infatti, si limita a determinare se due riferimenti sono legati allo stesso oggetto.

Quando si effettua l'override del metodo equals bisogna tenere bene a mente i seguenti requisiti:

- Se il parametro è null bisogna ritornare false;
- Dato un oggetto obj1 allora obj1.Equals(obj1) deve essere true (riflessività);
- Dati due oggetti obj1 ed obj2, se obj1.Equals(obj2) è true [false] allora anche obj2.Equals(obj1) deve essere true [false] (simmetria);
- Dati tre oggetti obj1, obj2 ed obj3, se obj1.Equals(obj2) è true ed obj2.Equals(obj3) è true allora anche obj1.Equals(obj3) deve essere true (transitività).

Bisogna ulteriormente ricordare che quando si effettua l'override del metodo Equals bisogna effettuare l'override anche del metodo GetHashCode (di cui parleremo fra poco), per fare in modo che due oggetti obj1 e obj2 per i quali obj1.Equals(obj2) ritorna true abbiano lo stesso valore di hashcode.

Finalize

I nostri oggetti occupano uno spazio in memoria. Quando essi non sono più usati e nessuna variabile fa riferimento a loro, essi vengono rimossi dalla memoria per risparmiare spazio. L'entità software in grado di effettuare questa pulizia si chiama *garbage collector*. Il metodo Finalize non viene dichiarato o chiamato esplicitamente dal programmatore, bensì è il garbage collector che si occupa di invocarlo nel momento in cui deve liberare la memoria occupata dall'oggetto in questione. Quando una classe contiene un metodo distruttore il compilatore rinomina questo metodo affinché questo effettui l'override del metodo Finalize.

GetHashCode

La tabella hash è una particolare struttura dati in cui le informazioni vengono memorizzate tramite una coppia formata da una chiave e da un valore. Il metodo GetHashCode restituisce un numero che viene utilizzato per determinare la locazione dove verrà inserito il corrispondente valore all'interno di una tabella hash. Vedremo successivamente i vari tipi di strutture dati e gli utilizzi comuni di questo metodo.

GetType

Il metodo GetType restituisce un oggetto della classe Type che contiene informazioni relative al tipo dell'oggetto, ad esempio il nome della sua classe.

Memberwise-Clone

Il metodo restituisce un riferimento a un oggetto che è una copia dell'oggetto sul quale è stato invocato. Se l'oggetto in questione ha delle variabili di istanza, esse saranno copiate nel nuovo oggetto, il quale avrà anche gli stessi riferimenti dell'oggetto originale.

Reference-Equals

Si tratta di un metodo statico che riceve due oggetti e restituisce true se entrambi sono la stessa istanza o se sono entrambi riferimenti nulli. In caso contrario restituisce false.

ToString

L'ultimo metodo che vediamo è molto usato, esso restituisce una stringa che rappresenta un oggetto. L'implementazione di default del metodo restituisce il namespace seguito da un punto e dal nome della classe della quale l'oggetto in questione è istanza.

Le enumeration in C#

In C# le enumeration sono un utile strumento per la creazione di un tipo distinto formato da una serie di costanti definite "elenco degli enumeratori". In realtà, un'enumeration possiede un tipo sottostante, che di default è un intero (int).

Perché le enumeration

Può capitare spesso di avere a che fare con campi o variabili generiche che possono assumere, per motivi estranei alla programmazione, soltanto determinati valori predefiniti.

Si pensi al tipo boolean. Una variabile di questo tipo può assumere solo due volri: true o false. Allo stesso modo una variabile di un tipo definito tramite enumeration potrà assumere solo determinati valori.

Immaginiamo di voler rappresentare, in una variabile, i giorni della settimana: si potrebbe dichiarare questa variabile di tipo intero e far si che assuma solo valori da 1 a 7. È possibile, però, utilizzare le enumeration.

Dichiarare un'enumeration

La dichiarazione di un'enumeration avviene tramite l'utilizzo della parola chiave **enum**: `enum Giorno {`

```
    LUNEDI, MARTEDI, MERCOLEDI, GIOVEDI, VENERDI, SABATO, DOMENICA;
}
```

Come puoi vedere, il codice è molto semplice, basta inserire, nel corpo della definizione (tra le parentesi graffe), tutti i valori che l'enumeration potrà assumere, separati da virgola. Dopo l'ultimo valore, si inserisce il ';'.

Per convenzione, i valori vengono tutti rappresentati con lettere maiuscole. Questo perché possiamo immaginare una sorta di correlazione tra le enumeration e le costanti. Ogni valore, infatti, è in realtà un intero (ma è anche possibile scegliere altri tipi, a eccezione di *char)*, LUNEDI vale 0, MARTEDI vale 1 e così via. Si può anche inserire un valore diverso per ogni variabile con il simbolo '='.

Ad esempio:

```
enum Giorno {
    LUNEDI = 1, MARTEDI, MERCOLEDI, GIOVEDI, VENERDI, SABATO,
DOMENICA;
}
```

In questo modo, LUNEDI varrà 1 e non 0. Di conseguenza MARTEDI varrà 2 e così via.

Una possibile operazione da eseguire con il tipo giorno è la seguente: `if (correnteGiorno == MARTEDI) ...`

o ancora:

```
switch(correnteGiorno){
      case LUNEDI: ...
      case MARTEDI: ...
}
```

Se vogliamo, è anche possibile, come accennato in precedenza, usare un tipo diverso da *int* in questo modo: `enum Range :long {Max = 2147483648L, Min = 255L};`

Usare le enumeration

Una volta che l'enumeration è stata definita, è sempre possibile dichiarare una variabile di questo tipo e inizializzarla con uno dei valori possibili come nell'esempio che segue: `Giorno correnteGiorno = Giorno.SABATO;`

Come puoi vedere, dal momento che l'enumeration viene dichiarata fuori dal contesto della classe in cui viene utilizzata, bisogna sempre utilizzare il nome dell'enumeration deguito dal '.' prima di indicare il valore specifico.

Le eccezioni

Cos'è un'eccezione

Un'eccezione è una condizione di errore o un comportamento imprevisto riscontrato da un programma in esecuzione. Vengono generate eccezioni in caso di errori nel codice dell'applicazione o nel codice chiamato (ad esempio, in una libreria condivisa), in caso di risorse del sistema operativo non disponibili, in caso di condizioni impreviste e così via. All'interno di un'applicazione, alcune di queste condizioni possono essere superate, ma altre no. Mentre il recupero è possibile con la maggior parte delle eccezioni checked (vedi dopo), non è possibile con la maggior parte delle eccezioni a runtime.

La classe Exception rappresenta la classe base da cui ereditano le eccezioni. Gli oggetti eccezione costituiscono per la maggior parte istanze di classi derivate da Exception, ma è possibile generare come eccezione qualsiasi oggetto derivato dalla classe Object.

Dalla classe Exception derivano due grandi famiglie di eccezioni:

- Classe ApplicationException: Informazioni di riferimento per la classe da cui dovrebbero derivare tutte le eccezioni generate a livello dell'applicazione.
- Classe SystemException: Informazioni di riferimento per la classe da cui derivano tutte le eccezioni generate a livello del sistema.

Gestire le eccezioni con il costrutto try/catch

Se una chiamata di metodo può generare un'eccezione, può essere racchiusa nel blocco **try** seguito da uno o più blocchi **catch** che contengano il codice di gestione dell'eccezione: `FileInfo file = new FileInfo(percorso);`

```
Try
    {
      file.OpenWrite();
    }catch(System.IO.DirectoryNotFoundException ex){
        Console.WriteLine(ex.StackTrace));
}
```

In questo esempio troviamo la classe FileInfo vista in precedenza. Il metodo openWrtite può sollevare un'eccezione del tipo DirectoryNotFoundException che indica che non è stato possibile aprire il file perché non esiste. Questa è una tipica eccezione che non può essere prevista dal programmatore e il compilatore sa che si può verificare, quindi obbliga il programmatore a usare un blocco try-catch per gestire l'eccezione. **Catch** (cattura) serve a **catturare** l'eccezione indicata tra parentesi e a eseguire il codice indicato quando questa viene sollevata.

L'istruzione ex.StackTrace insieme al metodo WriteLine visualizza sulla console l'eccezione in questione indicando dove è stata generata e da cosa.

È possibile gestire l'eccezione in diversi modi e uno di questi è tramite il costrutto throw che è utile per rigenerare l'eccezione (dopo aver salvato lo stato del sistema per esempio) che è stata intercettata dall'istruzione catch.

```
FileInfo file = new FileInfo(percorso);
Try
{
      file.OpenWrite();
}catch(System.IO.DirectoryNotFoundException ex){
      throw(ex);
```

```
}
```

Se si desidera rigenerare l'eccezione correntemente gestita da una clausola catch senza parametri, utilizzare l'istruzione throw senza argomenti **`catch`**

```
{
	throw;
}
```

Se si vuole però catturare e gestire l'eccezione in modo tale che l'applicazione provi a eseguire dell'altro codice nel blocco catch basta semplicemente utilizzare un codice simile al seguente: **`FileInfo file = new FileInfo(percorso);`**

```
Try
{
	file.OpenWrite();
}catch(System.IO.DirectoryNotFoundException ex){
	Console.WriteLine(ex.StackTrace);
	file= new FileInfo(nuovoPercorso);
	file.OpenWrite();
}
```

In questo caso non abbiamo fatto altro che istanziare di nuovo *file* assegnandogli un percorso alternativo e infine rieseguire il metodo.

Generalmente però, si sconsiglia un simile comportamento e capiremo in seguito il perché .

Durante la programmazione dei tuoi progetti ti ritroverai spesso davanti a degli errori che dovrai subito riconoscere e gestire. Per questo vedremo adesso alcune eccezioni comuni che spesso ci fanno tribolare.

System.IndexOutOfRangeException

Questo è l'errore più comune per chi inizia a programmare. Esso viene generato

quando si tenta di accedere ad una posizione di un array o di una stringa la quale non esiste. Un esempio è questo: `int [] myArray = new int[5];`

```
for(int i=0; i<6; i++)

      System.Diagnostics.Debug.WriteLine ("" + myArray[i]);
```

(Usiamo `System.Diagnostics.Debug.WriteLine` perché così lo vedremo nella finestrina output) In questo caso è semplice notare l'errore nel for ma spesso questi errori di disattenzione o logici succedono e passano inosservati. Per fortuna che c'è il compilatore che ci avverte dell'errore ma, un buon programmatore, per sicurezza, l'avrebbe catturata con un costrutto try/catch così composto: `int [] myArray = new int[5];`

```
try

{

    for (int i = 0; i < 6; i++)
      System.Diagnostics.Debug.WriteLine("" + myArray[i]);

}catch(IndexOutOfRangeException ex){

    System.Diagnostics.Debug.WriteLine("Sei stato davvero
disattento", ex);
}
```

> Si possono utilizzare più catch per catturare diverse eccezioni, ma stai attento all'ordine in cui li inserisci. In genere è consigliato inserire prima i catch specifici e alla fine quello generale che è Exception.

NullReferenceException

Equivalente alla NullPointerException di java e viene generata quando cercate di utilizzare una variabile di riferimento nulla, alla quale non corrisponde alcun oggetto.

