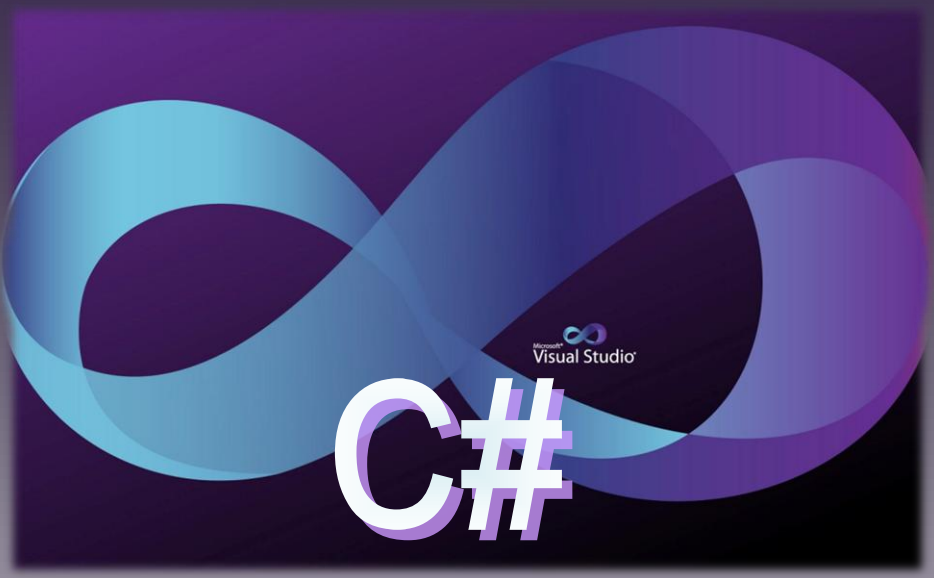
**ISTITUTO TECNICO INDUSTRIALE**
**G. M. ANGIOY**
**SASSARI**




# CORSO DI PROGRAMMAZIONE

## COLLEZIONI DI DATI

### DISPENSA 11.01

11-01_Liste_[15]




Versione del: 07/11/2015
Revisione numero: 15

Prof. Andrea Zoccheddu
Dipartimento di Informatica

**DIPARTIMENTO**
**INFORMATICA E TELECOMUNICAZIONI**

# LE COLLEZIONI DI DATI

## COLLEZIONI

### CONCETTO DI COLLEZIONE

#### VETTORI E COLLEZIONI

La sezione che tratta gli array accenna all'idea di collezione, che può essere definita come un contenitore capace di gestire più dati contemporaneamente, in contrapposizione all'idea di contenitore semplice, limitato a memorizzare un solo valore per volta.

Una **collezione** è definita come un contenitore che memorizza e gestisce un gruppo di oggetti, chiamati **elementi**.

Una collezione può memorizzare elementi anche **di tipo diverso**.

Esistono più tipi differenti di collezione ciascuna con **specifiche capacità** di memorizzazione e di gestione.

In questa trattazione però non affronteremo altre strutture dinamiche come gli alberi, le liste concatenate multiple, e le tabelle.

In questo corso tratteremo solo i tipi più semplici e comuni tra cui:

```
ArrayList         /* liste generiche */
List<Tipo>,       /* liste tipizzate */
Pile e Code       /* liste con politiche */
```

Il frame work di .NET offre vari tipi di collezioni. Le collezioni sono definite dentro specifici namespaces quali:

```
System.Collection
System.Collection.Specialized
System.Collection.Generics
```

Una o più di queste librerie deve essere specificata nel progetto che fa uso delle collezioni.

#### CARATTERISTICHE DELLE COLLEZIONI

Esistono diversi tipi di collezione, ciascuna caratterizzata da diverse regole di organizzazione, di accesso agli elementi, di gestione e memorizzazione. Il progettista deve scegliere quelle che sono più adatte ad essere impiegate per la gestione di uno specifico problema e che siano in grado di svolgere determinate azioni più facilmente e più correttamente.

Ogni collezione è caratterizzata dai seguenti aspetti:

- La natura dinamica oppure statica;
- Le operazioni consentite e la modalità di accesso agli elementi;
- La capacità di memorizzare elementi omogenei o eterogenei nel tipo;
- La struttura degli elementi;
- La capacità di mantenere gli elementi ordinati oppure l'assenza di questa possibilità;
- Il costo computazionale delle varie operazioni permesse sugli elementi;
- L'organizzazione interna degli elementi.

