List e LinkedList

La classe **List<Tipo>** consente di creare un array dinamico di oggetti di un certo tipo, specificato in fase di dichiarazione dell'oggetto lista.

Si dice che l'array è "dinamico" nel senso che non ci si deve preoccupare di specificarne la dimensione in fase di dichiarazione, perché essa viene gestita automaticamente dal sistema mano a mano che si aggiungono nuovi elementi.

Ad esempio, creazione di una lista di numeri interi e di una lista di oggetti di tipo Persona:

```
List<int> listaDiInteri = new List<int>();

List<Persona> listaDiPersone = new List<Persona>();  // supponiamo di avere
                                                     // definito la classe Persona
```

Il metodo più usato per l'aggiunta di un elemento è Add(nuovoElemento), che aggiunge in fondo alla lista il nuovo elemento.

```
listaDiInteri.Add(20);
listaDiInteri.Add(40);
listaDiInteri.Add(60);
int n = lista.Count;   // n vale 3
```

Per leggere un singolo elemento si può considerare la sua posizione, la numerazione degli elementi parte da 0:

```
int valore = listaDiInteri[0];  //  valore vale 20
```

In modo del tutto analogo si può agire su una lista di oggetti di tipo Persona:

```
Persona p1 = new Persona(){Nome = "Antonio", Età = 30};
Persona p2 = new Persona(){Nome = "Gianni", Età = 40};
Persona p3 = new Persona(){Nome = "Luca", Età = 50};
listaDiPersone.Add(p1);  // aggiunge in fondo
listaDiPersone.Add(p2);  // aggiunge in fondo
listaDiPersone.Add(p3);  // aggiunge in fondo
```

Di solito si ha la necessità di utilizzare tutti gli elementi di una lista: pertanto conviene usare il ciclo foreach:

```
// scansione di tutta la lista per calcolare la somma delle età
int somma = 0;
foreach(Persona p in lista)
{
     somma = somma + p.Età;
}
Console.WriteLine(somma);  // scrive 120
```

Quando, invece, si vuole trovare un qualche specifico elemento nella lista non occorre scorrerla tutta, bensì è opportuno fermare il ciclo appena si arriva all'obiettivo.

E' fortemente sconsigliato interrompere brutalmente un ciclo con un'istruzione di return, perché questo rende poco leggibile il codice (… questo ricorda vecchi stili di programmazione!).

Pertanto si deve usare un ciclo **while**

1° metodo

```
// ricerca sequenziale nella lista
// si considera la lista come un semplice array statico e quindi si accede direttamente
// all'i-esimo elemento con la sintassi  lista[i]

int i = 0;
bool trovato = false;
posizione = -1;  // inizialmente non l'ho trovato
while (i<lista.Count && !trovato)
{
      Persona p = listaDiPersone[i];  // l'elemento corrente
      if (p.Nome == "nome cercato")
      {  trovato = true; posizione = i; }
      else
      { i++; }
}
Console.WriteLine(posizione);   // scrive il valore della posizione dell'elemento
                                // -1 se non è stato trovato
```

2° metodo

In alternativa si può **utilizzare un Enumeratore**, ovvero un oggetto associato alla lista che consente di scorrerla in avanti un elemento alla volta.
L'Enumeratore consente **soltanto movimenti in avanti** a partire dall'inizio della lista, con il metodo MoveNext. Questo metodo effettua, se possibile, l'avanzamento nella lista posizionando l'Enumeratore sull'elemento successivo a quello corrente. Esso restituisce *true* se tale operazione ha successo altrimenti restituisce *false*.
E' tuttavia possibile far ripartire l'Enumeratore dall'inizio con il metodo Reset. Con questo metodo si imposta l'enumeratore sulla propria posizione iniziale, ovvero **prima del primo elemento** nella lista.

```
//  ricerca sequenziale in una lista
bool trovato = false;
var e = listaDiPersone.GetEnumerator();  //  è di tipo   List<Persona>.Enumerator
while (e.MoveNext()  && !trovato)    //  MoveNext()  ritorna true se il prossimo
{                                    //  elemento esiste  e fa avanzare l'enumeratore
    Persona p = e.Current;
    if (p.Nome == "nome cercato")
    {  trovato = true; }
}
// la variabile trovato mi fornisce l'esito della ricerca
// in questo caso non ho a disposizione la posizione dell'elemento trovato
```

La classe List in sostanza memorizza gli elementi in celle di memoria contigue, proprio come avviene con gli array statici; quando le celle a disposizione non sono più sufficienti a contenere un ulteriore nuovo elemento il sistema automaticamente sposta tutte queste celle in una zona di memoria più grande.

Se si deve eliminare un elemento, si devono spostare anche tutte le celle che lo seguono per non lasciare "buchi" nella lista. Questo meccanismo fa rallentare le operazioni sulla lista.