Esempio: Supponiamo di avere una variabile Impegno settata a null e si vuole stampare in output il luogo: `Impegno imp = null;`

```
System.Diagnostics.Debug.WriteLine(imp.Luogo);
```

Per prevenire l'eccezione basta semplicemente fare un controllo sulla variabile:

```
Impegno imp = null;

if(imp!=null)

      System.Diagnostics.Debug.WriteLine(imp.Luogo);
```

InvalidCastException

Questa eccezione viene lanciata quando si cerca di effettuare un cast su un oggetto che non corrisponde al tipo indicato. Esempio:

```
StringBuilder reference1 = new StringBuilder();

object reference2 = reference1;

StreamReader reference3 = (StreamReader)reference2;
```

È da notare che con i costrutti *is* e *as* non otteniamo questa eccezione ma semplicemente una booleana false.

ArgumentException e AgrumentNullException

Queste eccezioni vengono rispettivamente lanciate quando l'argomento passato a un metodo non è valido e quando a un metodo è stato passato un argomento nullo

Checked e Unchecked

In C# queste due parole chiave stabiliscono se l'inserimento di un valore troppo alto o troppo basso può causare la generazione di un'eccezione. Nel caso in cui il codice risulti di tipo checked, se viene inserito un valore troppo alto o troppo basso in una variabile, allora si verificherà un'eccezione. Se il codice è invece unchecked, il valore inserito verrà troncato per adattarlo alle dimensioni della variabile.

```
int maxValue=5;
```

```
checked
{
      maxValue = 6;
}
```

Questo esempio lancerà l'eccezione OverflowException.

> Per maggiori informazioni sulle eccezioni ti invito a visitare la documentazione di c# al seguente indirizzo:
>
> http://msdn.microsoft.com/it-it/library/ms173160.

Capitolo 6

I database

Cos’è un database

Esso indica un insieme di archivi (tabelle) collegati secondo un particolare modello logico in modo tale da consentire la gestione dei dati stessi (ricerca o interrogazione, inserimento, cancellazione e aggiornamento) da parte di particolari applicazioni software dedicate (**DBMS**).

Ecco un esempio di tabella di un database che descrive gli studenti di una scuola:

IDStudente	Cognome	Nome	Indirizzo	CAP	Comune	Provincia
1	Rossi	Mario	Via Po 1	10121	Torino	TO
2	Rossi	Mario	Via Po 1	10121	Torino	TO
3	Rossi	Mario	Via Po 11	10121	Torino	TO
4	Bianchi	Andrea	Via Milano 1	10122	Torino	TO
5	Bianchi	Andrea	Via Milano 1	10122	Torino	TO

È lecito chiedersi che ruolo possa avere un DBMS installato in un telefono cellulare. Le applicazioni mobili, infatti, non gestiscono grandi quantità di dati, e nemmeno devono soddisfare il requisito dell’accesso contemporaneo da parte di un elevato numero di utenti. Un’applicazione mobile, da questo punto di vista, potrebbe tranquillamente accontentarsi di salvare le proprie informazioni in uno o più file di testo (**IsolatedStorage** nel caso di Windows Phone). Perché allora si dovrebbe usare un DBMS?

Perché i DBMS

Tanto per cominciare, con un DBMS è tutto più facile. In un database, infatti, i dati possono essere strutturati e tipizzati. Si crea una tabella con tanti campi quanti sono quelli necessari, ognuno già abbinato al tipo più consono. Le operazioni di lettura e scrittura, servendosi della libreria di accesso al DBMS, necessitano ora di pochissime righe di codice.

Nelle versioni precedenti di WP non vi era alcun supporto nativo ai database relazionali e l'unico modo per salvare i dati della propria applicazione era sfruttare l'Isolated Storage e serializzare i nostri oggetti. Insomma come abbiamo fatto fino a ora.

Windows Phone 7.1 ha ovviato a questa lacuna, introducendo il supporto a SQL Server CE: si tratta di un database relazionale memorizzato su singolo file e che gira in process, perciò perfetto per essere ospitato in un file system come quello dell'Isolated Storage.

L'approccio adottato è quello di **LINQ to SQL** che è ottimo per un dispositivo mobile, più leggero e meno avido di risorse. L'approccio supportato da Windows Phone 7.1 è chiamato Code-First: il mapping tra il database e i nostri oggetti viene fatto direttamente da codice. Al primo avvio, l'applicazione creerà il nostro database, in base a una serie di convenzioni che possiamo utilizzare in fase di definizione delle nostre classi.

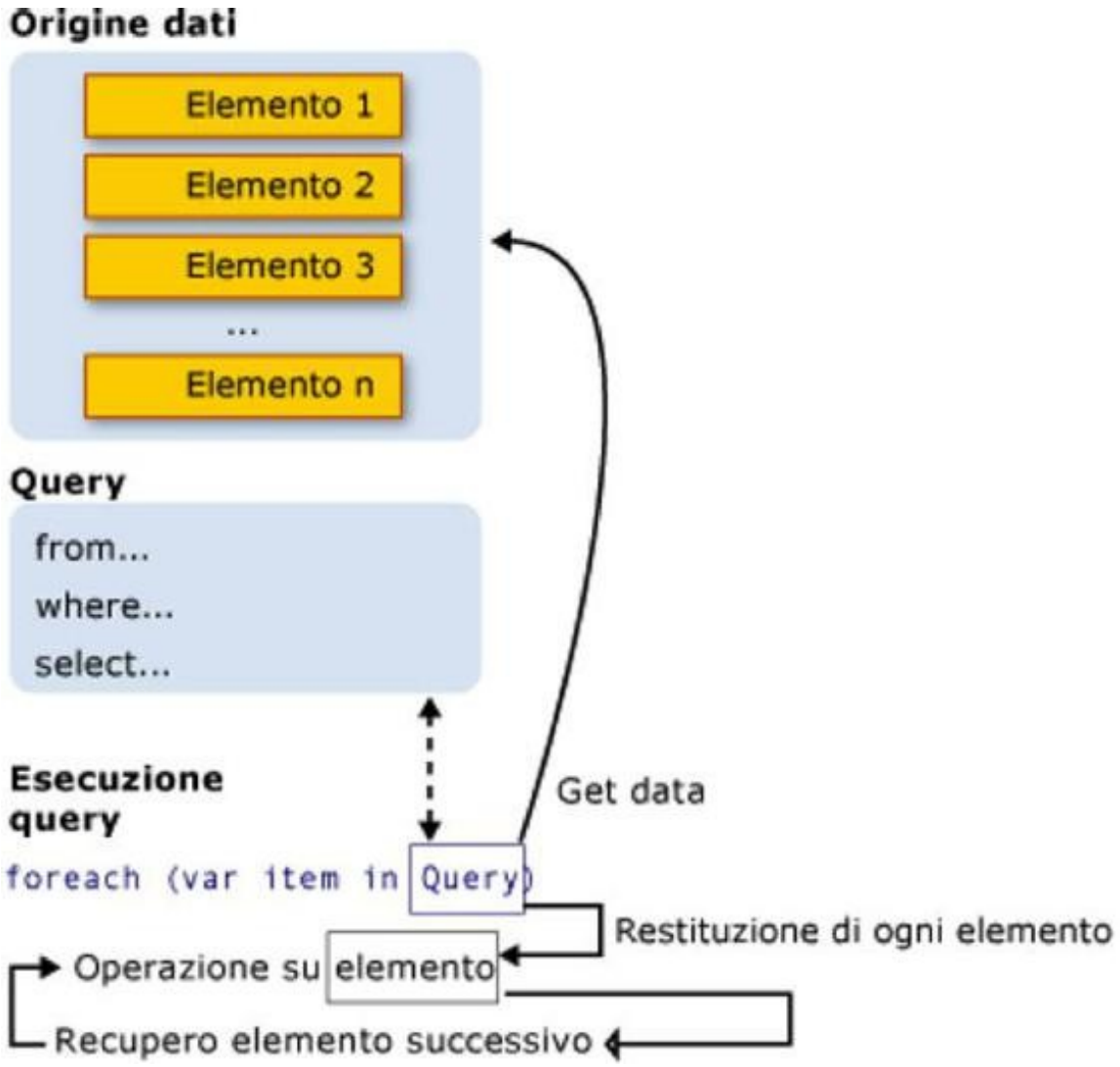
LINQ (Language-Integrated Query) colma il divario tra il mondo degli oggetti e il mondo dei dati.

Le query

Una **query** è un'espressione che recupera dati da un'origine dati ed è espressa generalmente in un linguaggio di query specializzato. Nel tempo sono stati sviluppati diversi linguaggi per i vari tipi di origini dati, ad esempio **SQL** per database relazionali e XQuery per XML. Gli sviluppatori hanno dovuto pertanto imparare un nuovo linguaggio di query per ogni tipo di origine dati o formato dati supportato. LINQ semplifica questa situazione offrendo un modello coerente per l'utilizzo dei dati con tutti i diversi tipi di origini e formati dati. In una **query LINQ** vengono utilizzati sempre gli oggetti e gli stessi modelli di codifica di base per eseguire una query e trasformare i dati in documenti XML, database SQL e qualsiasi altro formato per il quale sia disponibile un provider LINQ. Inoltre, rende una query un costrutto di linguaggio di prima categoria in C#. Quindi è possibile eseguire una query su qualsiasi tipo di struttura in cui viene supportata implicitamente l'interfaccia generica **IEnumerable<T>**: tali oggetti vengono denominati tipi **queryable**.

Struttura di una query

L’espressione di query contiene tre clausole: **from**, **where** e **select**. Se si ha dimestichezza con SQL, si sarà notato che l’ordine delle clausole è inverso rispetto all’ordine in SQL. La clausola **from** specifica l’origine dei dati, la clausola **where** applica il filtro e la clausola **select** specifica il tipo degli elementi restituiti. L’aspetto importante per il momento è che in **LINQ** la variabile di query stessa non effettua alcuna azione e non restituisce dati. Archivia solo le informazioni richieste per generare i risultati quando successivamente viene eseguita la query. Nell’immagine sottostante viene illustrata l’operazione di query.



Esempio di query che seleziona dalla tabella ImpegnoItems un impegno che abbia il tipo uguale a scuola: `from impegno in impegnoDB.ImpegnoItems`

```
where impegno.Tipo == "Scuola"

select impegno;
```

Ordinamento

È possibile ordinare i dati di una query in base a uno dei suoi attributi. È possibile utilizzare OrderBy per ordinare in modo crescente e OrderByDescending per ordinare in modo decrescente.

