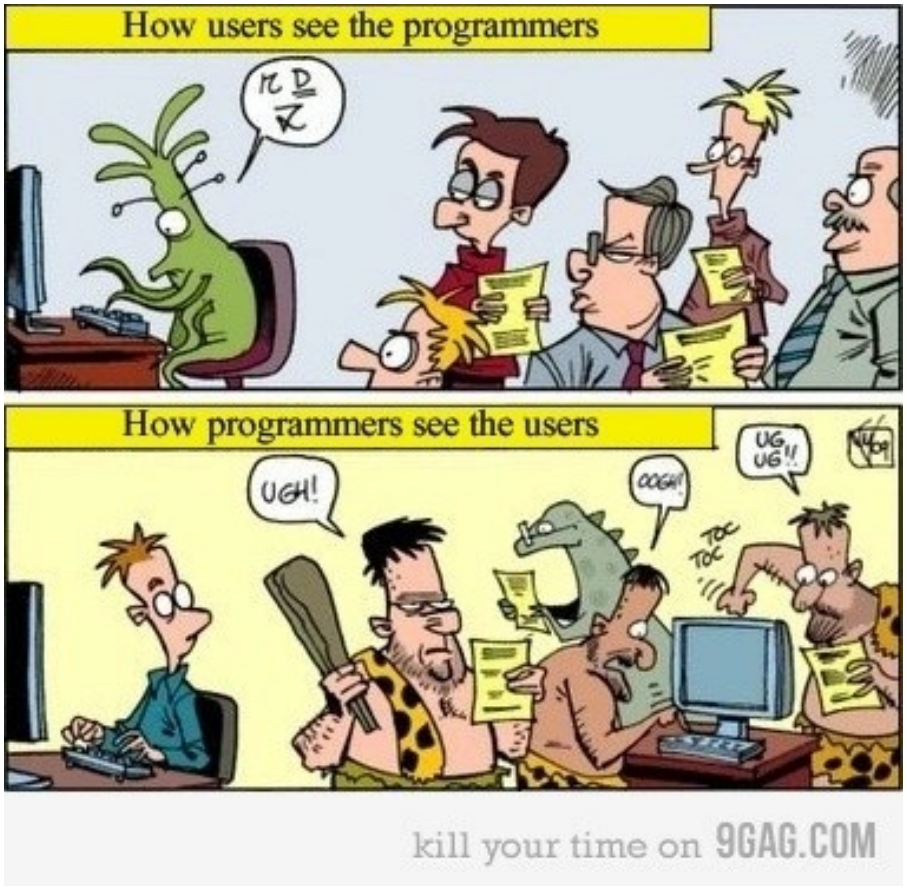
Strutture Dati Dinamiche

Prof. Francesco Viglietti
Www.in4matika.altervista.org



ADT(Abstrat Data Type)

Specifica un insieme di dati e le operazioni possibili su di esso, ma non come implementarle o come memorizzare i dati.
Una struttura dinamica offre la possibilità di adattarsi al variare della dimensione dei dati da trattare, quindi la loro dimensione può variare continuamente.

Operazioni tipiche:

- **operazioni di modifica**: inserimento modifica ed eliminazione
- **operazioni di interrogazione**: ricerca di un certo elemento nella struttura.

Array dinamici

L'array dinamico è un vettore di oggetti senza una dimensione prefissata, può aumentare o diminuire in base alle necessità.

Classe **Vector** e classe **ArrayList**, incluse nel package **java.util**

Essendo un vettore di oggetti devo istanziare il vettore ed ogni singolo oggetto!

Vantaggi: dimensione variabile evita spreco di memoria

Svantaggi: efficienza rispetto ad Array, e memorizzazione di soli oggetti

Array dinamici

Tre diversi costruttori per ciascuna struttura:

```
Vector   v = new Vector();
ArrayList a  = new ArrayList();
Vector   v = new Vector(5);
ArrayList a  = new ArrayList(5);
Vector   v = new Vector(5,2);
ArrayList a  = new ArrayList(5,2);
```

Il primo crea un vettore senza capacità iniziale, il secondo crea un vettore con capacità iniziale di 5 elementi, il terzo crea un vettore con capacità iniziale di 5 elementi e quando si occupano questi, vengono aggiunti gruppi da due elementi per volta.

Classi wrapper

Possono essere memorizzati solo elementi di tipo Object.

Per ovviare a questo, se volessimo usare tipi primitivi (int, char, double) si fa ricorso alle classi **wrapper** (involucro), che definiscono i metodi che possono agire sui valori di un tipo primitivo.

