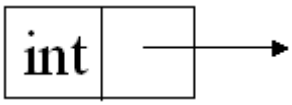
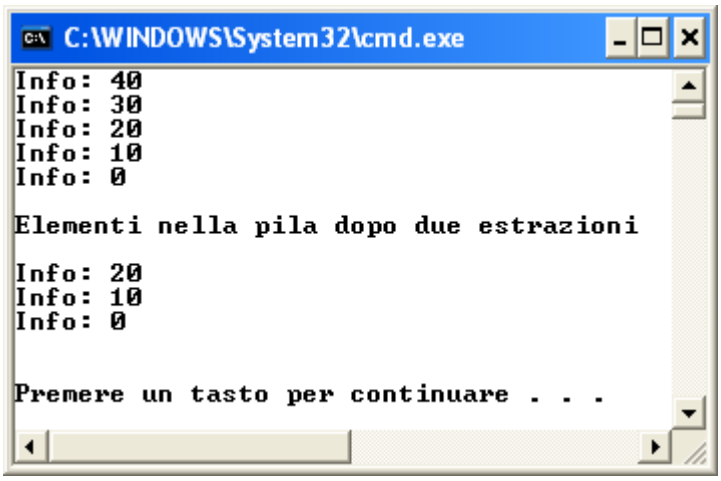
# Implementazione del ADT **Pila** senza array dinamici

```
import java.io.*;

class Nodo{             // classe self-referential

  public int Info;      // campo informativo
  public Nodo Next;     // campo puntatore

  Nodo(){
            Info =0;
            Next = null;
  }
  Nodo(int i){
            Info =i;
            Next = null;
  }
  Nodo(int i, Nodo ptr){
            Info =i;
            Next = ptr;
  }
}// fine classe Nodo

class Pila{                         // ... senza Fondo rispetto a Coda
  private Nodo Testa;

  public void Pila(){
            Testa=null;
  }

  // inserimento
  public void push (Nodo N){

            if(testVuota()) {
                      Testa =N;           // inserisci come primo nodo
                      N.Next =null;       // primo e unico
            }
            else {
                      N.Next =Testa;      // inserisci all'inizio
                      Testa=N;
            }
  }

  //estrazione
  public Nodo pop (){
            Nodo New = new Nodo();
            if (!testVuota())
                      {
                                New.Info=Testa.Info;
                                Testa=Testa.Next;
            }
            else
                      underFlow();

            return New;
  }

  //underFlow.... coda vuota
  private void underFlow(){
    System.out.println("\nPila vuota\n");          // da generare eccezione
  }
```

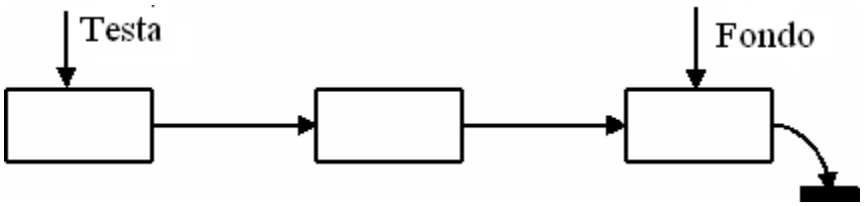



```
//testVuota
   public boolean testVuota(){
   return (Testa==null);
   }

   //stampa
   public void stampa(){

   Nodo attuale;
   attuale = Testa;
   while(attuale !=null){

                        System.out.println("Info: "+ attuale.Info);
                        attuale = attuale.Next;
               }
   }
}// fine classe Pila

public class TestPila{

public static void main (String args[]) {

   Nodo temp;
    int N =5;

   Pila pila1 = new Pila();
    for (int i =0; i< N;i++)   {

               temp = new Nodo(i*10);   // riempie la pila con multipli di 10
               pila1.push(temp);
   }
   pila1.stampa();
   System.out.println("\nElementi nella pila dopo due estrazioni\n");
   pila1.pop();
   pila1.pop();        // primi due nodi estratti
   pila1.stampa();

    System.out.println("\n");
}

}// fine classe principale
```



# Implementazione del ADT **Coda** senza array dinamici

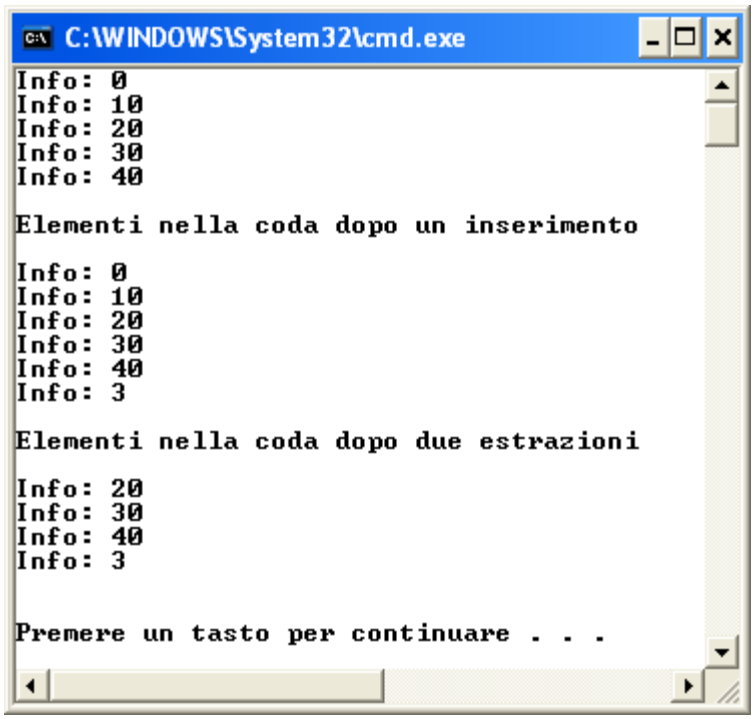
```
import java.io.*;

class Nodo{

   public int Info;          // campo informativo
   public Nodo Next;     // campo puntatore

   Nodo(){
            Info =0;
            Next = null;
   }

   Nodo(int i){
            Info =i;
            Next = null;
   }
   Nodo(int i, Nodo ptr){
            Info =i;
            Next = ptr;
   }
}// fine classe Nodo
```




```
class Coda{
   public Nodo Testa;
   public Nodo Fondo;

   public void Coda(){
            Testa=null;
            Fondo= null;
   }

   // inserimento
   public void inserisci (Nodo N){

            if(testVuota()){

                     Testa =N;          // inserisci come primo nodo
            }
            else{
                     Fondo.Next =N;      // inserisci alla fine
            }
            Fondo =N;
            N.Next=null;
   }
```

```
   //estrazione
   public Nodo estrai (){
            Nodo New = new Nodo();
            if (!testVuota()){
                              New.Info=Testa.Info;
                              Testa=Testa.Next;
            }
            else
                     underFlow();

            return New;
   }

//underFlow.... coda vuota
   private void underFlow(){
     System.out.println("\nCoda vuota\n");          // da generare eccezione
   }
```

```
//testVuota
   public boolean testVuota(){
   return (Testa==null);
   }

   //stampa
   public void stampa(){

   Nodo attuale;
   attuale = Testa;
   while(attuale !=null){

                        System.out.println("Info: "+ attuale.Info);
                        attuale = attuale.Next;
              }
   }
}// fine classe Coda

public class TestCoda{

public static void main (String args[]) {

   Nodo temp;
    int N =5;

   Coda coda1 = new Coda();
    for (int i =0; i< N;i++)   {

              temp = new Nodo(i*10);   // riempie la coda con multipli di 10
              coda1.inserisci(temp);
    }

   coda1.stampa();
   System.out.println("\nElementi nella coda dopo un inserimento\n");
   temp = new Nodo(3);              // aggiunge nodo con info 3
   coda1.inserisci(temp);
   coda1.stampa();
   System.out.println("\nElementi nella coda dopo due estrazioni\n");
   coda1.estrai();
   coda1.estrai();   // primi due nodi estratti
   coda1.stampa();

    System.out.println("\n");
}

}// fine classe principale
```