Vediamone un esempio:

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words

orderby word.Length

select word;

foreach (string str in query)

    Console.WriteLine(str);
```

Il risultato sarà “the fox quick brown jumps” e per ordinare in modo decrescente `orderby word.Length descending`

Aggregazione

Un’operazione di aggregazione calcola un singolo valore da una raccolta di valori.

Esempio se si vuole calcolare il max da una query : `int queryMax = evenNumberQuery.Max()`

e così anche con Min, Count e Average.

Capitolo 7

Il Model View ViewModel (MVVM)

Cos è questo MVVM e perchè?

Si tratta di un pattern la cui finalità è quella di fornire uno strato di separazione, il più possibile elevato, tra dati e strato di presentazione, in modo che al suo interno ci sia esclusivamente codice relativo alla gestione della user interface, ma non della gestione dei dati. Questo è possibile grazie al di data-binding di WPF (il toolkit di sviluppo dell'interfaccia utente) che consente a uno strato intermedio (ViewModel) che si pone tra dati (Model) e interfaccia (View) di eseguire le operazioni richieste attraverso binding di oggetti e tecniche di commanding.

M-V-VM è un pattern, quindi un insieme di linee guida che perseguono un obiettivo. Il che significa che non è "la" regola assoluta né che bisogna utilizzarlo sempre e comunque. Anzi, ci sono scenari in cui M-V-VM non è il massimo..

In particolare l'MVVM consente di:

- Favorire al massimo il paradigma di separazione del codice dall'interfaccia grafica. Lo sviluppatore non conosce l'interfaccia e mette a disposizione una serie di informazioni e funzionalità che il grafico sfrutta;
- Migliorare il testing dell'applicazione Attraverso la separazione logica/XAML;
- Ottimizzare il riuso del codice.

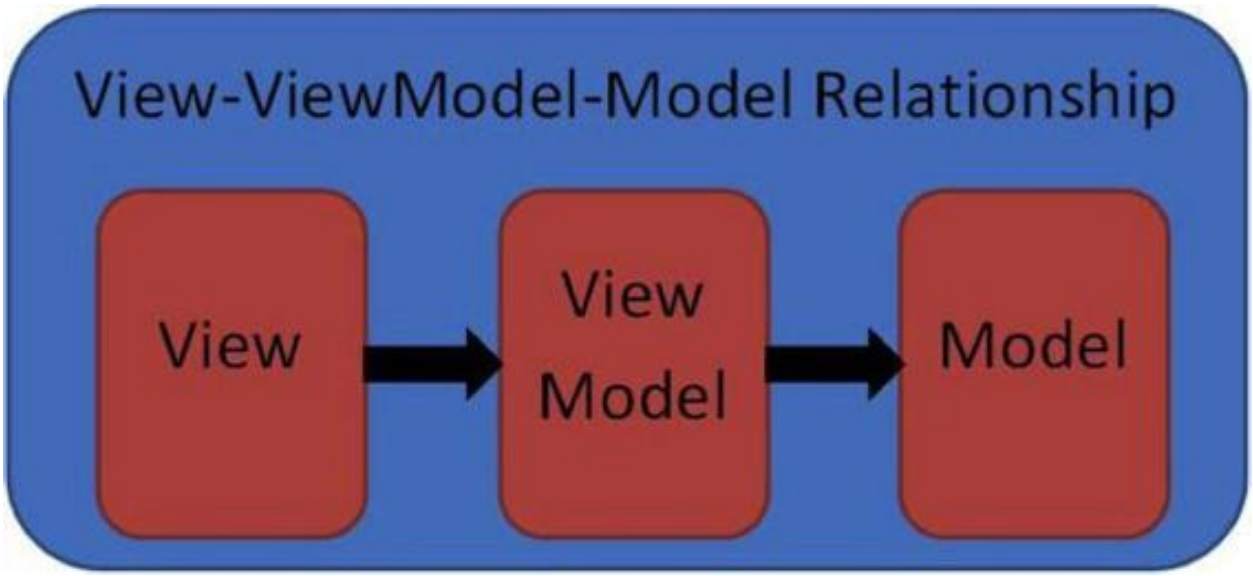
I dettagli

Model-View-ViewModel (MVVM) pattern divide il codice di interfaccia utente in 3 parti concettuali - Model, View e ViewModel.

Model è un insieme di classi che rappresentano i dati provenienti dai servizi o dal database.

View è il codice corrispondente alla rappresentazione visiva dei dati (in pratica il codice scritto in XAML).

ViewModel funge da collante tra la view e il model. Racchiude i dati del modello e lo rende “amichevole” per essere presentato e modificato dalla view. Inoltre controlla anche le interazioni della view con il resto dell’applicazione.



Nel grafico si mostrano le tre parti del pattern e le frecce mostrano quali parti dell’ MVVM sono a conoscenza dell’altro. Utile quindi per l’approccio alla programmazione del nostro esempio.

Le collections in C#

Cosa sono

Le *collections* (letteralmente collezioni) sono classi utilizzate per raggruppare e gestire oggetti correlati, permettendo anche di effettuare iterazioni su di essi, e rappresentano uno strumento fondamentale per i programmatori. Si può dire che tali classi rappresentino un livello superiore rispetto agli array, che sono utili ma non forniscono molte funzionalità avanzate come il ridimensionamento

automatico *etc.*

Al giorno d'oggi le applicazioni si trovano a gestire grandi quantità di dati e spesso si presenta l'esigenza di memorizzare tali dati secondo un certo ordine. Il .NET Framework risponde a questa esigenza fornendo una grande varietà di collections da utilizzare a questo scopo.

Il *namespace* che supporta i vari tipi di collections è il **System.Collections** e tutte le collezioni implementano l'interfaccia *IEnumerable* che viene estesa da *ICollection* importante per potervi utilizzare LinQ ed eseguire quindi iterazioni. Le collezioni si possono dividere in tre grandi gruppi: liste, dizionari e code.

Le liste

Nell'ambiente di programmazione per Windows Phone è possibile utilizzare solo la classe List (che sarebbe una ArrayList) e LinkedList.

La classe **List** è un semplice contenitore, non ordinato di oggetti di vario tipo e aggiungere o eliminare oggetti da esso è molto semplice. In particolare per aggiungere un elemento alla collezione possiamo utilizzare **Add** che aggiunge un singolo oggetto e **AddRange** che aggiunge alla collezione un insieme di oggetti.

Inoltre poichè gli oggetti List sono collezioni dinamiche essi supportano anche l'inserimento di oggetti in posizioni specifiche. Per fare ciò è possibile utilizzare i metodi **Insert** e **InsertRange**.

Per eliminare gli elementi è possibile utilizzare tre metodi: **Remove**, **RemoveAt** e **RemoveRange**. Il primo rimuove un oggetto specifico, il secondo rimuove un oggetto con un particolare indice all'interno della collection, mentre il terzo permette di rimuovere un insieme di oggetti con determinati indici.

La classe List supporta anche vari altri utili metodi, tra cui:

- **Clear**: utilizzato per svuotare una collezione;
- **IndexOf**: utilizzato per ottenere l'indice di un particolare oggetto della collezione;
- **Contains**: utilizzato per verificare se un particolare oggetto sia presente all'interno della collezione;
- **Sort**: ordina gli elementi.

I dizionari

Diametralmente opposti alle liste sono i cosiddetti **dizionari**, ossia collezioni pensate per memorizzare liste di coppie chiave/valore e per permettere la successiva ricerca di valori tramite le chiavi corrispondenti.

Le classi **Dictionary** del .NET Framework vengono utilizzate proprio a tale scopo. Il metodo **Add** richiede come parametri una chiave e un valore. Quindi, diversamente dalle collezioni viste in precedenza, i dizionari richiedono sempre due entità: la chiave e il corrispondente valore. Tutte le classi di dizionari del .NET Framework supportano l'interfaccia IDictionary che deriva dall'interfaccia *ICollection* e derivando da quest'ultima contiene i metodi base **add**, **remove**, **contains** e **clear** visti precedentemente.

Le code

La classe **Queue** è una classe che permette di gestire strutture sequenziali di tipo **FIFO**. In particolare combina le due operazioni di accesso e rimozione degli elementi in un unico metodo chiamato **Dequeue** ed è possibile utilizzare il metodo **Enqueue** per aggiungere un elemento.

In pratica, proprio come in una coda, quando viene inserito un elemento esso viene posto alla fine della collezione, mentre quando ne viene rimosso uno è il primo della lista ad essere rimosso. Un funzionamento diverso dalla classe Queue ha la classe **Stack**, che permette di gestire strutture sequenziali di tipo LIFO.

In una collezione Stack è possibile aggiungere e rimuovere elementi all'inizio e alla fine tramite le operazioni di push e pop, ma non è possibile scorrere la collezione per accedere a un determinato elemento. È possibile quindi operare solo sugli elementi finali.

Classe List

Come detto in precedenza, la classe **List** è un semplice contenitore, non ordinato, di oggetti di vario tipo e aggiungere o eliminare oggetti da esso è molto semplice. In pratica è un array dinamico su cui si possono effettuare ricerche, inserimenti e rimozioni e tutto con un ridimensionamento automatico. Vediamo in dettaglio l'uso di alcuni metodi citati nella lezione precedente. Innanzitutto

viene istanziata con il comando new, come qualunque altro oggetto `List<string> words = new List<string>();` ed è possibile anche definirne le dimensioni iniziali inserendo all'interno delle parentesi tonde un intero. È possibile accedere a un valore della lista tramite il nome della lista seguita dalle parentesi quadre al cui interno va inserito l'indice di accesso all'elemento:

```
words[2] = "prova"; //assegnazione
string s = words[1];// ricerca
```

Tramite il metodo Add si aggiunge alla lista un elemento e con Remove si cerca l'elemento e poi viene rimosso: `words.Add("aggiungo string");`