### *Collezioni dinamiche e statiche*

Una collezione si dice dinamica quando consente di ampliare o ridurre la quantità di elementi memorizzati nel tempo; se è possibile, allora consente di aggiungere o rimuovere elementi anche dopo che è stata creata. Una collezione non dinamica si dice statica e, pur ammettendo la modifica degli elementi memorizzati, non permette di modificarne il numero stabilito all'atto della creazione.

### *Inizializzazione delle collezioni*

Le recenti versioni di C# consentono di inizializzare qualunque collezione in fase di dichiarazione.

### *Operazioni consentite e modalità di accesso agli elementi*

Le collezioni permettono diversi modi di aggiungere elementi.

Alcuni tipi di collezione consentono di aggiungere elementi solo alla fine (in coda) e non altrove. Altre consentono di aggiungere e rimuovere elementi soltanto all'inizio (in testa) della collezione. Infine, altre collezioni consentono l'accesso e l'inserimento di elementi attraverso un indice che fa riferimento alla posizione dell'elemento nella collezione, mentre altre non lo permettono o comunque forniscono una modalità di accesso diversa.

### *Ordinamento degli elementi*

Alcuni tipi di collezione mantengono gli elementi sempre ordinati secondo un criterio. Ad una collezione di questo tipo ci si riferisce spesso con il termine **collezione ordinata**.

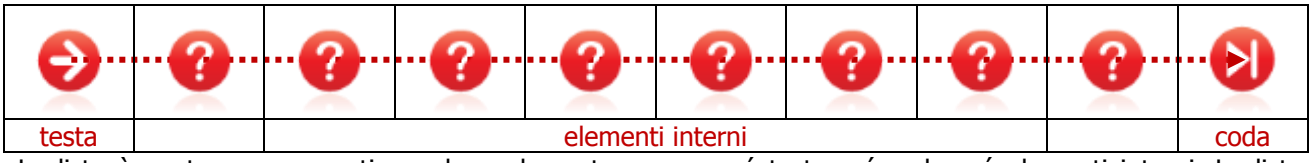
Solitamente con l'inserimento di un nuovo elemento in una lista, non c'è alcuna garanzia che l'ordine sia mantenuto. Alcune collezioni consentono di riordinare gli elementi in seguito, con un costo computazionale non indifferente.

## LISTE

### CONCETTO DI LISTA DI DATI

Per lista si intende una collezione di elementi che può essere ampliata e ridotta. La lista spesso è considerata una base per implementare altri tipi di collezione, più sofisticati.

Nella sua forma generale un lista può essere così schematizzata:




La lista è vuota se non contiene alcun elemento, ovvero né testa, né coda, né elementi interni. La lista spesso offre un modo per conoscere il numero degli elementi memorizzati, testa e coda inclusi.

Poiché la lista è dinamica consente di aumentare o diminuire il numero degli elementi, inserendoli o rimuovendoli, durante il suo ciclo di vita.

Come avviene per gli array, possiamo immaginare di disporre di un indice per ciascun elemento; in realtà questo non è così scontato e in molti casi le liste non offrono simili informazioni.

In C# le Liste sono quegli oggetti che permettono di collezionare insieme tanti elementi e di eseguire operazioni sia sulla collezione che sui singoli elementi. Puoi pensare ad una lista come un vettore, che però può cambiare dimensione a seconda delle esigenze e permette di aggiungere o togliere elementi. In C# ci sono tanti oggetti lista. Qui ne vedremo qualcuno.

### SPECIFICAZIONE DELLA LIBRERIA OPPORTUNA

Per usare una collezione occorre aggiungere una direttiva using di inclusione dello spazio System.Collections. Prima di usare perciò dovrai scrivere:

```
using System.Collections;
```

dopo le altre direttive **using** del tuo programma.

## ARRAYLIST (ELENCO)

### LA CLASSE ARRAYLIST

È una classe semplice con cui collezionare elementi di tipo eterogeneo. L'arrayList consente per esempio di mettere insieme numeri, stringhe, booleani e oggetti nella stessa collezione.