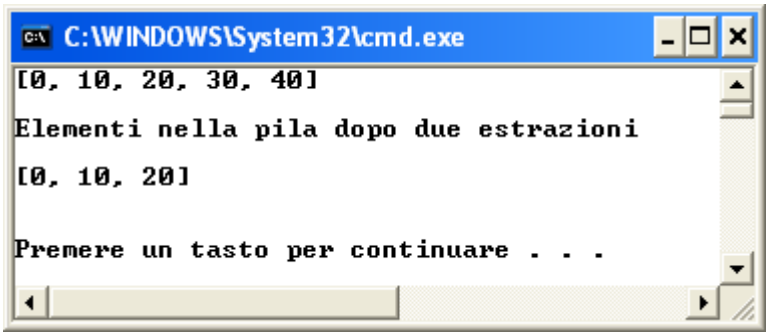



>> Implementazione di una **coda** con array dinamici

# Implementazione del ADT **Pila** con array dinamici

```
// Pila con array dinamici e Generics

/*
  ArrayList sono contenitori "estendibili" e "accorciabili" dinamicamente
  Prima di JDK5 potevano contenere solo oggetti Object
  In JDK5 sono parametrici (generici) rispetto al tipo degli oggetti contenuti
  (Prima ancora c'erano i Vector, analoghi ad ArrayList non generici)
*/

import java.io.*;
import java.util.*;

class Nodo{

  public int Info;          // campo informativo
  public Nodo Next;     // campo puntatore
  Nodo(){
        Info =0;
        Next = null;
  }
  Nodo(int i){
        Info =i;
        Next = null;
  }

  Nodo (int i, Nodo ptr){
        Info =i;
        Next = ptr;
  }
}// fine classe Nodo

class PilaV <T> {                              // ... senza Fondo rispetto Coda

   private ArrayList <T> elementi;

  PilaV(){
        elementi=new ArrayList <T>();
  }

  // push
  public void push(T o){

        elementi.add(o);        // aggiungo
  }

  //estrazione
  public T pop(){

         T o = null;

        if (elementi.size()>0)    {      // se non vuota

                o=elementi.get(elementi.size()-1);
                elementi.remove(elementi.size()-1);
        }else
                underFlow();

        return o;
  }
```

```
//underFlow.... coda vuota
   private void underFlow(){
            System.out.println("\nPila vuota\n");          // da generare eccezione
   }

   //testVuota
   public boolean testVuota(){
            if (elementi.size()==0)          // se vuota
                return true;
           else
                return false;
   }

   //stampa
   public void stampa(){

           System.out.println(elementi);
   }
}// fine classe PilaV

public class TestPilaAG{

public static void main (String args[]) {

   Integer temp;
    int N =5;

   PilaV <Integer> pila1 = new PilaV <Integer> ();                    // controllo sul tipo
    for (int i =0; i< N;i++)   {

            temp = new Integer (i*10);   // riempie la pila con multipli di 10
            pila1.push (temp);

    }
   pila1.stampa();
   System.out.println("\nElementi nella pila dopo due estrazioni\n");
   pila1.pop();
   pila1.pop();       // primi due nodi estratti
   pila1.stampa();

    System.out.println("\n");
}

}// fine classe principale
```



# Implementazioni del ADT Lista: collo di bottiglia nelle scansioni sequenziali della lista

/* Confronto tra implementazione di Liste:

- come **Array** (uso classe ArrayList)
- come **Lista di riferimenti** (uso classe LinkedList)

La classe ArrayList usa un array per contenere gli elementi, ed è particolarmente efficiente negli accessi con indici (sia diretti, sia sequenziali grazie a cache dei dati). Tempo d'accesso elemento i-esimo: **costante**; tempo d'inserimento o cancellazione: **lineare**. Impossibilità di ridimensionamento se la memoria è fragmentata

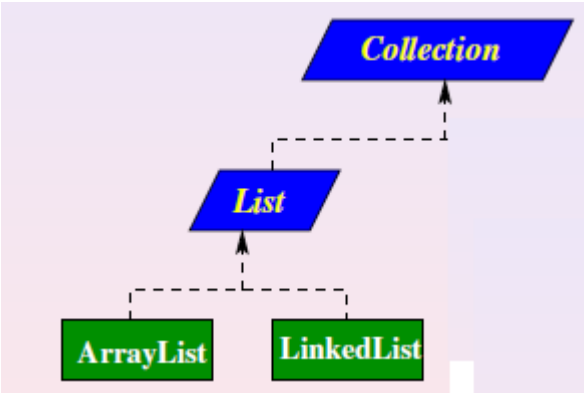
La classe LinkedList usa una lista collegata, e la sua specialità sono gli inserimenti e le rimozioni di sequenze di elementi a "*metà*" della lista ma è critico l'accesso con indice (solo sequenziale). In realtà, grazie ai metodi disponibili, permette collezioni che condividono la stessa "coda" (struttura dati persistente).
Tempo d'accesso indicizzato: **lineare** (cioè che dipende dal numero degli elementi); tempo d'inserimento o cancellazione: **costante**

Si utilizza la classe **Cronometro** (esterna) :

- il costruttore resetta il cronometro invocando il metodo d'istanza **azzera()** che ferma e azzera ma non avvia il conteggio
- per avviare il conteggio si possono usare i metodi: **avanza()** e **avanzaDaCapo()**
- per fermare: si può usare il metodo **ferma()**
- per leggere: si può usare il metodo **leggi()** che ritorna un tipo long
- per conversione in stringa del conteggio corrente: si può usare il metodo **toString()** ridefinito