```
words.Remove(stringa);
words.RemoveAt(2);
```

Per cercare degli oggetti in una lista ci sono diversi modi e il più semplice è tramite il metodo *Contains*. Però, per essere utilizzato su un oggetto da noi creato, quest'ultimo deve implementare l'interfaccia *IEqualityComparer* e quindi il metodo *Equals*. Presupposto questo, fare la ricerca è davvero semplice: `bool esiste = words.Contains("hello");`

Ciò restituisce un valore booleano che in caso di un risultato positivo sarà true altrimenti false. Se volessimo però l'indice in cui si trova l'elemento dobbiamo utilizzare il metodo *IndexOf*: `int indice = words.IndexOf("hello");`

Esiste anche il metodo *Sort* per poter ordinare una lista, ma per poterlo fare l'oggetto deve implementare l'interfaccia *IComparable* e quindi realizzare il metodo *CompareTo* per poter dare un ordinamento tra gli elementi. String implementa da sola anche questa funzionalità e quindi la chiamata sarà `Words.sort();`

LinkedList

È la struttura dati più usata nella programmazione e oltre ad avere gli stessi metodi di List (da cui essa deriva) ne aggiunge altri. Questo perché la linkedlist è una lista linkata, cioè ogni elemento è collegato sia al suo predecessore che al suo successore e quindi attraverso essa si posso emulare code(AddLast , RemoveFirst) e pile (AddLast, RemoveLast) .

I Dictionary

La classe generica Dictionary fornisce un'associazione da un gruppo di chiavi a un gruppo di valori. Ogni aggiunta al dizionario è costituita da un valore e dalla relativa chiave e il recupero di un valore tramite la relativa chiave è un'operazione molto veloce poiché la classe Dictionary viene implementata come tabella hash. Questo tipo di struttura dati è ideale per tutti quei casi in cui è necessario gestire con facilità la memorizzazione e la ricerca di dati, come avviene esattamente per una rubrica telefonica o una mailing list.

Essa viene istanziata in questo modo: `Dictionary<Key, TValue> dict = new Dictionary<Key, TValue>();`

Se un oggetto viene utilizzato come chiave nella classe *Dictionary*, non deve essere modificato e ogni chiave deve essere univoca.

Possiamo inserire una chiave e un valore tramite il metodo *add* e prendendo come esempio un dizionario con chiave *int* e valore *string* faremo: `dict.Add(1, "Marco");`

Possiamo inserire gli elementi anche direttamente, specificando la chiave ed associandogli il valore, in questo modo: `dict[1] = "marco";`

In questo caso però, se la chiave non esiste crea un nuovo elemento, se la chiave esiste sovrascrive il valore esistente, se invece provassimo a inserire un nuovo valore per una chiave esistente con il metodo *Add* otterremo un'eccezione.

Per cancellare un elemento invece basta richiamare il metodo *Remove*, specificano la chiave `dict.Remove(1);` o eliminare tutto tramite *clear*: `dict.clear();`

Esistono tutta una serie di altri utili metodi per copiare, spostare, contare gli elementi, controllare se esiste una chiave o un valore, che non staremo qui a elencare in quanto sono di facile ed intuitivo utilizzo richiamandoli semplicemente con l'intellisense di Visual Studio.

Quello che invece vogliamo mettere in evidenza è la facilità con cui possiamo ricavare una lista di chiavi e valori ed effettuare delle ricerche all'interno dei Dictionary, proprio perché indicizzate, basta specificare la chiave per ottenere subito l'elemento.

Per ricavare la lista delle chiavi si utilizza il metodo *Keys* `var lista = dict.Keys;` e per ottenere i valori `var lista = dict.Values;`

Per effettuare delle ricerche basta fare **`string stringa = dict[1];`** utilizzando la chiave, o anche similmente utilizzando il valore.

La ricerca degli elementi nelle Dictionary a mezzo indice è immediata, presenta infatti una complessità costante. Lo svantaggio presentato dalle Dictionary si evidenzia invece nell'ordinamento, poiché gli elementi non sono ordinati in alcun mono ed è previsto un ordinamento solo per chiave e non per valore.

I tipi generici

I generici sono una funzionalità della versione 2.0 del linguaggio C# e introducono il concetto di parametri di tipo, grazie ai quali è possibile progettare classi e metodi che rinviano la specifica di uno o più tipi fino a quando un codice che li utilizza non ne dichiara o crea un'istanza.

I generici, quindi, rappresentano la soluzione a una limitazione presente nelle versioni precedenti di C#, in cui la generalizzazione viene effettuata mediante il cast di tipi in e dal tipo base universale Object.

Esistono anche le classi generiche che consentono invece di creare un insieme indipendente dai tipi in fase di compilazione.

Essi semplificano di molto la vita del programmatore perché, senza di questi, dovremmo avere operazioni di casting ogni qual volta dovesse servirci un oggetto da una collezione. Perciò, il concetto dietro ai generici è questo: dichiarare prima (ovvero in fase di dichiarazione della collection) quale sarà il tipo attuale di ogni dato della stessa, in modo tale che il compilatore possa costruire una collezione di oggetti specifici e risparmiare tempo per l'accesso ad ogni elemento.

Come si può notare, tutte le classi e interfacce del framework delle Collection sono parametriche (usano i generici), questo per dare la massima libertà al programmatore di utilizzare qualsiasi tipo di dato all'interno della struttura dati.

Prendiamo ad esempio una LinkedList<Giorno> parametrizzata con l'oggetto Giorno di Agenda. Se provassimo a inserire all'interno una String o qualsiasi altro tipo di oggetto il compilatore ci darebbe un errore sull'operazione:

```
LinkedList<Giorno> giorni = new LinkedList<Giorno>();

String s = "Ciao";

giorni.AddFirst(s);
```

Questo perché esso ci avverte che il tipo di dato inserito non coincide.

Creazione di una semplice classe generica

Le classi generiche risultano utili quando sappiamo che dovremo avere a che fare con un altra classe specifica, ma lo sapremo solo al momento in cui dichiareremo una variabile.

Quindi dichiarare una classe generica è semplice. Basta aggiungere un nome convenzionale alla nostra classe (ma anche più di una, separate da virgole) dopo il nome della classe: **`public class Cavani<Tipo>{`**

```
    public static int trovaIndice(Tipo oggetto, Tipo [] oggetti) {
      for(int i=0;i < oggetti.Length;i++){
        if (oggetto.equals(oggetti[i]))
          return i;
      }
    return -1;
    }
}
```

Come l'esempio banale ci mostra, abbiamo creato una semplice classe che ha un metodo statico, il quale trova l'indice in cui è memorizzata la prima occorrenza di un oggetto all'interno di un array, tutto questo non sapendo di che tipo è l'oggetto in questione.

È importante sottolineare che non solo le classi possono essere parametriche, ma anche le interfacce come IEquatable, IComparable ecc..

Collection parametriche personali

Uno studio che richiede più attenzione, col quale si entrerebbe nei dettagli che non sono interessanti al fine del manuale, è quello su come creare una lista parametrica personalizzata e quindi non utilizzando quelle già scritte dagli sviluppatori di C#. Quindi, per chi vuole approfondire l'argomento si consiglia di leggere la documentazione messa a disposizione da Microsoft sul sito: http://msdn.microsoft.com/it-it/library/512aeb7t.aspx

Cos’è un iteratore

Abbiamo usato spesso l’iteratore anche se in una sua versione “invisibile”. Infatti il costrutto *foreach* ne fa un uso nascosto consentendo al programmatore una maggiore velocità nella scrittura del codice. Prendiamo come esempio un *foreach* su una Lista di stringhe: `List<string> lista = new List<string>() { "te", "me", "il mondo" };`

```
...

foreach(string elemento in lista)

{

      Console.WriteLine("elemento in: {0}!", elemento);

}
```

Si può notare come l’accesso agli elementi divenga veramente semplice e intuitivo, ma ne è comodo l’utilizzo solo quando non bisogna apportare modifiche alla struttura dati. Inoltre, per poter utilizzare tale ciclo, la classe su cui viene eseguito deve estendere IEnumerator (tutte le Collection la estendono) e quindi devono provvedere all’implementazione del metodo GetEnumerator il quale ritorna un iteratore. Infatti, dietro il foreach in realtà vi è questo codice: `var it = lista.GetEnumerator();`

```
while (it.MoveNext())

{

      string elemento = it.Current;

      Console.WriteLine("elemento in: {0}!", elemento);

}
```

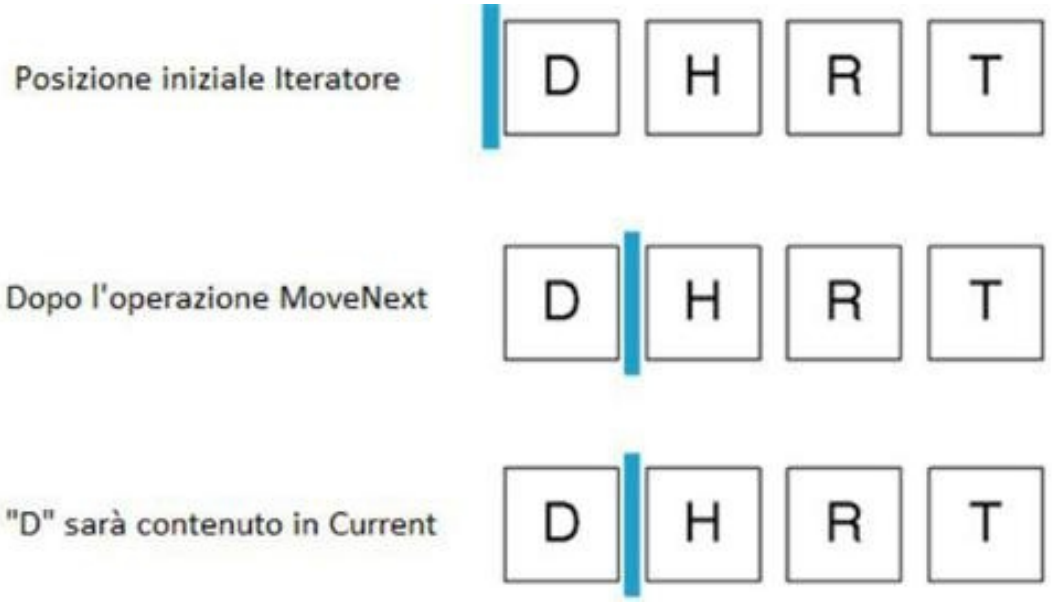
Nella prima riga si associa alla variabile *it* l’iteratore della lista e da questo punto in poi è possibile utilizzare su tale variabile questi metodi:

- MoveNext e Current :MoveNext ha la stessa funzionalità del metodo hasNext di Java. Infatti restituisce una booleana che sarà true se esiste un elemento su cui poter effettuare un operazione e lo faccia ritornare per essere poi chiamato dal metodo Current. Quindi Current non può essere

invocato se prima non è stato chiamato MoveNext poiché essi sono strettamente legati.

- Equals:Consente di controllare se l'oggetto nell'iteratore è uguale a un altro.
- Reset: fa partire dall'inizio l'iteratore, cioè prima del primo elemento della Collection.

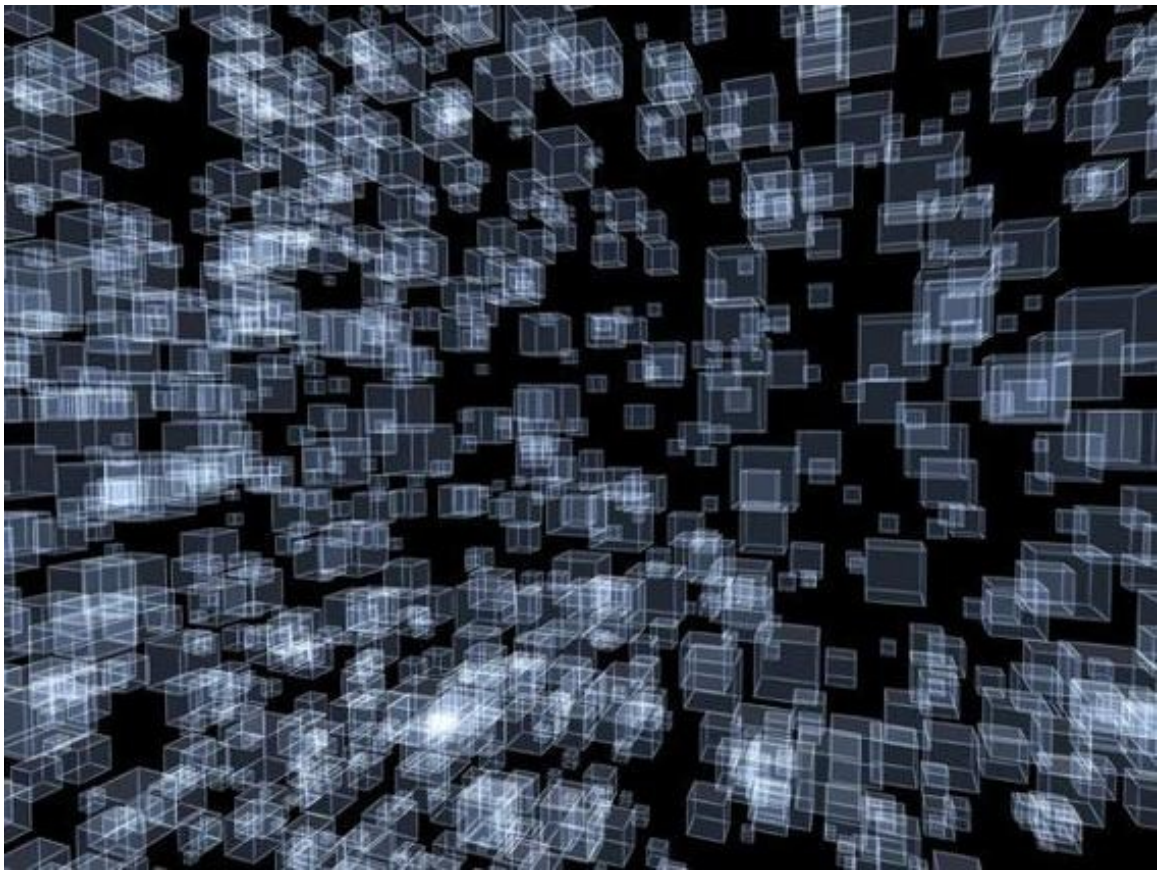
Ecco un esempio di come il puntatore (la riga blu) si sposta all'interno di una lista:



A differenza di Java gli iteratori in C# non hanno il metodo remove, per cui per poter eleminare un elemento da una struttura dati bisogna utilizzare uno dei loro metodi. Nel nostro esempio per eliminare un elemento dalla lista possiamo fare così: **`var it = lista.GetEnumerator();`**