La collezione **ArrayList** è definita nel Namespace: «`System.Collections.Generic`»

La dichiarazione di un oggetto ArrayList è simile alla seguente:

```
ArrayList collezione;
```

ovviamente "collezione" è un nome qualsiasi; si sarebbe potuto usare pippo.

### COSTRUTTORI DI ARRAYLIST

Dopo aver dichiarato la lista è possibile invocare un costruttore per prepararla in memoria.

che predispone l'identificatore ma non lo istanzia. Se vuoi anche istanziarlo devi fare:

```
ArrayList collezione = new ArrayList();
```

### PROPRIETÀ DI ARRAYLIST

#### *Proprietà Count*

La proprietà Count restituisce il numero degli elementi presenti nella lista. Gli usi consueti sono per esempio assegnare a una variabile intera il numero degli elementi:

```
int quanti = collezione.Count;
```

Oppure usare il numero degli elementi per compiere un ciclo for:

```
for (int indice = 0; indice < collezione.Count; indice++)
     //istruzioni sulla collezione
```

Esempio:

```
ArrayList colori = new ArrayList();
colori.Clear();
colori.Add("rosso");
colori.Add("verde");
colori.Add("azzurro");
colori.Add("verde");
int quanti = colori.Count; //assegna 4 alla variabile quanti
MessageBox.Show(Convert.ToString(quanti));
```

## OPERAZIONI SULLA LISTA ARRAYLIST

### *Metodo Add*

**Prototipo: void Add(Tipo valore)**

Aggiunge un elemento alla lista. Esempi:

```
ArrayList collezione = new ArrayList();
collezione.Add(13); //aggiunge l'elemento alla lista
collezione.Add("ciao"); //aggiunge l'elemento alla lista
collezione.Add(True); //aggiunge l'elemento alla lista
```

Poiché la lista accetta oggetti generici è possibile archiviare in essa elementi di tipo diverso. Tuttavia di solito conviene usare una lista di tipi omogenei.

Per esempio:

```
for (int i = 0; i < 50; i++)
  collezione.Add(i);
```

per inserire 50 numeri nella lista.

### *Metodo Clear*

**Prototipo: void Clear()**

Rimuove tutti gli elementi dalla lista. Dopo l'esecuzione di questo metodo, la lista ha lunghezza zero. Per cancellare tutti gli elementi è possibile ad esempio eseguire l'istruzione:

```
collezione.Clear();
```

### *Metodo Contains*

**Prototipo: bool Contains (Object valore)**

Rende true se il valore esiste nella lista, false altrimenti.

```
if (collezione.Contains(13)) //se il valore 13 è contenuto nella collezione
     MessageBox.Show("è presente!"); //
```

### *Metodo IndexOf*

**Prototipo: int IndexOf (Object valore)**

Rende l'indice della prima occorrenza del valore specificato; se il valore non esiste ritorna -1.

```
int pos = collezione.IndexOf(13); //pone in pos l'indice del 13
if ( pos == -1 ) //se il valore 13 non è contenuto nella collezione
     MessageBox.Show("NON è presente!"); //
```

### *Metodo Insert*

**Prototipo: void Insert (int indice, Object valore)**

Inserisce il valore nella posizione indice.

Per inserire un elemento in mezzo alla lista puoi usare il metodo Insert, che richiede di specificare il dato da inserire e la posizione; per esempio:

```
numeri.Insert(25, 100); //inserisce il 100 in posizione 25
```

### *Metodo Remove*

**Prototipo: void Remove (Object valore)**

Elimina il valore dalla lista, se presente. Altrimenti non ha alcun effetto. Se ci sono più occorrenze di uno stesso valore, rimuove solo la prima che trova.

Per esempio:

```
ArrayList colori = new ArrayList();
colori.Clear();
colori.Add("rosso");
colori.Add("verde");
colori.Add("azzurro");
colori.Add("verde");
//qui hai quattro elementi in lista
colori.Remove("verde");
//qui ne restano tre
```

Il programma prepara una lista con quattro elementi; poi però ne rimuove uno. Alla fine nella lista restano tre elementi ("rosso", "azzurro" e "verde"), mentre il primo verde è stato rimosso.

### *Metodo RemoveAt*

**Prototipo: void RemoveAt (int indice)**

Elimina l'elemento che si trova nella posizione indice.