Converte un valore di tipo **primitivo** in tipo **wrapper**

Integer n=new Integer (valore); //classe wrapper di int
Double d=new Double (valore); //classe wrapper double
Character c=new Character (valore); //classe wrapper char

Converte un valore di tipo **wrapper** in tipo **primitivo**

int i=n.intValue() ...

Classi wrapper

Ogni tipo primitivo ha una corrispondente classe wrapper. Esse permettono di avere un oggetto di tipo classe che corrisponde a un valore ti tipo primitivo.
Le classi wrapper non hanno un costruttore di default!
Quindi la dichiarazione dev'essere del tipo:

```
Character c=new Character('f');
```

oppure

```
Character c='f';
```

Le classi wrapper contengono anche costanti e metodi statici molto utili...

List in java

Un oggetto List contiene oggetti ordinati in base all'ordine di inserimento, può contenere duplicati e permette di inserire e ottenere gli elementi in base all'indice. E' la versione "evoluta" dell'Array in quanto non contiene la limitazione della dimensione massima prefissata.

Vector: è una implementazione di List, è simile ad ArrayList ma sincronizzata (può essere usata da più thread paralleli senza problemi di concorrenza). Però viene consigliato l'uso di arrayList.

List in java

ArrayList: è l'implementazione di List che memorizza gli elementi in un Array, si occupa della gestione della dimensione dell'Array in modo trasparente per lo sviluppatore. Se si conosce già la dimensione massima può essere specificata nel costruttore. Visto che i dati sono memorizzati in un Array l'accesso all'i-esimo elemento è veloce. Per lo stesso motivo l'operazione di rimozione di un elemento è lenta, tutti gli elementi successivi all'elemento da eliminare devono essere spostati di una posizione.

LinkedList: è l'implementazione di List che usa una lista concatenata bidirezionale, permette di scorrere gli elementi partendo sia dall'inizio che dalla fine. Accedere all'i-esimo elemento è un'operazione lenta, infatti devono essere percorsi tutti gli elementi precedenti; l'operazione di rimozione è invece veloce, in quanto non devono essere spostati oggetti, ma prevede solo il cambiamento di alcuni collegamenti fra i nodi della lista.

Classe Vector

Metodi principali:

- **addElement**(Object obj) aggiunge un oggetto al vettore
- **removeElementAt**(**int** index) rimuove l'oggetto alla posizione specificata e sposta tutti gli oggetti successivi. (Indice varia per tutti gli elementi spostati)
- **Size()** restituisce il numero di elementi del vettore
- **elementAt**(**int** index) restituisce l'oggetto (di classe Object) alla posizione specificata dall'indice, quindi è necessario il casting per riportarlo alla classe originale
- **contains**(Object obj) restituisce vero le l'oggetto è presente nella lista.

Classe Vector

Abbiamo detto che la classe Vector contiene oggetti! E se volessi inserire numeri interi o reali?
Dovrei usare le classi **wrapper** Integer o Double che sono sottoclassi della classe Object!
Ad esempio voglio inserire dei numeri reali (num) in un vettore. Devo istanziare la classe wrapper **Double**:
Inserisce l'elemento num reale in un vettore v

```
double num; //dichiaro variabile double
obj = new Double(num); //inserisco il valore num nell'oggetto
v.addElement(obj); //lo aggiungo al Vector
```

Quando devo estrarre un elemento da un vettore e assegnarlo ad un valere reale devo ricorbarmi il casting:

```
obj= (double)v.elementAt(int index); //estraggo l'elemento dal vector
num=obj.doubleValue()//assegno l'oggetto alla variabile double
```

Classe ArrayList

Abbiamo detto che anche la classe ArrayList contiene oggetti! Quindi se volessi usare tipi primitivi, anche in questo caso avrei necessità delle classi **wrapper**!
Dichiarazione e istanziazione di oggetto ArrayList:
ArrayList<*tipo base*> variabile=**new** ArrayList<*tipo base*>();
Dove <tipo base> dev'essere una classe non può essere un tipo primitivo!
METODI PRINCIPALI:
add(Object obj) Inserisce obj come ultimo elemento di questa lista
clear() Rimuove tutti gli elementi da questa lista
contains(Object obj) Verifica se questa lista contiene almeno un elemento uguale a obj.
get(int i) restituisce l'elemento di questa lista di posizione i.
indexOf(Object obj) Calcola l'indice del primo elemento di questa lista uguale a obj, oppure -1 se la lista non contiene nessun elemento uguale a obj.