```
while (it.MoveNext())
{
    string elemento = it.Current;
    if(elemento.equals("stringa da rimuovere")
      lista.remove(elemento);
}
```

Capitolo 8



Introduzione a Xaml

XAML (acronimo di *eXtensible Application Markup Language*) è un linguaggio dichiarativo, sviluppato da Microsoft, atto alla rappresentazione di grafi di oggetti, molto adatto per creare interfacce utente.

In XAML dichiarare un elemento equivale a creare un'istanza, dichiarare elementi annidati equivale ad aggiungere istanze alla collezione di oggetti figlio, mentre valorizzare un attributo equivale a impostare una proprietà.

Vista la sua natura general-purpose, attualmente XAML è utilizzato in **Silverlight**, **WPF** e **WF**, in ognuno con il proprio parser e runtime, questo perchè XAML è composto solo da una serie di regole e parole chiave che indicano, sia al parser, sia al compilatore, come trattare l'XML, quindi il linguaggio da solo non offre nessuna funzionalità, ma necessita di un runtime per esprimersi.
Comunque il pregio di XAML è che consente un flusso di lavoro in cui parti distinte possono operare nell'interfaccia utente e nella logica di un'applicazione, utilizzando strumenti potenzialmente diversi.

Sintassi

La sintassi degli elementi inizia sempre con una parentesi angolare di apertura (<), seguita dal nome del tipo in cui si desidera creare un'istanza. Se si desidera è possibile dichiarare attributi o proprietà (sono equivalenti) nell'elemento e per completare il tag si deve terminare con una parentesi angolare di chiusura (>). In alternativa, è possibile utilizzare una forma di chiusura automatica senza contenuto, completando il tag con una barra e una parentesi angolare di chiusura in successione (/>). Osserviamo un semplice esempio che crea un Button:

```
<StackPanel>
    <Button Content="Cliccami" Background="Blue" Foreground="Red"/>
</StackPanel>
```

All'interno di Button sono state definite delle proprietà che è possibile ridefinire

anche in questo modo:

```
<Button>
    <Button.Background>
      <SolidColorBrush Color="Blue"/>
    </Button.Background>
    <Button.Foreground>
      <SolidColorBrush Color="Red"/>
    </Button.Foreground>
    <Button.Content>
      Cliccami
    </Button.Content>
</Button>
```

Quest'ultima sintassi viene utilizzata se si desidera effettuare particolari modifiche alle proprietà le quali non si possono effettuare semplicemente passando una stringa.

Namespace

XAML, come tutti i linguaggi derivati da XML, eredita il concetto di spazio dei nomi. Solitamente un file XAML è composto da almeno due **namespace**, il primo, posto sull'elemento principale, definito tramite l'attributo xmlns è il namespace:
`xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"`

Viene utilizzato per validare la struttura e il contenuto del file XAML.

Il secondo, convenzionalmente definito con il prefisso x e posto anche questo sull'elemento principale, è il namespace: `xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"`

Questo namespace ci permette di aggiungere una serie di direttive utili al parser XAML, come l'elemento x:Class che specifica il nome della classe di riferimento nel code-behind.

È possibile, come abbiamo visto, creare namespace personali per referenziare classi contenute nelle nostre librerie, la sintassi è semplice ed è così composta: `xmlns:PrefissoCustomNamespace="clr-namespace:NamespaceLibreria;assembly=Libreria"`

Tramite la direttiva assembly impostiamo la libreria da referenziare, mentre con la direttiva clr-namespace impostiamo il namespace della libreria. Vediamo un esempio: `xmlns:myControl="clr-namespace:ControlloPersonalizzato;assembly=ControlloPersonalizzato"`

Scrivere un elemento in XAML è un po' come istanziare un oggetto in .NET tramite il costruttore privo di parametri, mentre impostare un attributo equivale ad impostare una proprietà dell'oggetto.
Quest'ultimi contengono valori di tipo stringa, racchiusi tra singolo (') o doppio (") apice. Con l'attributo **x:Name** impostiamo il nome dell'elemento XAML e questo ci permetterà di richiamare tale controllo nel codice scritto nel code-behind. Sempre grazie agli attributi possiamo definire un event handler (listener o gestione evento) per un evento esposto dall'elemento come Click in un Button.

Prefisso x

Il prefisso x: viene utilizzato per il mapping del namespace principale di xaml e viene usato nei modelli per i progetti, negli esempi e nella documentazione in tutto l'SDK. Tre sono i principali prefissi che la utilizzano:

- `x:Key`: imposta una chiave univoca per ogni risorsa.
- `x:Class`: specifica lo spazio dei nomi CLR e il nome della classe che fornisce il code-behind per una pagina XAML.
- `x:Name`: imposta il nome dell'elemento.

Markup Extensions

Markup Extension è una particolare caratteristica che permette di estendere l'espressività di XAML. Se utilizzate per specificare il valore di una sintassi per attributi, le parentesi graffe ({ e }) indicano l'utilizzo di un'estensione di

markup. Questo utilizzo indica all'elaborazione XAML che, diversamente dal solito, i valori degli attributi non devono essere considerati valori di stringa letterale o valori convertibili in stringa.

Vediamo un frammento di codice XAML che mostra un esempio di utilizzo di una Markup Extension:

```
<Button Content="{StaticResource test}"/>
```

Quando il Markup Extension verrà processato a runtime, restituirà un riferimento alla risorsa richiesta, precedentemente istanziata.

In Xaml possiamo trovare i seguenti Markup Extension:

- **`Binding`**: utilizzato per il supporto al Data Binding.
- **`StaticResource`**: per ottenere risorse contenute negli apposti ResourceDictionary.
- **`TemplateBinding`**: utilizzato nella definizione del Template di un controllo.
- **`x:Null`**: rappresenta il valore null in XAML.

Canvas

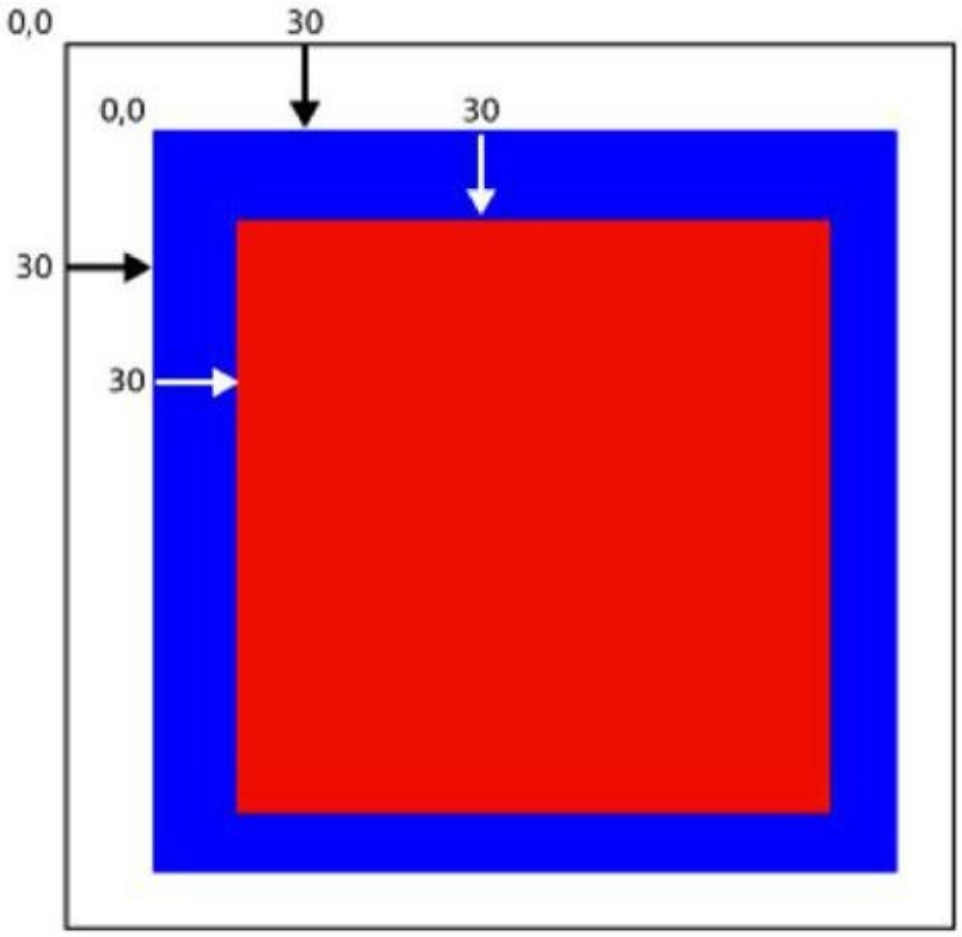
Quando si crea un'applicazione e quindi l'interfaccia bisogna definire gli spazi da assegnare alla grafica e ai controlli. I Layout Controls sono utilizzati per questo scopo e ne esistono di diversi. Andiamo a vedere i Canvas. Esso è il più primitivo dei pannelli e permette di posizionare gli elementi figlio in maniera assoluta tramite Canvas.Left e Canvas.Top, che rappresentano le coordinate del controllo all'interno della superficie, indicando rispettivamente la distanza dal bordo sinistro e da quello superiore. Possiamo identificare un Canvas come un contenitore di elementi e anche lui deriva da un UIElements.

Vediamo un semplice esempio di due Canvas annidati e l'uso delle proprietà basilari:

```
<Canvas Width="300" Height="300" Background="White">
    <Canvas Width="250" Height="250" Canvas.Left="30"
            Canvas.Top="30" Background="blue">

        <Rectangle Canvas.Left="30" Canvas.Top="30"
                   Fill="red" Width="200" Height="200" />
    </Canvas>
</Canvas>
```

Il risultato sarà questo:



Dependency Property

La natura dichiarativa di XAML, ha portato gli sviluppatori che hanno realizzato Windows Presentation Foundation a privilegiare le proprietà degli oggetti rispetto ai metodi, arrivando a rappresentare con esse anche i comportamenti e gli eventi. Per questo motivo, a supporto delle proprietà è stato creato il Dependency Property System, che ha l'onere di conferire a dei particolari tipi di proprietà, denominate DependencyProperty, una serie di caratteristiche. Comunque l'obiettivo è di permettere funzionalità avanzate direttamente dall'interno del linguaggio di markup, senza bisogno di impostare le proprietà con codice procedurale.

È possibile creare una proprietà di dipendenza a un qualsiasi oggetto che derivi da DependencyObject, vale a dire tutti gli elementi che derivano da UIElement. Ci sono tre passaggi fondamentali per implementare una Dependency Property:

1. Definire un membro statico pubblico di tipo System.Windows.DependencyProperty e registrare tale proprietà nel costruttore statico della classe attraverso una chiamata al metodo DependencyProperty.Register, anche esso statico, il quale richiede come parametri, il nome, il tipo della proprietà, il tipo della classe che la espone e il metodo di callback che viene invocato quando il valore della proprietà cambia. Ecco un esempio in cui definiamo una proprietà CustomText:

```
public static readonly DependencyProperty CustomTextProperty = DependencyProperty.Register(
      "CustomText",          //Nome
      typeof(string),        //Tipo
      typeof(MainPage),      //Proprietario Proprietà
       new PropertyMetadata("test", new PropertyChangedCallback(MainPage.OnCustomTextPropertyChanged)) //Callback
);
```

2. Definire un metodo il cui nome corrisponde al nome del campo della proprietà di dipendenza, meno il suffisso Property, a cui corrisponde anche il parametro nome del metodo Register. Utilizza i metodi SetValue() e GetValue() ereditati dall'oggetto DependencyObject, per assegnare e leggere il valore. Nell'immagine continuiamo con l'esempio:

```
public string CustomText
{
    get { return (string)GetValue(CustomTextProperty); }
    set { SetValue(CustomTextProperty, value); }
}
```

3. Questo passaggio è opzionale se nel campo CallBack di Register si è inserito null. Nel nostro esempio dopo che viene inserito un testo all'interno della textbox facciamo visualizzare un messaggio.