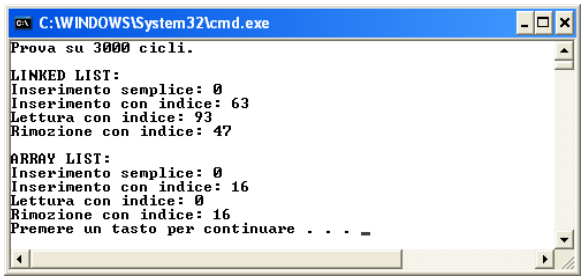
*/

```
import java.util.*;

public class ProvaListe {

  private static int cicli;

  public static void main ( String []args )  {
  if(args.length > 0)
        cicli = Integer.parseInt(args[0]);        // da linea comando eventuale
  else
        cicli = 3000;
  System.out.println( "Prova su " + cicli + " cicli." );
System.out.println( "\nLINKED LIST: " );
  test( new LinkedList <String>() );
  System.out.println( "\nARRAY LIST: " );
  test( new ArrayList <String>() );
 }
 private static void test (List<String> l )  {   // interfaccia List

  Cronometro c = new Cronometro();          // resetta
  int indice;
  String o;
  long sec;

  // inserimento semplice
  c.avanza();                               // partenza
  for ( int i = 0; i < cicli; i++ ) {
      l.add( new String () );
   }
  sec = c.leggi();
  System.out.println( "Inserimento semplice: " + sec );

  // inserimento con indice
  c.avanzaDaCapo();                         // ferma e riavvia
  for ( int i = 0; i < cicli; i++ )  {
    l.add( ( int )Math.rint( cicli / 2 ), new String() );
  }
  sec = c.leggi();
   System.out.println( "Inserimento con indice: " + sec );
```

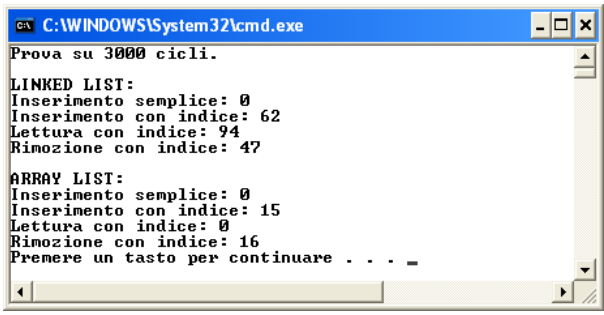


```
    // lettura con indice
   c.avanzaDaCapo();                          // ferma e riavvia

   for ( int i = 0; i < l.size(); i++ )
       o= l.get( i );

   sec = c.leggi();
   System.out.println( "Lettura con indice: " + sec );

   // rimozione con indice
   c.avanzaDaCapo();                          // ferma e riavvia

   for ( int i = 0; i < l.size(); i++ ) {
     indice = ( int )( Math.random() * l.size() );
     l.remove( indice );
   }
   sec = c.leggi();
   System.out.println( "Rimozione con indice: " + sec );

  c.ferma();                                  // arresto
 }
}
```

Valori diversi per lanci diversi (conseguenza dell'ambiente **multitasking** e dello stesso linguaggio Java, con il suo imprevedibile ***garbage collector***). Quello che interessa è una misurazione approssimativa:




**array**: accesso praticamente immediato
inserimenti/rimozioni lenti




Un array di oggetti: è una sequenza di celle di memoria, ciascuna delle quali può contenere un riferimento ad un oggetto. Le celle sono numerate. Accedere ad una cella è facilissimo: il sistema conosce la posizione in memoria della prima cella e le dimensioni di tutte le celle, quindi si limita a fare un semplice calcolo:

*Posizione cella di indice N = Posizione prima cella + N * Dimensione di una cella*

Dove *N* rappresenta l'indice dell'array, e parte da 0. Quindi, se voglio accedere al millesimo elemento di un array che contiene un milione di elementi basta "puntare il dito" con questo calcolo. L'operazione è praticamente immediata.

Un array ha anche alcuni ***svantaggi***. Per cominciare, è una **struttura statica**. Quando creiamo un array di cento elementi, il sistema alloca una fetta sequenziale di memoria sufficiente per cento riferimenti. Se vogliamo aggiungere un centounesimo elemento ci tocca creare un altro array più grande, copiare tutti gli elementi dal vecchio al nuovo array e poi distruggere il vecchio array. Dietro le quinte, la classe *ArrayList* ricorre spesso a questo sistema.

Il secondo svantaggio dell'array è che gli **elementi** sono tutti "**uno accanto all'altro**". Poniamo di avere un array di diecimila elementi ordinati, con i primi mille elementi "pieni" (che contengono dei riferimenti a oggetti) e gli altri "vuoti" (che contengono null). Se decidiamo di inserire un nuovo elemento nella cinquantesima posizione, dobbiamo prima fargli spazio spostando in avanti di una posizione tutti gli elementi dal cinquantesimo al millesimo. Se l'array è grande questa operazione può diventare molto lenta.

Una struttura dati alternativa agli array è la "**lista di collegamenti**", o "lista collegata". Tipicamente, ciascuna cella di una lista collegata contiene tre riferimenti: uno all'oggetto contenuto nella lista, uno alla cella precedente nella lista e uno alla cella successiva nella lista.

Le celle **non devono essere adiacenti** come quelle di un array, ma possono essere sparse per la memoria. Basta conservare un riferimento al primo elemento della lista, e si potrà sempre scorrerla passando da un elemento all'altro, avanti e indietro.

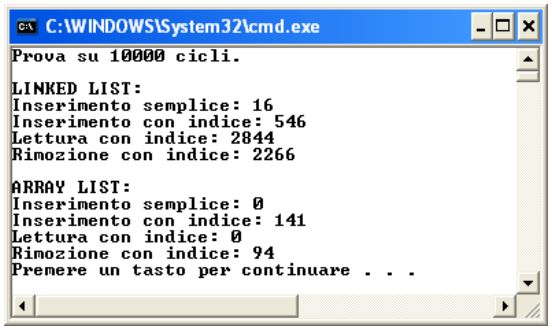
Una lista collegata è come una catena dove ogni anello è collegato con il precedente e il successivo. Si può arrivare all'ultimo anello partendo dal primo e saltando al secondo, poi al terzo, e così via fino alla fine della catena.