Per esempio:

```
ArrayList colori = new ArrayList();
colori.Clear();
colori.Add("rosso");   //elemento in posizione 0
colori.Add("verde");   //elemento in posizione 1 che dopo vogliamo rimuovere
colori.Add("azzurro"); //elemento in posizione 2
colori.Add("verde");   //elemento in posizione 3
//qui hai quattro elementi in lista
colori.RemoveAt(1);    //rimuove elemento in posizione 1 (verde)
//qui ne restano tre
```

Il programma prepara una lista con quattro elementi; poi però ne rimuove uno. Alla fine nella lista restano tre elementi ("rosso", "azzurro" e "verde"), mentre il primo verde è stato rimosso.

### ACCESSO DIRETTO AD ARRAYLIST

Per accedere ad un singolo elemento è possibile usare la notazione vettoriale con un indice tra parentesi quadre; ricorda che però è un trucco, reso possibile da un metodo nascosto e non è un accesso in memoria. Comunque puoi usare, ad esempio, le seguenti istruzioni:

```
int i = numeri[1]; //assegna ad i il secondo elemento della lista
numeri[0] = 17; //assegna 17 al primo elemento della lista
//non puoi accedere ad un elemento che non esiste
numeri[100] = 23; //errore: l'elemento 100 non esiste
int j = numeri[111]; //errore: l'elemento 111 non esiste
```

### ISTRUZIONE FOREACH PER ESPLORARE ARRAYLIST

Quando una lista ha degli elementi puoi esplorarli con la istruzione **foreach**. Se per esempio si dispone di una listBox1 in cui visualizzarli, allora puoi fare:

```
foreach (object num in numeri)
  listBox1.Items.Add (Convert.ToString(num));
```

ma osserva che se nella lista ci sono elementi diversi da int avrai un errore. Se usi foreach non puoi aggiungere o rimuovere elementi dalla lista, o avrai un errore.

## LIST <T>, LISTA TIPIZZATA

### LA CLASSE LIST<TIPO>

La collezione **List< T >** è definita nel Namespace: «`System.Collections.Generic`»

La classe List<> è una collezione generica che implementa la lista array. Data la sua grande flessibilità è il tipo di collezione impiegato più frequentemente. La lista tipizzata richiede, per il suo uso di esplicitare il tipo usato. Per esempio:

```
List < int > numeri;   //lista di numeri interi
List < string > nomi;  //lista di stringhe
List < double > temperature; //lista di numeri con la virgola
```

La lista tipizzata può essere definita per qualsiasi tipo, ma una volta scelto tutti gli elementi devono essere dello stesso tipo.

La lista tipizzata ammette l'accesso indicizzato agli elementi, come per i vettori.

### COSTRUTTORI DI LISTA TIPIZZATA

Dopo aver dichiarato la lista è possibile invocare un costruttore per prepararla in memoria.

```
List<int> numeri = new List<int>();
List<string> numeri = new List< string >();
```

Si osservi le parentesi tonde subito dopo il nome del costruttore invocato.

### PROPRIETÀ DI LISTA TIPIZZATA

#### *Proprietà Count*

La proprietà Count restituisce il numero degli elementi presenti nella lista. Gli usi consueti sono per esempio assegnare a una variabile intera il numero degli elementi:

```
int quanti = mialista.Count;
```

Oppure usare il numero degli elementi per compiere un ciclo for:

```
for (int indice = 0; indice < mialista.Count; indice++)
     //istruzioni sulla lista
```

## OPERAZIONI SULLA LISTA TIPIZZATA

### *Metodo Add*

**Prototipo:** **void Add(Tipo valore)**

Aggiunge un elemento alla lista.

Esempi:

```
Mialista.Add(13); //aggiunge l'elemento alla lista
```

### *Metodo Clear*

**Prototipo:** **void Clear()**

Rimuove tutti gli elementi dalla lista. Dopo l'esecuzione di questo metodo, la lista ha lunghezza zero.

### *Metodo Contains*

**Prototipo:** **bool Contains (Tipo valore)**

Rende true se il valore esiste nella lista, false altrimenti.

### *Metodo IndexOf*

**Prototipo:** **int IndexOf (Tipo valore)**

Rende l'indice della prima occorrenza del valore specificato; se il valore non esiste ritorna -1.

### *Metodo Insert*

**Prototipo:** **void Insert (int indice, Tipo valore)**

Inserisce il valore nella posizione indice.