```
private static void OnCustomTextPropertyChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
{
    MessageBox.Show("ciao");
}
```

E nel file xaml avremo **`<TextBox Text="{Binding CustomText, Mode=TwoWay}"`**

Il binding deve essere twoway poichè si deve attivare l'evento quando il testo cambia.

Se volete approfondire l'argomento:
http://msdn.microsoft.com/en-us/library/system.windows.dependencyproperty.aspx

Attached Property

Per alcuni elementi, esistono delle proprietà che non sono relative all'oggetto stesso, queste classi conferiscono proprietà agli oggetti figlio, chiamate Attached Properties.

Sono una forma particolare di Dependency Property e vengono definite nella solita modalità, ma non dispongono di proprietà wrapper per facilitarne l'utilizzo e hanno una sintassi speciale in XAML: **`ParentElement.PropertyName`**

Un esempio pratico in xaml di attached property è per esempio Grid.Column, Canvas.Left etc…Esse sono un meccanismo "furbo" per arricchire elementi con proprietà che, sebbene memorizzate nell'oggetto stesso, sono definite in un altro elemento.

In molte tecnologie (come xml), gli elementi e i componenti sono organizzati in una struttura ad albero in cui gli sviluppatori modificano direttamente la struttura ad albero per influire sul rendering o sul comportamento di un'applicazione.
Il Visual Tree (chiamato anche Object Tree) è la rappresentazione concettuale dell'interfaccia grafica, la quale è composta da una serie di elementi che possono

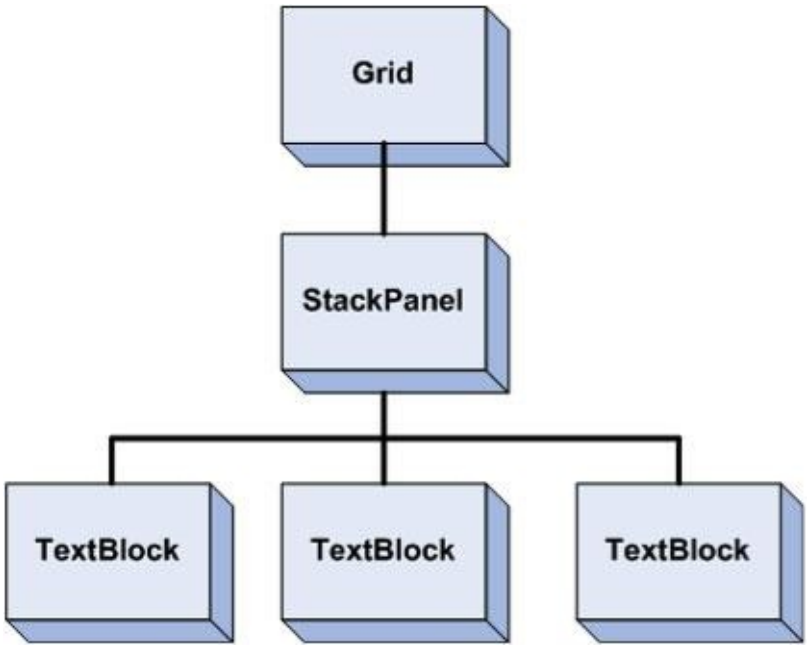
avere a sua volta uno o più oggetti figlio relazionati fra loro. Il risultato è una rappresentazione ad albero che permette l'accesso all'Object Tree tramite code-behind.

Quindi nel linguaggio XAML, sono disponibili elementi, ad esempio <Button> e <Grid>, che possono avere altri elementi (nodi) sotto di essi (figli). Questa relazione padre/figlio consente di specificare il modo in cui gli oggetti sono posizionati sullo schermo e le relative risposte agli eventi avviati dall'utente. Ecco un esempio:

```
<Grid x:Name="ContentPanel" Background="Red" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel Margin="20" Background="Blue" >
      <TextBlock Name="primaTextBlock" FontSize="30">Prima TextBlock</TextBlock>
      <TextBlock Name="secondaTextBlock" FontSize="30">Seconda TextBlock</TextBlock>
      <TextBlock Name="terzaTextBlock" FontSize="30">Terza TextBlock</TextBlock>
    </StackPanel>
</Grid>
```

L'elemento StackPanel è contenuto all'interno di un elemento Grid. Gli elementi TextBlock sono contenuti all'interno dell'elemento StackPanel (questi elementi TextBlock sono figli di StackPanel). Inoltre, gli elementi TextBlock sono impilati uno sopra l'altro nell'ordine in cui vengono dichiarati nel codice XAML.

Questo diagramma della struttura ad albero mostra le relazioni tra gli elementi.



Oltre a determinare il modo in cui il contenuto viene presentato, la struttura ad albero visuale può avere effetti sul modo in cui gli eventi vengono elaborati. Molti eventi correlati all'interfaccia utente propagano gli eventi verso l'alto nella struttura ad albero. Puoi ad esempio associare un gestore di eventi all'elemento StackPanel che gestisce quando l'utente preme o fa clic su uno degli oggetti TextBlock. Ecco come aggiungere un gestore di eventi PointerPressed denominato "commonHandler" all'elemento StackPanel: **`<StackPanel Margin="20" Background="Blue" PointerPressed="commonHandler">`**

E il codice C# sarà:

```
private void commonHandler(object sender, PointerRoutedEventArgs e)
{
FrameworkElement feSource = e.OriginalSource as FrameworkElement;
switch (feSource.Name)
{
    case "primaTextBlock":
      primaTextBlock.Text = primaTextBlock.Text + " Click!";
      break;
```

```
}
}
```

Quando esegui questo esempio e premi o fai click su l'elemento primaTextBlock, l'evento viene acquisito da TextBlock e successivamente si propaga all'elemento padre (StackPanel) che gestisce l'evento.

Poiché l'evento continua a propagarsi verso l'alto nella struttura ad albero, puoi ascoltare l'evento PointerPressed anche nell'elemento Grid.

L'**animazione** è un'illusione creata attraverso una serie di immagini ognuna leggermente diversa dalla precedente. Il cervello percepisce il gruppo di immagini come il cambiamento di una singola scena. Nei film, ad esempio, questa illusione viene creata utilizzando telecamere che registrano numerosi fotogrammi, ogni secondo. Quando i fotogrammi vengono riprodotti da un proiettore, il pubblico vede un film.

L'animazione su computer è simile. Ad esempio, un programma che fa un disegno di un rettangolo che aumenta automaticamente di dimensione potrebbe funzionare come segue:

- Il programma crea un timer.
- Il programma controlla il timer a intervalli regolari per vedere quanto tempo è trascorso.
- Ogni volta che il programma controlla il timer, si calcola il valore della dimensione per il rettangolo in base a quanto tempo è trascorso.

Il programma quindi aggiorna il rettangolo con il nuovo valore e lo ridisegna.

Silverlight e Xaml ci danno la possibilità ti poter effettuare tali animazioni su degli oggetti direttamente all'interno del codice Xaml utilizzando pochissimo codice C#. Possiamo considerare questo tipo di animazioni come un modo semplice per modificare il valore di una *Dependecy Property* in un certo intervallo di tempo.

Le caratteristiche fondamentali di un'animazione sono:

- la durata, espressa come intervallo di tempo (ore, minuti, secondi);
- il tipo di dato della proprietà sulla quale agire;

- i valori iniziale e finale.

Ad esempio per poter modificare delle *Dependency Property* di tipo double (come Height o Width) utilizziamo l'oggetto *DoubleAnimation* che anima l'elemento:

```
<DoubleAnimation From="150" To="300" Duration="0:0:3"
   Storyboard.TargetName="MioTarget"
   Storyboard.TargetProperty="MioTargetProprietà" />
```

Con questo si modificherà la proprietà del target da 150 a 300 in 3 secondi.

Per manipolare tipi come Color e Point esistono anche *ColorAnimation* e *PointAnimation*, mentre per tutti gli altri è necessario crearne una personalizzata.

In particolare questi tipi di animatione possiedono queste proprietà:

- **From**: Il valore iniziale per l'animazione.
- **To**: La fine del valore per l'animazione.
- **By**: Un valore relativo che indica di quanto cambiare l'animazione (un approccio alternativo all'utilizzo di To).
- **Duration**: La durata di un animazione, utilizzando la sintassi "hh: mm: ss" (ore, minuti, secondi).
- **AutoReverse**: Se inverte l'animazione quando è finita (ad esempio, reimposta la dimensione originale del controllo).
- **RepeatBehavior**: Cosa fare quando l'animazione è terminata; è possibile indicare una durata totale, un certo numero di ripetizioni o se l'animazione deve ripetere all'infinito.

Per poter applicare l'animazione a un oggetto dobbiamo introdurre lo StoryBoard che fornisce gli strumenti per poter applicare l'animazione alla proprietà. Infatti ci mette a disposizione le proprietà TargetName e TargetProperty che definiscono su chi e cosa applicare l'animazione. Quindi, se volessimo cambiare il colore e la dimensione di un rettangolo a un evento DoubleTap avremmo questo codice:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Resources>
        <Storyboard x:Name="MyStoryboard">
            <DoubleAnimation AutoReverse="True"  From="150" To="300" Duration="0:0:3"
               Storyboard.TargetName="MioRettangolo"
               Storyboard.TargetProperty="Width" />

            <ColorAnimation From="Green" To="Orange" Duration="0:0:3"
              Storyboard.TargetName="MyBrush"
              Storyboard.TargetProperty="Color" />
        </Storyboard>
    </Grid.Resources>
    <Rectangle x:Name="MioRettangolo"
     Canvas.Left="25" Canvas.Top="40"
     Width="150" Height="50"
     Margin="6,6,0,712"
     HorizontalAlignment="Left"
     DoubleTap="MyRectangle_DoubleTap">
        <Rectangle.Fill>
            <SolidColorBrush x:Name="MyBrush" Color="Green" />
        </Rectangle.Fill>
    </Rectangle>
</Grid>
```

Abbiamo dichiarato il colore tramite la proprietà *Fill* per una questione di tipo di dato: il valore letterale in realtà rappresenta un elemento di tipo *Brush* e non *Color*, quindi non modificabile tramite *ColorAnimation*. La soluzione al problema consiste nel definire *Fill* utilizzando un oggetto *SolidColorBrush* e applicandogli un nome.