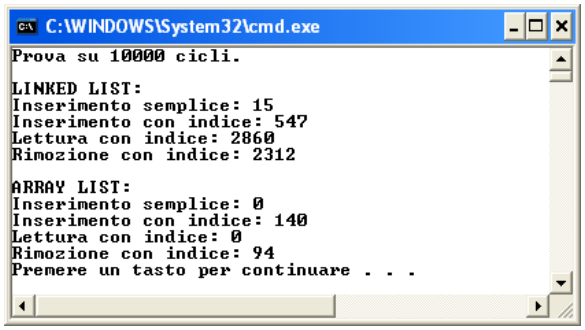
E' chiaro che questa catena può **allungarsi o accorciarsi** senza problemi. Quando si crea una lista, non ci si deve preoccupare di quanti elementi dovrà contenere. Se si vuole aggiungere un undicesimo elemento ad una lista che ne contiene dieci, basterà aggiungere un anello in coda alla catena.

E' anche molto facile aggiungere gruppi di elementi in una ***posizione centrale***: basta "spezzare la catena", inserire la nuova sequenza di anelli e agganciarla all'anello precedente e al successivo. In questo caso una lista collegata è ***più efficiente*** di un array.

Ma le liste collegate lasciano davvero a desiderare nel caso di accesso indicizzato. Se si deve accedere a tutti gli elementi tra uno e diecimila, una lista collegata funziona bene come un array.

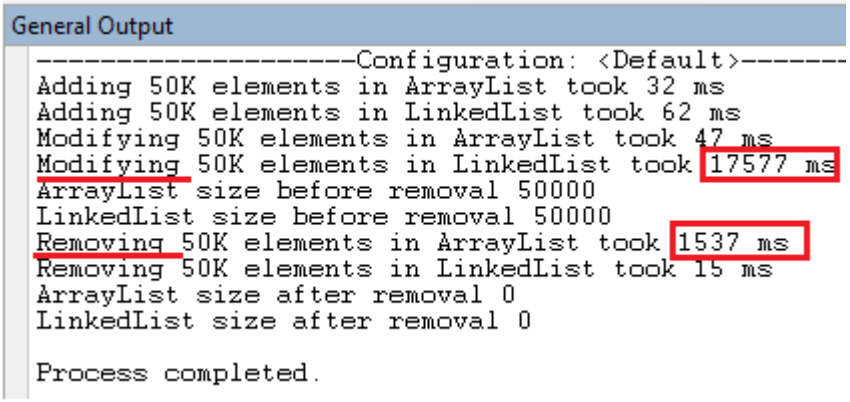
Si può partire dal primo elemento e chiedergli l'indirizzo del secondo, poi leggere il secondo e chiedergli l'indirizzo del terzo, e così via. Il problema nasce se dobbiamo accedere direttamente al centomillesimo elemento di una lista.



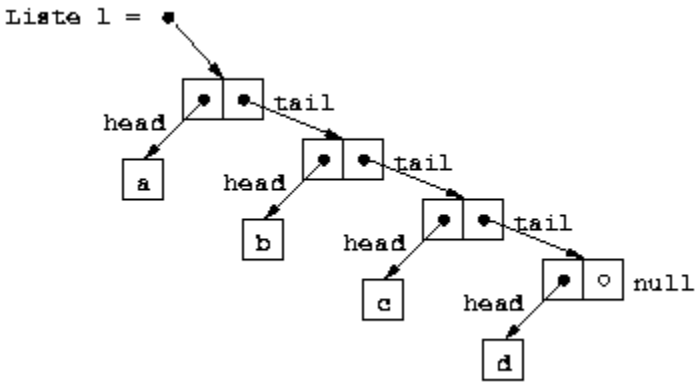

Mentre un array ci permette di "puntare il dito" sull'elemento giusto, nel caso della lista collegata abbiamo un solo modo per individuare l'indirizzo dell'elemento che ci serve: partire dal primo elemento e scorrere la lista in avanti, contando, finché non siamo arrivati a centomila.

>> altro esempio (codice senza classe esterna)

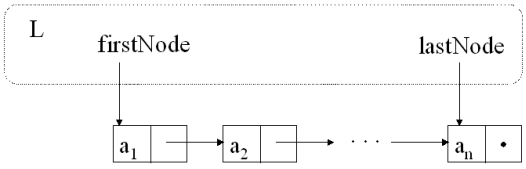


# Lista

In una struttura dati "**lista**", tipicamente, ciascuna cella contiene due riferimenti: uno all'oggetto contenuto nella lista (campo *informazione*) e uno alla cella successiva nella lista (campo *puntatore*).



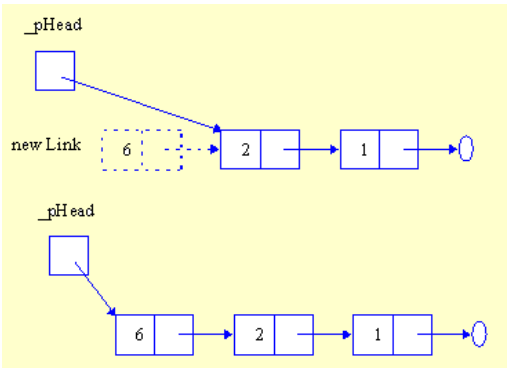

Esempio di classe:

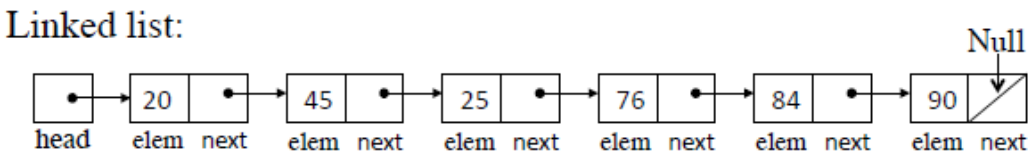



```
class List {
    private Node firstNode;
    private Node lastNode;

    // . . . . . .
}
```

In una struttura dati "**lista di collegamenti**" o "lista collegata", tipicamente, ciascuna cella contiene tre riferimenti: uno all'oggetto contenuto nella lista, uno alla cella precedente nella lista e uno alla cella successiva nella lista. Solitamente il campo informativo viene evidenziato come valore:

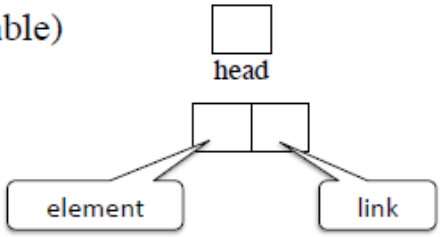
linked list
*unidirezionale*



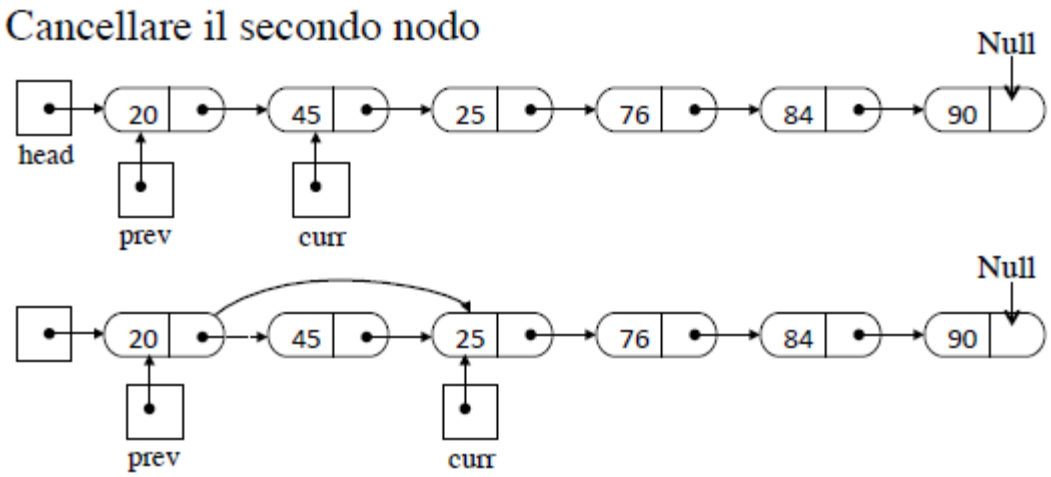




Strutture dati:

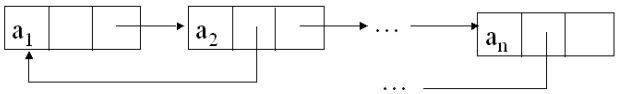
- una variable (reference variable) di tipo node
- node data type



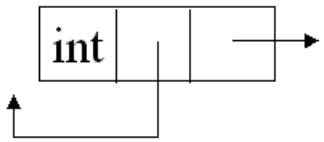
Esempio di ***rimozione***




*Doppia* linked list (*bidirezionalità*) :



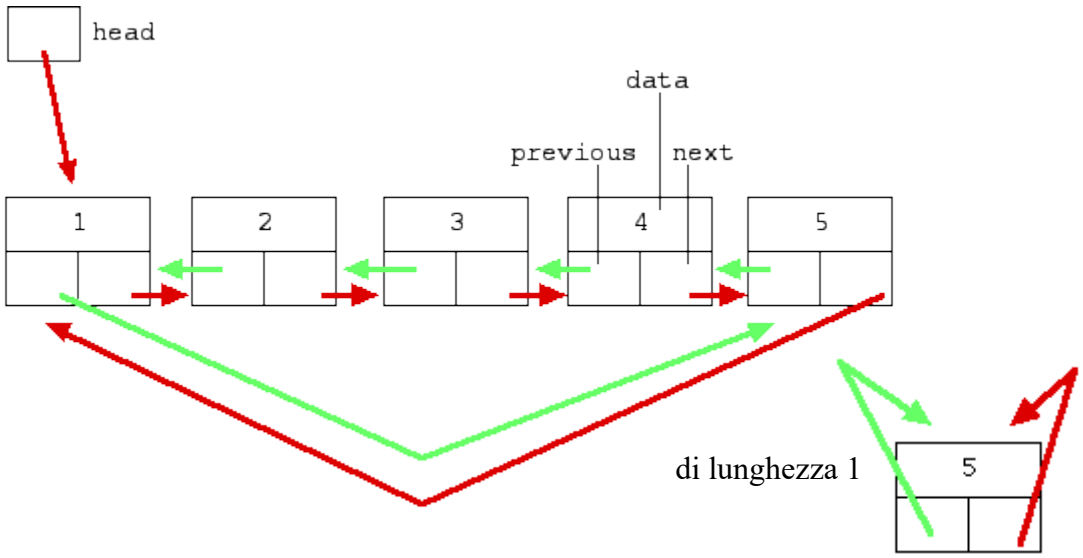

La bidirezionalità:




è implementata con

```
class Node {
    private int data;
    private Node next;
    private Node prev;
}
```

Doppia linked list *circolare*



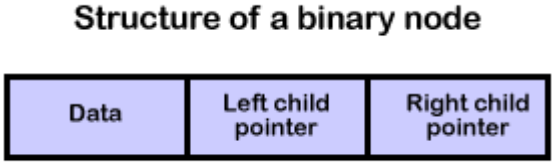

http://courses.cs.vt.edu/csonline/DataStructures/Lessons/OrderedListImplementationView/Lesson.html audio per modalità di ordinamento strutture dati tipo **lista**: inserimento e accodamento

**Stack** http://courses.cs.vt.edu/csonline/DataStructures/Lessons/StacksAbstractView/index.html

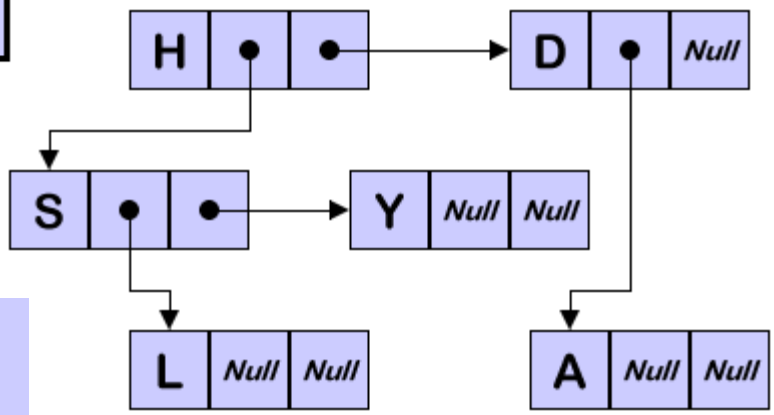
**Coda** http://courses.cs.vt.edu/csonline/DataStructures/Lessons/QueuesAbstractView/index.html
Dalla versione 1.5 nel JCF interfaccia Queue (esempi: classe VectorQueue javadoc e listing) e dalla versione 1.6 interfaccia Dequeue *(specializzazione per doppia coda)*

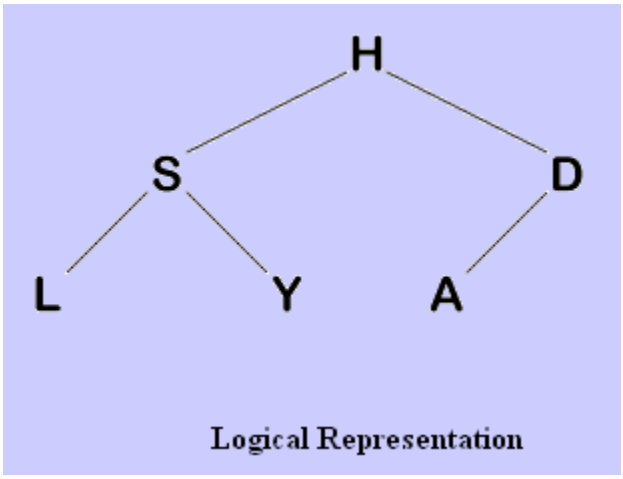
Strutture dati ***NON LINEARI***:

Array multidimensionali
http://courses.cs.vt.edu/csonline/DataStructures/Lessons/2DArrays/index.html audio



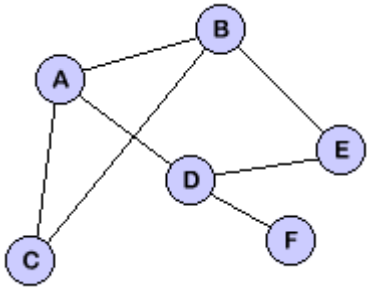

per **alberi binari**:




**Grafi**

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| **A** | -- | 1 | 1 | 1 | -- | -- |
| **B** | 1 | -- | 1 | -- | 1 | -- |
| **C** | 1 | 1 | -- | -- | -- | -- |
| **D** | 1 | -- | -- | -- | 1 | 1 |
| **E** | -- | 1 | -- | 1 | -- | -- |
| **F** | -- | -- | -- | 1 | -- | -- |




http://courses.cs.vt.edu/csonline/DataStructures/Lessons/Graphs/index.html

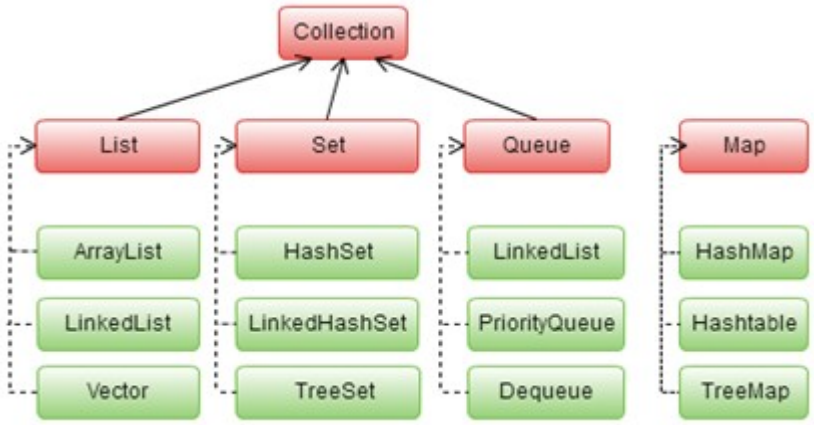
**ADT**: 1) una particolare struttura dati,
2) operazioni relative alla struttura dati

"un *data type* consiste nel **valore** che rappresenta e nelle **operazioni** consentite su di esso"
"*abstract*" perchè definiti senza nulla sapere dell'implementazione

| Data Structure | Operazioni tipiche | Abstract View |
|---|---|---|
| Ordered List | **Aggiungi elemento**<br>**Rimuovi elemento** |  |
| Stack | **Push**<br>**Pop** |  |
| Queue | **Accoda**<br>**Estrai** |  |

http://courses.cs.vt.edu/csonline/DataStructures/Lessons/AbstractDataTypes/index.html

**Java Collection Framework**




>> elaborazioni disponibili per tutte le **Collections** con **esempio**

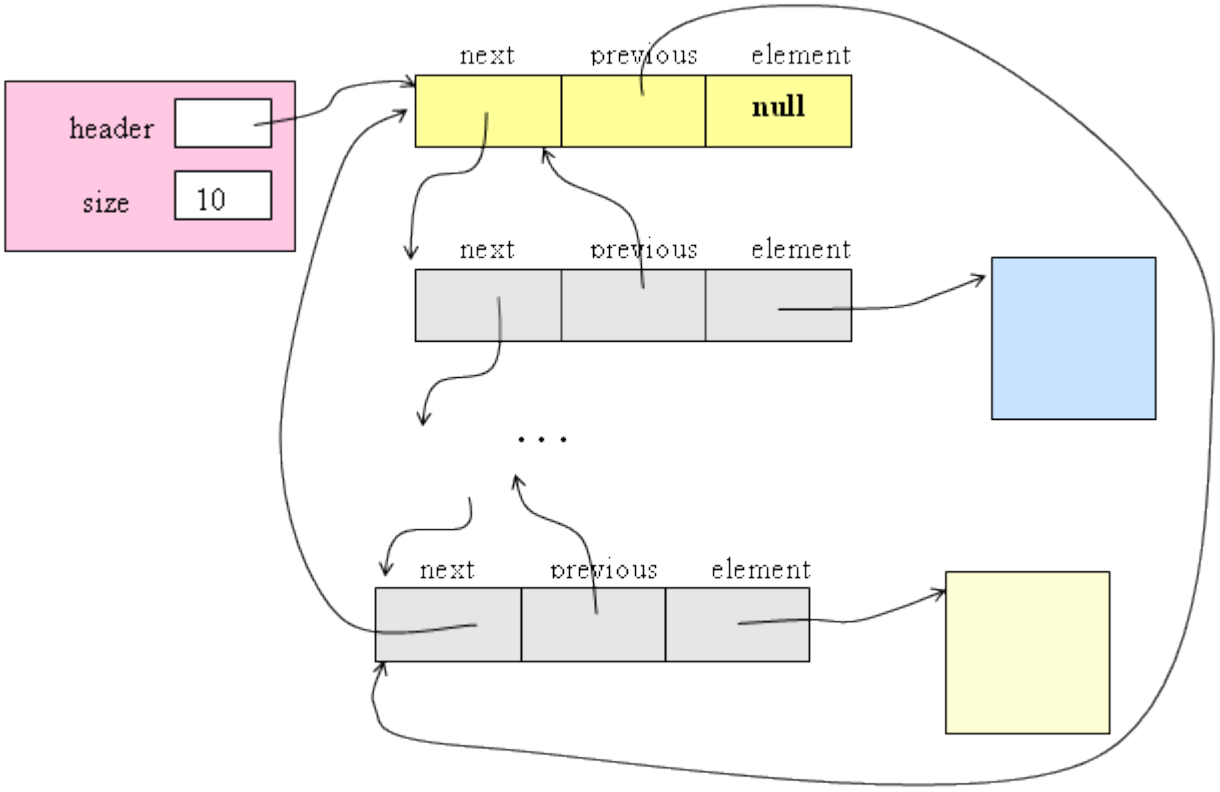
**>> Ordinamento:**

**IParte**

**II Parte**

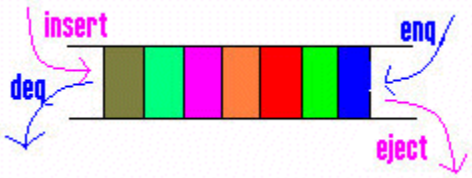
## LinkedList

La classe *LinkedList* implementa una **lista d'oggetti** rappresentabili nel modo seguente:




Il primo elemento della lista (in giallo nel disegno) facilita la scrittura degli algoritmi

La classe *LinkedList* implementa l'interfacccia *List* (ne è un'implementazione che permette l'elemento null) ed introduce metodi per leggere (*get*), rimuovere (*remove*) ed inserire (*insert*) un elemento anche all'inizio o alla fine in modo da poter implementare sia una pila o una coda sia una *double-ended queue* (deque).