Esiste pertanto la classe **LinkedList<Tipo>** (lista concatenata) che analogamente alla List ha una dimensione dinamica; la differenza è che essa non occupa celle contigue in memoria, ma piuttosto da celle liberamente collocate in memoria che sono collegate tra loro da "puntatori", in modo da poterle scorrere dall'inizio alla fine, senza problemi: si crea come una catena di celle di memoria.

Creazione di una lista concatenata di oggetti di tipo Persona:

```
LinkedList<Persona> listaConcatenata = new LinkedList<Persona>();
```

L'aggiunta di elementi può avvenire in fondo alla lista, oppure in testa alla lista:

```
listaConcatenata.AddLast(p1);   // aggiunto in coda
listaConcatenata.AddLast(p2);   // aggiunto in coda
listaConcatenata.AddFirst(p3);  // aggiunto in testa
```

Per la scansione della lista concatenata si può usare tranquillamente il ciclo foreach:

```
foreach(Persona p in listaConcatenata)
{
     Console.WriteLine(p.Nome);
}
// si ottiene
   Luca
   Antonio
   Gianni
```

Gli elementi di una LinkedList non possono essere presi direttamente specificandone la posizione:

listaConcatenata[i]   non si può fare!!!

Si può usare un Enumeratore per fare una scansione della lista, esattamente come con le List:

```
//  ricerca sequenziale in una lista
bool trovato = false;
var e = listaConcatenata.GetEnumerator();  //è di tipo LinkedList<Persona>.Enumerator
while (e.MoveNext()  && !trovato)   // MoveNext()  ritorna true se il prossimo
{                                   // elemento esiste  e fa avanzare l'enumeratore
     Persona p = e.Current;
     if (p.Nome == "nome cercato")
     {  trovato = true; }
}
```

```
// la variabile trovato mi fornisce l’esito della ricerca
```

Il metodo GetEnumerator() è previsto dall’interfaccia IEnumerable<T> che è implementata sia dalla classe List<T> che dalla classe LinkedList<T>

L’istruzione foreach è prevista per tutte le strutture dati che possiedono il metodo GetEnumerator().

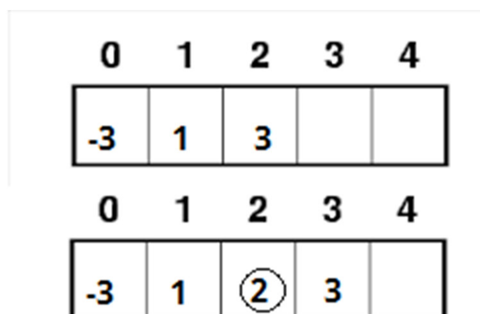
Infatti il ciclo

```
 foreach(Persona p in lista)
{
   // operazione su p
}
```

corrisponde a

```
var e = lista.GetEnumerator();
while(e.MoveNext())
{
   Persona p = e.Current;
   // operazione su p
}
```

Sia con le List che con le LinkedList è anche possibile inserire un nuovo elemento in mezzo alla lista, tuttavia nel caso della classe List viene eseguito lo spostamento di tutti gli elementi che si trovano a partire dalla posizione del nuovo inserimento fino alla fine della lista:



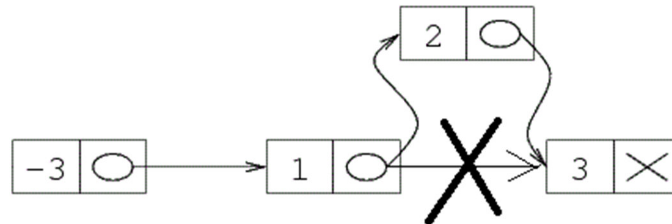
L’inserimento dell’elemento “2” comporta lo spostamento di tutti gli elementi che lo seguono

Le istruzioni per una lista di stringhe sono:

```
List<string> lista = …..

string nuovo = “2”;

// trovo la posizione per sapere dove inserire il nuovo valore

int posizione = lista.IndexOf(“1”);
```

```
lista.Insert(posizione+1, p);
```

Invece nel caso della LinkedList, l'inserimento a metà è velocissimo perché non si deve spostare nessun elemento, infatti è sufficiente modificare il puntatore dell'elemento precedente a quello nuovo per ricostruire tutta la catena di elementi:



Inserimento dell'elemento "2" nella lista

Le istruzioni sono

```
LinkedList<string> linked = ……
string nuovo = "2";   // elemento nuovo da inserire
// trovo il nodo precedente per sapere dove inserire il nuovo valore
var nodoPrecedente = linked.Find("1");  // è di tipo LinkedListNode<string>
linked.AddAfter(nodoPrecedente, nuovo);
```