In questo modo possiamo risalire alla proprietà Color di quest'ultimo (Storyboard.TargetName="MyBrush" e Storyboard.TargetProperty="Color") e applicargli una *ColorAnimation*. Quindi, anche con i pochi oggetti Animation disponibili, possiamo animare qualsiasi elemento.

Infine implementiamo il listener e facciamo partire l'esempio:

```
private void MyRectangle_DoubleTap(object sender, System.Windows.Input.GestureEventArgs e)
{
    MyStoryboard.Begin();
}
```

Abbiamo introdotto le animazioni utilizzando Xaml e utilizzando tra le oltre cose l'oggetto **StoryBoard** per avviare l'animazione. In questo paragrafo

andremo a vedere in modo più approfondito quest'ultimo oggetto, ma prima introduciamo gli **EventTrigger**.

Esso è un elemento che incapsula una porzione di codice XAML che viene attivata solo al verificarsi di un determinato **RoutedEvent**. Un *RoutedEvent* invece rappresenta e identifica un evento indirizzato e dichiara le relative caratteristiche. Comunque, prendendo come riferimento un rettangolo, ecco come si implementa un *EventTrigger*:

```
<Rectangle Name="MyRectangle" Width="100" Height="100" Fill="Blue">
    <Rectangle.Triggers>
        <EventTrigger RoutedEvent="Rectangle.Loaded">

        </EventTrigger>
    </Rectangle.Triggers>
</Rectangle>
```

Il codice che andremo a inserire all'interno sarà chiamato solo quando si verificherà l'evento *Loaded* di *Rectangle*, cioè quando il rettangolo sarà caricato all'interno dell'interfaccia. Per completare l'esempio inseriamo uno *StoryBoard*, all'interno *dell'EventTrigger*, che opacizza il rettangolo fino a renderlo trasparente e viceversa:

```
<BeginStoryboard>
    <Storyboard x:Name="myStoryboard">
        <DoubleAnimation
    Storyboard.TargetName="MyRectangle"
    Storyboard.TargetProperty="Opacity"
    From="1.0" To="0.0" Duration="0:0:5"
    AutoReverse="True" RepeatBehavior="Forever" />
    </Storyboard>
</BeginStoryboard>
```

Tramite *RepeatBehavior* possiamo settare il ciclo di ripetizione, che in questo caso è settato su infinito (Forever).

Infine il codice C# sarà il seguente:

```
public MainPage()
{
    InitializeComponent();
    this.MyRectangle.Loaded += new RoutedEventHandler(myRectangleLoaded);
}

private void myRectangleLoaded(object sender, RoutedEventArgs e)
{
   this.myStoryboard.Begin();
}
```

E se volessimo eseguire un azione alla fine del processo di animazione?

Esiste l'evento **`Completed Completed="DoubleAnimation_Completed"`**

Continuando con lo *Storyboard*, possiamo immaginarlo come un player video, quindi eseguire su di esso le tipiche operazione di avvio (Begin), pausa (Pause), ripresa (Resume), interruzione (Stop) e tante altre.

Per dimostrare alcune di queste funzionalità partiamo dall'animazione precedente e impostiamo le proprietà Duration con valori più alti e aggiungiamo sotto al rettangolo una serie di Button per eseguire, da codice C#, le attività descritte prima. Quindi per fare pausa si inserisce nel listener del bottone relativo *myStoryboard.Pause()*, per riprendere invece si utilizza *myStoryboard.Resume()* e per stoppare *myStoryboard.Stop()*.

Con gli oggetti visti sin qui si possono realizzare solo animazioni lineari, che modificano il valore di una proprietà, passando da un valore iniziale a uno finale in un dato periodo di tempo, il tutto in maniera sequenziale.

In scenari complessi questi oggetti non potrebbero soddisfare le nostre esigenze ed è per questo che esiste un modello a oggetti alternativo che permette di esprimere con maggior dettaglio gli stati degli elementi che partecipano all'animazione, descrivendo i fotogrammi che la compongono. In questo tipo di animazioni indichiamo per ogni istante (frame) il valore esatto da assegnare a una proprietà. Purtroppo questo tipo di animazioni sono molto difficili, verbosi e complessi; realizzare grandi effetti applicati contemporaneamente a diversi elementi dell'interfaccia scrivendo tutto il codice dichiarativo a mano è impensabile. In questo ci viene in aiuto Expression Blend col quale possiamo registrare l'animazione in modo visuale spostando gli oggetti sulla scena e lavorando con la timeline. Una volta impostato tutto, sarà il programma a generare tutto lo XAML necessario a riprodurre l'animazione.

Capitolo 9

I Thread in C#

Un **Thread** permette l'esecuzione di un lavoro per conto del processo che lo ha generato ed esso viene creato con chiamata esplicita.

All'interno dello stesso processo tutti i thread sono eseguiti in modo concorrente secondo le regole definite dal programmatore. Ogni thread è caratterizzato da:

- una propria sezione di codice;
- una struttura dati riservata, informazioni sulle eccezioni, prossima istruzione da eseguire;
- una zona di memoria comune al processo dal quale viene generato.

L'utilizzo dei Thread offre diversi vantaggi a livello di esecuzione. In particolare si sfruttano appieno le prestazioni delle CPU multi-core, le quali costituiscono attualmente la tecnologia dominante nel mercato dei personal computer. Non è pensabile che applicazioni moderne di una certa complessità siano concepite in maniera sequenziale, in tal modo potrebbero sfruttare alla volta, uno solo dei core che la macchina mette a disposizione.

Nelle applicazioni si può fare uso di Thread che girano in Background, cioè in parallelo con il Thread dell'interfaccia grafica cosicchè essa non venga bloccata dall'operazione. Per default, i threads sono foreground, cioè tengono l'applicazione in vita fintanto che uno di loro è in esecuzione.

C# dà il supporto per i thread background, i quali non tengono in piedi l'applicazione per conto loro — terminando immediatamente una volta che tutti i thread foreground sono terminati.

Ecco un semplice esempio: **`BackgroundWorker bw = new BackgroundWorker();`**

```
bw.RunWorkerCompleted += new
RunWorkerCompletedEventHandler(DoneWork); bw.DoWork += new
DoWorkEventHandler(DoWork);

bw.RunWorkerAsync(arg1);
```

Sicuramente ricorderete l'uso di questo codice quando abbiamo parlato delle

ProgressBar e SplashScreen.

In questo fascicolo abbiamo utilizzato il Dispatcher che fornisce i servizi per la gestione della coda di elementi di lavoro per un thread. Su quest'ultimo abbiamo invocato il metodo BeginInvoke che esegue in modo asincrono il delegato specificato con la priorità specificata sul thread al quale Dispatcher è associato (nel nostro caso il Thread della UI).

Creare e eseguire Thread

I Thread sono creati usando il costruttore della classe **Thread**, passandogli un delegato **ThreadStart** — indicando il metodo in cui l'esecuzione comincia. Ecco come il delegato **ThreadStart** viene definito: **`public delegate void ThreadStart();`**

Il Thread parte chiamando il metodo **Start**. E continua fino a quando il suo metodo ritorna, cioè è a quel punto che il thread muore. Ecco un esempio, usando la sintassi estesa di C# per creare un delegato **ThreadStart**: **`class ThreadTest {`**

```
    static void Main() {
      Thread t = new Thread (new ThreadStart (Go));
      t.Start();.
}
static void Go() { Debug.WriteLine ("hello!"); }
```

In questo esempio, il thread **t** esegue **Go()**.Il risultato sarà un hello in output Un thread può essere creato più convenientemente tramite la sintassi compatta per istanziare i delegati: **`static void Main() {`**

```
    Thread t = new Thread (Go);  // Non è necessario usare ThreadStart
    t.Start();
    ...
```

```
}
```

In questo caso, un delegato ThreadStart è determinato automaticamente dal compilatore, tramite *inferenza*. Un'altra scorciatoia è usare un metodo anonimo per inizializzare un thread:

```
static void Main() {
    Thread t = new Thread (delegate() { Console.WriteLine
("Hello!"); });
    t.Start();
}
```

Un thread ha una proprietà **IsAlive** che ritorna il booleano vero dopo che il metodo Start() è stato chiamato, fino a quando il thread non finisce.

Classi Timer e DispatcherTimer

La classe Timer

Essa implementa un timer che genera un evento a intervalli definiti dall'utente. Questa classe è utile se si vogliono effettuare operazioni di controllo periodici. Si pensi ad esempio di voler creare un conto alla rovescia, alla fine del quale si deve verificare un evento. La cosa più spontanea è quella di usare la classe **Timer**. Vediamone quindi un esempio:

```
System.Threading.Timer timer;
int tempo = 5;
// Costruttore
public MainPage()
{
    InitializeComponent();
    timer  = new Timer( Callback, null, 1000, Timeout.Infinite );
    timer.Change(1000, Timeout.Infinite);
}

private void Callback(Object state)
{
    if (tempo > 0)
    {
        Debug.WriteLine("" + tempo);
        tempo--;
    }
    else
    {
        Debug.WriteLine("finito");
        timer.Dispose();
    }
}
```

Il costruttore riceve come parametri un metodo di "*CallBack*" che viene chiamato a ogni fine intervallo. 1000 è un secondo in millisecondi e indica dopo quanto tempo deve partire il Timer. Infine *Timeout.Infinite* indica la vita del timer che è infinita. Il costruttore può ricevere direttamente solo il metodo di Callback. Poi viene chiamato il metodo *Change* che consente di modificare e/o settare l'intervallo di tempo. Dato che vogliamo simulare un conto alla rovescia lasciamo 1000 millisecondi ed infine il metodo *Callback* controlla che il tempo sia maggiore di 0. Se lo è stampa in output la variabile tempo, altrimenti stampa finito e infine rilascia il Timer. Possiamo notare che in questo metodo non si è utilizzato nessuna TextBlock o MessageBox. Questo perché se provassimo a farlo otterremo il seguente errore: "*System.UnauthorizedAccessException non è stata gestita: Invalid cross-thread access*" che abbiamo già visto in precedenza. Perciò non si può aggiornare l'interfaccia utente utilizzando tale classe. Ecco quindi che ci viene in aiuto il **DispatcherTimer**.