Implementa anche l'*interfaccia Queue* (operazioni FIFO *in una delle nuove interfacce base che sono 9 nella versione 5*) consentendo l'implementazione di *doubly-linked list* (liste bidirezionali). Altre interfacce e relative classi permettono di realizzare ***code** a priorità* o *orientate a sistemi multithreading.*

**nb**: **ArrayList** *ha prestazioni nettamente superiori a* **LinkedList**, *che conviene utilizzare solo per gestire collezioni di tipo "coda". Infatti mette a disposizione metodi come:*

addFirst, getFirst, removeFirst,
addLast, getLast, *e* removeLast.

*È quindi opportuno scegliere l'utilizzo di una* `LinkedList` *in luogo di* `ArrayList`, *solo quando si devono* ***spesso aggiungere elementi all'inizio*** *della lista, oppure* ***cancellare elementi all'interno*** *della lista durante le iterazioni.*

*Queste operazioni hanno infatti un tempo costante nelle* `LinkedList`*, e un tempo "lineare" (ovvero che dipende dal numero degli elementi) in un* `ArrayList`*.*

*In compenso però, l'accesso posizionale in una* `LinkedList` *è lineare mentre è costante in un* `ArrayList`*. Questo implica una performance superiore dell'*`ArrayList` *nella maggior parte dei casi.*

*Anche l'*`ArrayList` *possiede un parametro di configurazione: la capacità iniziale. Se istanziamo un* `ArrayList` *con capacità iniziale* `20`*, avremo un oggetto che ha* `20` *posizioni vuote (ovvero in ogni posizione c'è un reference che punta a* `null`*), che possono essere riempite. Quando sarà aggiunto il ventunesimo elemento, l'ArrayList si ridimensionerà automaticamente per avere capacità ventuno, e questo avverrà però ogni nuovo elemento. Ovviamente, per ogni nuovo elemento che si vuole aggiungere oltre la capacità iniziale, l'arraylist dovrà prima ridimensionarsi e poi aggiungere l'elemento.*
*Questa doppia operazione porterà ad un decadimento delle prestazioni.*

*N.B.: È però possibile ottimizzare le prestazioni di una arraylist nel caso si vogliano aggiungere nuovi elementi superata la capacità iniziale. Infatti, quest'ultima si può modificare a nostro piacimento "al volo", in modo tale che l'arraylist non sia costretto a ridimensionarsi per ogni nuovo elemento. Per fare ciò, basta utilizzare il metodo* `ensureCapacity()` *passandogli la nuova capacità, prima di chiamare il metodo* `add()`*. Per avere un'idea di quanto sia importante ottimizzare viene presentato un semplice esempio. Sfruttiamo il metodo statico* `currentTimeMillis()` *della classe* `System` *per calcolare i milllisecondi che impiega un ciclo a riempire l'arraylist:*

```
ArrayList<String> list = new ArrayList<String>(1);
//capacità iniziale 1
long startTime = System.currentTimeMillis();
int N =100000;
list.ensureCapacity(N);
for (int i = 0; i < N; i++) {
     list.add("nuovo elemento");
}
long endTime = System.currentTimeMillis();
System.out.println("Tempo = " + (endTime - startTime));
```

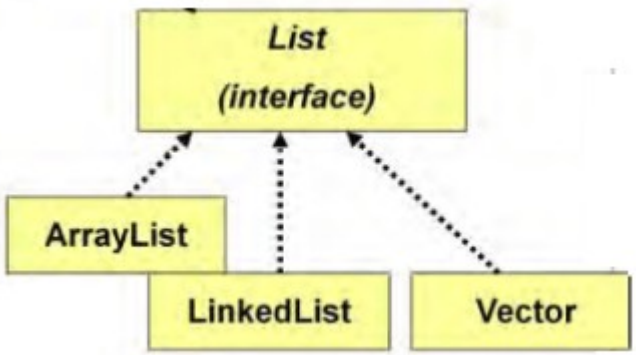
*L'output su PC con precessore Pentium 4 (2.8 Ghz) è circa*
`Tempo = 16 se N = 100000`

*La performance è nettamente peggiore, e la differenza sarà ancora più evidente aumentando il numero di elementi:* `Tempo = 78 se N = 1000000`

*N. B. : se rimuoviamo un elemento da un arraylist, la capacità dell'arraylist non diminuisce. Esiste il metodo* `trimToSize()`*, per ridurre la capacità dell'arraylist al numero degli elementi effettivi.*

## La classe Vector

*In generale la classe* **Vector** *offre prestazioni inferiori rispetto ad un* `ArrayList` *essendo sincronizzata. Per questo utilizza due parametri per configurare l'ottimizzazione delle prestazioni: la capacità iniziale e la capacità d'incremento.*




*Per quanto riguarda la capacità iniziale vale quanto detto per l'arraylist. La capacità d'incremento (specificabile tramite un costruttore), permette di stabilire di quanti posti si deve incrementare la capacità del vettore, ogni qual volta, si aggiunga un elemento che "sfora" il numero di posizioni disponibili. Se per esempio istanziamo un Vector nel seguente modo:*

```
Vector <String> v = new Vector<String> (10, 10);
```

*dove il primo parametro è la capacità iniziale e il secondo la capacità di incremento, quando aggiungeremo l'undicesimo elemento, la capacità del vettore sarà resettata a* `20`*. Quando aggiungeremo il ventunesimo elemento, la capacità del vettore sarà resettata a* `30` *e così via. Il seguente codice per esempio:*

```
Vector<String> list = new Vector <String> (10,10);
for (int i = 0; i < 11; i++) {
                    list.add("1");
}
System.out.println("Capacita' = " + list.capacity());
for (int i = 0; i < 11; i++) {
                    list.add("1");
}
System.out.println("Capacita' = " + list.capacity());
```

*produrrà il seguente output:*

```
Capacita'= 20
Capacita'= 30
```

*Se istanziamo un vettore senza specificare la capacità di incremento, oppure assegnandogli come valore un intero minore o uguale a zero, la capacità verrà raddoppiata ad ogni "sforamento". Se quindi per esempio istanziamo un* `Vector` *nel seguente modo:*

```
Vector<String> v = new Vector<String> ();
```