### *Metodo Remove*

**Prototipo:** **void Remove (Tipo valore)**

Elimina il valore dalla lista, se presente. Altrimenti nulla.

### *Metodo RemoveAt*

**Prototipo:** **void RemoveAt (int indice)**

Elimina l'elemento che si trova nella posizione indice.

### *Metodo Sort*

**Prototipo:** **void Sort ()**

Ordina gli elementi della lista.

### *Metodo ToArray*

**Prototipo:** **Tipo[ ] ToArray ( )**

Restituisce un vettore costruito con copie degli elementi della lista

## USO DEL FOREACH SU LISTA TIPIZZATA

Per le liste tipizzate è possibile usare l'istruzione foreach nel seguente modo:

```
foreach (Tipo variabile in mialista)
     Istruzione;
```

esempio:

```
foreach (int k in numeri)
     if ( k = 0)
          Text += " § " + Convert.ToString(k);
```

A differenza dello ArrayListgli elementi di una lista tipizzata sono tutti del tipo specificato e quindi è possibile usare suddetto tipo anche nell'istruzione foreach.

# ESERCIZI

## USARE LE LISTE

### Esercizio 1) LISTA DI NOMI

Prepara un programma visuale con un pulsante che memorizza il nome scritto in una textBox dentro una lista di nomi; con un altro pulsante visualizza a video gli elementi nella lista; un terzo pulsante elimina dalla lista un nome digitato nella textBox; infine un quarto pulsante permette di eliminare un nome in base alla sua posizione individuato con un comboBox.

### Esercizio 2) LISTA DI NUMERI

Prepara un programma visuale; dichiara tre liste A, B e C. Il primo pulsante genera casualmente numeri positivi nella lista A, negativi nella B e ripulisce la lista C. Il secondo pulsante copia tutti gli elementi di A e B nella lista C; il secondo pulsante cambia di segno i numeri pari delle liste A e B; il terzo rimuove dalla lista C tutti gli elementi negativi.

### Esercizio 3) LISTA DI STRUTTURE

Di sicuro ti sei chiesto se sia possibile costruire una lista di strutture; prova a dichiarare una struttura Studente con i campi Nome, Età, Classe; prova poi a costruire un programma (possibilmente visuale) che prende in ingresso i dati di uno studente, prepara una struttura adatta e lo inserisce nella lista; prova anche a visualizzare l'elenco degli studenti esistenti.

### Esercizio 4) TEMPERATURE GIORNALIERE

Prepara un programma visuale che memorizza in una lista le temperature rilevate ogni ora per un intero giorno. Un pulsante pulisce la lista e simula la rilevazione e genera valori casuali per tutte le temperature. Un altro pulsante determina i valori massimo e minimo delle temperature rilevate.

### Esercizio 5) TEMPERATURE MENSILI

Prepara un programma visuale che memorizza in una lista le **temperature** di un intero mese, ma per ogni giorno sono rilevate a ogni ora. In effetti si può realizzare con un vettore di 30 liste, oppure con una lista di liste. Un pulsante pulisce tutte le liste e simula la rilevazione e genera valori casuali per tutte le temperature. Un altro pulsante determina i valori massimo e minimo giornalieri delle temperature rilevate.

### Esercizio 6) INTERROGAZIONI

Prepara un programma visuale per memorizzare le interrogazioni, ciascuna delle quali è composta da **data**, **materia** e **voto**; materia e voto sono scelte da rispettivi combo Box; un pulsante memorizza in una lista una interrogazione; un pulsante calcola la media dei voti in un materia scelta in una comboBox; un pulsante elimina dalla lista tutte le interrogazioni insufficienti; infine un pulsante elimina l'interrogazione zero.

### Esercizio 7) CAMPIONATO DI CALCIO

Prepara un programma per memorizzare i dati di partite di calcio; ogni partita è formata da due squadre (stringhe) e dai gol di ciascuna squadra (interi); un pulsante aggiunge una partita (dati prelevati da 4 comboBox); un pulsante calcola i gol segnati dalla squadra del comboBox1; un pulsante elimina tutte le partite pareggiate; un pulsante calcola e mostra in una listBox la classifica delle squadre;

Nota: nei primi due comboBox metti le stesse 6 squadre; quando aggiungi una partita verifica che una squadra non giochi contro sé stessa; nei due combo restanti proponi gol in numero da 0 a 10 (massimo);

# SOMMARIO