Classe DispatcherTimer

Essa implementa un timer che genera un evento a intervalli definiti dall'utente. La differenza con la classe Timer sta nel fatto che è possibile utilizzarlo mentre si sta accedendo all'interfaccia utente. Quindi il conto alla rovescia sarà:

```
DispatcherTimer dTimer;
int tempo = 5;
// Costruttore
public MainPage()
{
    InitializeComponent();
    dTimer = new DispatcherTimer();
    dTimer.Interval = new TimeSpan(0, 0, 1);
    dTimer.Tick += new EventHandler(Callback);
    dTimer.Start();
}

private void Callback(object sender, EventArgs e)
{
    if (tempo > 0)
    {
        textBlock1.Text = ""+tempo;
        tempo--;
    }
    else
    {
        textBlock1.Text = "Finito";
        dTimer.Stop();
    }
}
```

Il suo utilizzo è semplice. Lo si inizializza con un costruttore vuoto. Tramite l'attributo *Interval* settiamo l'intervallo del timer (in questo caso un secondo) e ci sottoscriviamo all'evento *Tick*. In pratica finchè non ci sarà uno *Stop* del DispatcherTimer esso continuerà a chiamare a ogni intervallo di tempo il metodo Callback. Infine lo si fa partire con il metodo *Start*.

Appendice

La classe Math

Quando c’è bisogno di fare operazioni aritmetiche un po’ più complesse delle semplici quattro operazioni, oppure si vuole accedere a costanti come il pi-greco, in C# (come in Java) esiste la classe Math, utile in tutti i casi in cui si voglia lavorare con i numeri.

La classe

La classe **Math** mette quindi a disposizione metodi statici per funzioni trigonometriche (per esempio Cos(), Sin(), Tan(),), logaritmiche (log(), log10()) e le altre fuznioni matematiche più comuni (Abs(), Exp(), Min(), Max(), Pow(), Round() e così via).

Per vedere come si utilizza, creiamo una semplice classe che calcola l’assoluto di un numero: **`using System;`**

```
class Program
{
    static int Main()
    {
      int number = -6844;

      Console.WriteLine("Valore Originale = {0}", number);
      Console.WriteLine("Valore Assoluto = {0}\n",
Math.Abs(number));
      return 0;
```

```
    }
}
```

L’output sarà: `Valore Originale = -6844`
`Valore Assoluto = 6844`

I metodi

Andiamo a scoprire i metodi più importanti della classe Math:

- **Abs**: restituisce il valore assoluto di un numero.
- **Exp**: restituisce il valore elevato alla potenza specificata.
- **Log** e **Log10**: restituiscono il logaritmo naturale o in base dieci del numero.
- **Max** e **Min**: restituiscono il valore più grande o piccolo tra due numeri.
- **Pow**: restituisce la potenza del numero specificato per l’esponente.
- **Sign**: restituisce un valore che rappresenta il segno del numero. -1 se negativo, 1 se positivo, 0 altrimenti.
- **Pi**: restituisce la costante pi-greco.
- **E**: restituisce la costante *e*.
- **Sqrt**: restituisce la radice del numero.
- **Round**: arrotonda un valore a virgola mobile e precisione doppia a un numero di cifre frazionarie specificato.
- **Ceiling**: restituisce il valore intero minimo maggiore o uguale. Ad esempio Math.Ceiling(4.8); ritorna 5.
- **Floor:** restituisce l’intero massimo minore del o uguale.

In particolare vediamo un esempio per il metodo Round:

```
Console.WriteLine(Math.Round(5.44, 1)) ' ritorna 5.4

Console.WriteLine(Math.Round(5.45, 1)) ' ritorna 5.4

Console.WriteLine(Math.Round(5.46, 1)) ' ritorna 5.5

Console.WriteLine(Math.Round(5.54, 1)) ' ritorna 5.5
```

```
Console.WriteLine(Math.Round(5.55, 1)) ' ritorna 5.6
Console.WriteLine(Math.Round(5.56, 1)) ' ritorna 5.6
```

Bibliografia essenziale

Daniele Bochicchio, Cristian Civera, Marco De Sanctis, Riccardo Golia, Marco Leoncini, Alessio Leoncini, Stefano Mostarda, *C#4 espresso: Impara a sviluppare in C# — Scopri le novità di LINQ e Entity Framework — Con accesso ai dati, LINQ, ASP.NET*, Hoepli, 2012

Daniele Bochicchio, Cristian Civera, Marco De Sanctis, Riccardo Golia, Stefano Mostarda, Alessio Leoncini, Marco Leoncini, *C#5: Guida completa per lo sviluppatore*, Hoepli, 2013

Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, Peter Golde, *The C# Programming Language (Covering C# 4.0) (4th Edition)*, Addison-Wesley Professional, 2010

Antonio Agliata, Alessandro Forte, Paolo Gambardella, *Programmare in C# partendo da zero*, Edizionifutura.Com, 2012

Enrico Amedeo, *C# 4.0*, Apogeo, 2010

John Sharp, *Microsoft Visual C# 2012*, Mondadori Informatica, 2013

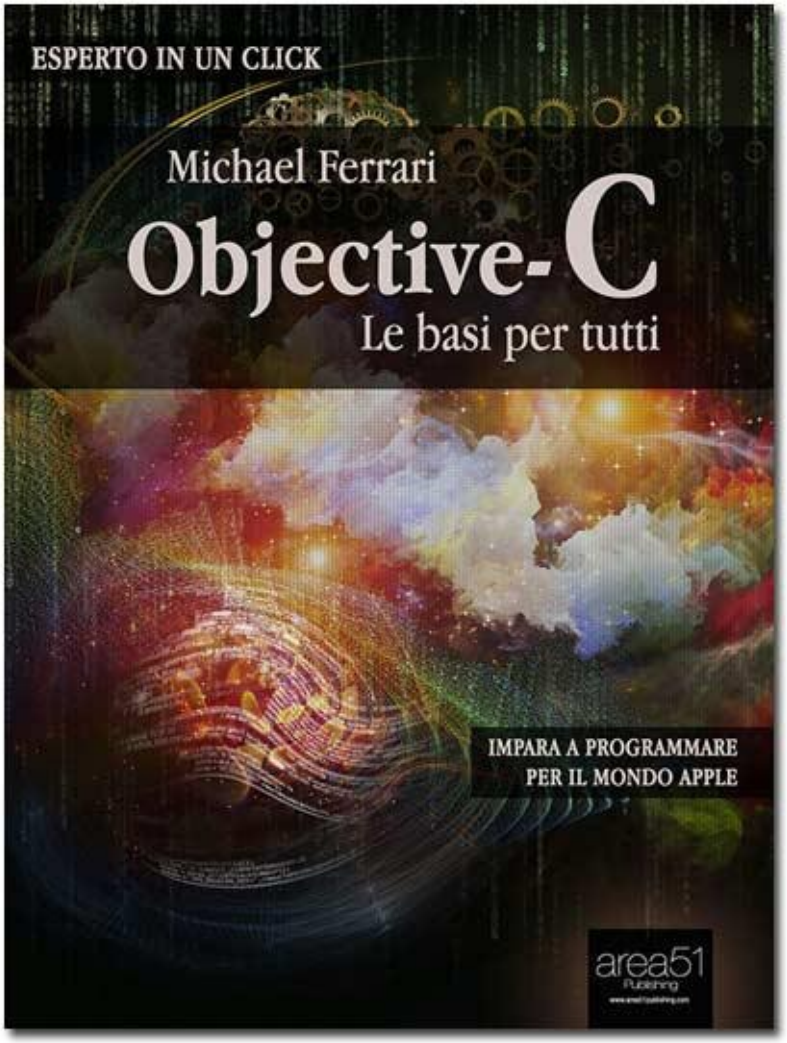
Christian Nagel, *C# e NET 4*, Hoepli, 2010

Nota sull’autore

Michael Ferrari nasce in Francia nel 1980 da madre americana e padre italiano. Giovanissimo si trasferisce in Italia e dopo aver frequentato il liceo scientifico si iscrive alla facoltà di Informatica, conseguendo la laurea nel 2005. Attualmente è un libero professionista e si occupa di consulenze informatiche per importanti aziende finanziarie. Nel tempo libero ama programmare e sviluppare software per i principali sistemi operativi. Tra i suoi hobby ci sono la robotica e la costruzione di droni.


area51
Publishing


ESPERTO IN UN CLICK
Mirco Baragiani
Programmazione C
Le basi per tutti
I FONDAMENTI PER CHI
VUOLE DIVENTARE PROGRAMMATORE
area51
Publishing


ESPERTO IN UN CLICK
Michael Ferrari
Objective-C
Le basi per tutti
IMPARA A PROGRAMMARE
PER IL MONDO APPLE
area51
Publishing


ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI:
+ iOS


ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI:
+ iOS


ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI:
+ iOS


ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI:
+ iOS


ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI:
+ iOS


ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
IMPARERAI:
+ iOS


ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
SVILUPPA APPLICAZIONI PER ANDROID E iOS
IMPARERAI:
+ iOS


ESPERTO IN UN CLICK
MIRCO BARAGIANI
CORONA SDK
SVILUPPA APPLICAZIONI PER ANDROID E iOS
IMPARERAI:
+ iOS


Esperto in un click
Corso di
BLENDER
1 Grafica 3D
Realizza un edificio in
3D


Esperto in un click
Corso di
BLENDER
2 Grafica 3D
Realizza oggetti avanzati con
BLENDER


Esperto in un click
Corso di
BLENDER
3 Grafica 3D


Esperto in un click
Corso di
BLENDER
4 Grafica 3D


Esperto in un click
Corso di
BLENDER
5 Grafica 3D
Lo sviluppo del
PAESAGGIO


Esperto in un click
Corso di
BLENDER
6 Grafica 3D
Dalla Blueprint alla mesh


Esperto in un click
Corso di
BLENDER
7 Grafica 3D
La dinamica dei corpi rigidi


Esperto in un click
Corso di
BLENDER
8 Grafica 3D
Materiali avanzati con Cycles


Esperto in un click
Corso di
BLENDER
9 Grafica 3D
Modellazione e render di
un elemento architettonico
complesso


ESPERTO IN UN CLICK
iOS 6
IMPARERAI:


ESPERTO IN UN CLICK
iOS 6
IMPARERAI:


ESPERTO IN UN CLICK
ALESSANDRO MONTARIELLO
iOS7
CORSO DI PROGRAMMAZIONE PRATICO
LIVELLO 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
da iOS 6 a iOS7
VOLUME SPECIALE
CONTIENE TUTTI GLI AGGIORNAMENTI
A IOS7 DEI VOLUME PRECEDENTI
INOLTRE IMPARERAI:


ESPERTO IN UN CLICK
LAMP
Guida per creare il tuo sito
LIVELLO 1
IMPARERAI:


ESPERTO IN UN CLICK
LAMP
Guida per creare il tuo sito
LIVELLO 2
DATABASE E FUNZIONI AVANZATE
IMPARERAI:


ESPERTO IN UN CLICK
LAMP
Guida per creare il tuo sito
LIVELLO 3
IMPARERAI:


ESPERTO IN UN CLICK
Alessandro Di Nicola
LINUX
CORSO COMPLETO
LIVELLO 1
PRIMI PASSI CON LINUX
IMPARERAI:
LIVELLO 2
LE APPLICAZIONI PER INTERNET E PER L'UFFICIO
IMPARERAI:
LIVELLO 3
LE APPLICAZIONI MULTIMEDIALI
IMPARERAI:
LIVELLO 4
LINUX SUI SERVER
IMPARERAI:
LIVELLO 5
LE APPLICAZIONI PER SERVER
IMPARERAI:

ALBERTO MISURACA

HTML

PRATICO PER WEB

area51
Publishing


ALBERTO MISURACA
HTML
PRATICO
PER WEB APP
orientamento e design
App Web - native
API PhoneGap
jQuery
jQuery
area51 Publishing

Indice