*dove non sono state specificate capacità iniziale (che di default viene settata a* `10`*) e capacità di incremento (che di default viene settata a* `0`*), quando aggiungeremo l'undicesimo elemento, la capacità del vettore sarà resettata a* `20`*. Quando aggiungeremo il ventunesimo elemento, la capacità del vettore sarà resettata a* `40` *e così via.*

*N.B. : Un* `ArrayList` *si può sincronizzare sfruttando uno dei tanti metodi statici di utilità della classe* `Collections` *(da non confondere con* `Collection`*):*

```
List list = Collections.synchronizedList (new ArrayList(...));
```

*in tal caso, pur se in generale le prestazioni di un* `Vector` *sono inferiori rispetto a quelle di un* `ArrayList`*, la situazione si capovolge: il* `Vector` *ha performance superiori a quelle di un* `ArrayList` *sincronizzato (modificato in modo che non possa accede a risorse finché esse sono impegnate da altri metodi).*

## *Appendice: classe cronometro*

**All Classes**

Cronometro

### Class Cronometro

```
java.lang.Object
  └ Cronometro
```

---

```
public class Cronometroextends java.lang.Object
```

Classe di utilità che realizza un cronometro per misurare intervalli temporali espressi in millisecondi. Basata sull'orologio di sistema (vedere `System.currentTimeMillis()`).

La precisione delle misure di tempo effettuate dipende dunque dall'accuratezza con la quale è realizzato il *timer* di sistema Java: esperimenti pratici rivelano che in ambiente Windows la sensibilità di `System.currentTimeMillis()` è di 10 ms, mentre su stazioni SUN Sparc, ad esempio, è di 1 ms.

Corretta anche in situazioni *multi-threading*.

**Constructor Summary**

| | |
|---|---|
| `Cronometro()` | Costruttore: resetta il cronometro invocando il metodo d'istanza `azzera()`. |

**Method Summary**

| | |
|---|---|
| void | `avanza()` Metodo che fa (ri)partire il conteggio. |
| void | `avanzaDaCapo()` Azzera il cronometro e ne fa partire il conteggio. |
| void | `azzera()` Metodo per (fermare ed) azzerare del cronometro. |
| void | `ferma()` Metodo che blocca l'avanzamento del cronometro. |
| long | `leggi()` Lettura del conteggio corrente effettuato dal cronometro. |
| java.lang.String | `toString()` Conversione in stringa del conteggio corrente. |

### *Listing*

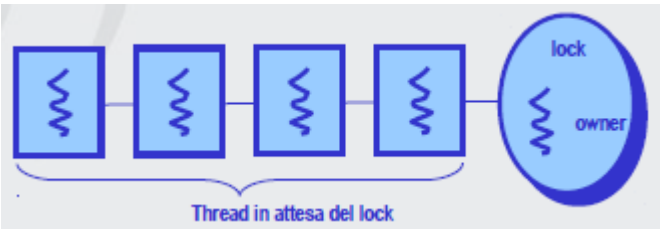
```
/**
 * Classe di utilità che realizza un cronometro per misurare
 * intervalli temporali espressi in millisecondi. Basata sull'orologio
 * di sistema (vedere <code>System.currentTimeMillis()</code>).
 * <p>
 * La precisione delle misure di tempo effettuate dipende dunque
 * dall'accuratezza con la quale è realizzato il <i>timer</i> di
 * sistema Java: esperimenti pratici rivelano che in ambiente Windows
 * la sensibilità di <code>System.currentTimeMillis()</code> è di
 * 10 ms, mentre su stazioni SUN Sparc, ad esempio, è di 1 ms.
 * <p>
 * Corretta anche in situazioni <i>multi-threading</i>.
 * <p>
 */
public class Cronometro {

  /** Accumulatore contenente il numero dei millisecondi trascorsi. */
  private long contatore;
```

```
/** Istante temporale dell'ultimo avvio del cronometro. */
private long avviato_a;

/** Variabile di stato che indica se il cronometro sta avanzando oppure no. */
private boolean avanzando;

/**
 * Costruttore: resetta il cronometro invocando il metodo d'istanza
 * <code>azzera()</code>. Non avvia il conteggio; per fare ciò usare
 * i metodi <code>avanza()</code> ed <code>avanzaDaCapo()</code>.
 *
 * @see   #azzera()
 * @see   #avanza()
 * @see   #avanzaDaCapo()
 */
public Cronometro() { azzera(); }

/** Metodo per (fermare ed) azzerare del cronometro. */
public void azzera() {
  synchronized (this) {
    contatore = 0;
    avanzando = false;
  }
}

/**
 * Metodo che fa (ri)partire il conteggio. Non azzera il
 * cronometro, ma fa procedere la misura del tempo partendo dal
 * valore immagazzinato nell'accumulatore.
 * <p>
 * Il cronometro può essere fermato mediante <code>ferma()</code>.
 *
 * @see   #ferma()
 */
public void avanza() {
  synchronized (this) {
    avviato_a = System.currentTimeMillis();
    avanzando = true;
  }
}

/**
 * Metodo che blocca l'avanzamento del cronometro. Usare
 * <code>avanza()</code> per far ripartire il conteggio,
 * <code>avanzaDaCapo()</code> per azzerare il tutto prima di
 * dare inizio al conteggio.
 *
 * @see   #avanza()
 * @see   #avanzaDaCapo()
 */
public void ferma() {
  synchronized (this) {
    contatore += System.currentTimeMillis() - avviato_a;
    avanzando = false;
  }
}
```

```
/** Azzera il cronometro e ne fa partire il conteggio. */
  public void avanzaDaCapo() {
    azzera();
    avanza();
  }

 /**
   * Lettura del conteggio corrente effettuato dal cronometro.
   * Chiamate successive a questo metodo riportano valori diversi
   * nel caso in cui il cronometro stia avanzando.
   *
   * @return   il numero totale di millisecondi contati dall'istanza.
   */
  public long leggi() {
    synchronized (this) {
      return avanzando ? contatore + System.currentTimeMillis() - avviato_a : contatore;
    }
  }

  /**
   * Conversione in stringa del conteggio corrente. La lettura del
   * valore viene effettuata mediante il metodo <code>leggi()</code>.
   *
   * @return   una stringa rappresentante il numero di millisecondi
   *           contati dall'istanza in questione.
   * @see      #leggi()
   */
  public String toString() {
    return "" + leggi();
  }
}
```

Sincronizzazione: synchronized (this) {}

ricordando che, se un thread acquisisce un *lock* su un oggetto (in questo caso l'oggetto è l'attuale), nessun altro thread può acquisire un lock sullo stesso oggetto




>> multithreading in Java e blocco synchronized (..) {} per rendere "atomiche" una serie di istruzioni rispetto alla concorrenza tra i thread. In sostanza impone una "serializzazione" tra i thread per la esecuzione di quel blocco di codice sincronizzato.