Classe ArrayList

Ancora metodi...
isEmpty() Verifica se questa lista è vuota.
lastIndexOf(Object obj) Calcola l'indice dell'ultimo elemento di questa lista uguale a obj, oppure -1 se la lista non contiene nessun elemento uguale a obj.
remove(int i) Rimuove da questa lista l'elemento di posizione i e lo restituisce.
remove(Object obj) Rimuove da questa lista il primo elemento uguale a obj, se presente (opzionale).
size() Calcola la lunghezza di questa lista.
toArray() Restituisce un array che contiene gli elementi di questa lista, ciascuno nella posizione in cui compare nella lista.
toString() Restituisce una descrizione di questa lista.

Classe ArrayList

Per leggere tutti gli elementi della Lista è comodo usare for-each:

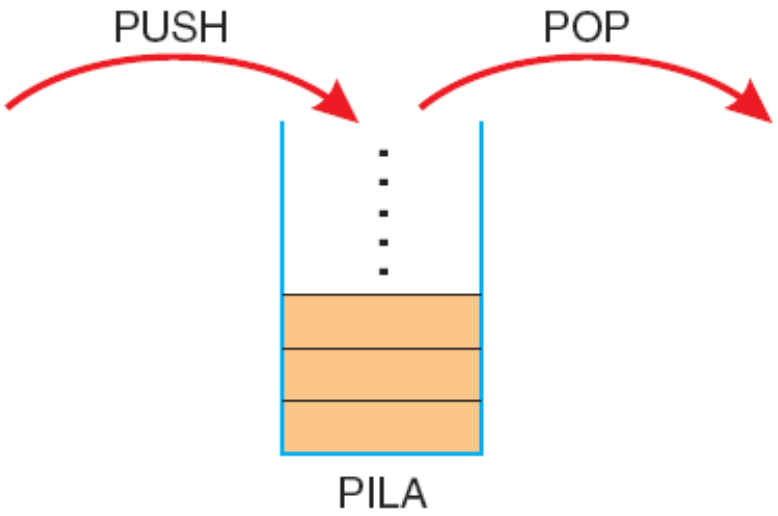
```
for (Object elemento : lista)
    System.out.print(elemento);
```

Esempio con tipi primitivi:

```
ArrayList dati = new ArrayList(); //istanziazione ArrayList
int n = 30; //dichiarazione variabile intera
Integer numero = new Integer(n); //uso classe wrapper
dati.add(numero); //aggiungo alla lista
Integer numero = (Integer)dati.get(0); //estraggo dalla lista
int n = numero.intValue(); //assegno a variabile tipo primitivo
```

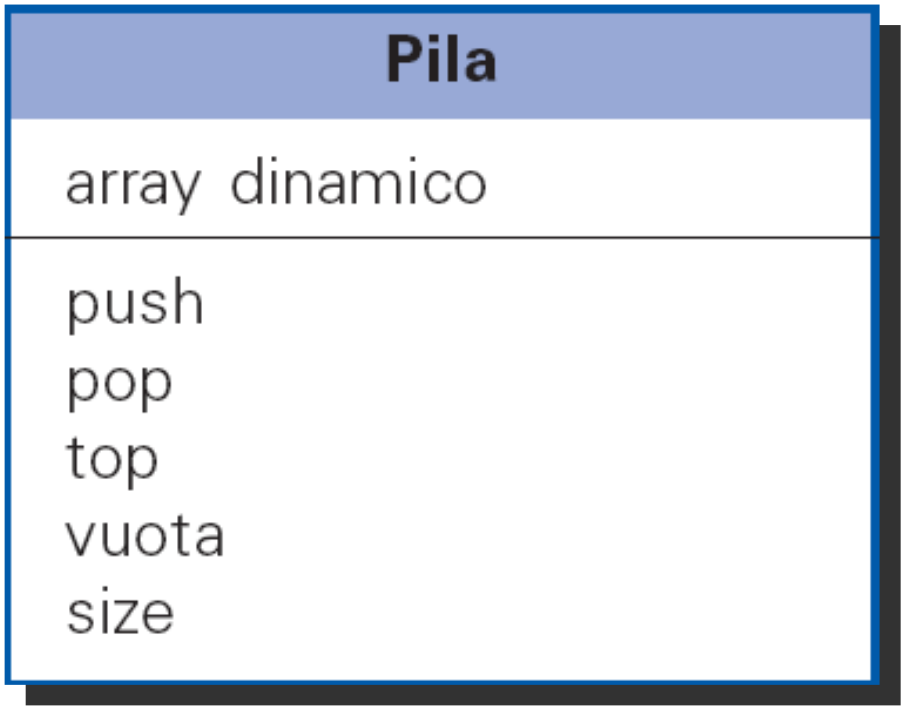
Pila (Stack)

La **pila** (**stack**) è una struttura di dati dinamica gestita usando la modalità **LIFO** (Last-In First-Out): l'inserimento e l'estrazione dei dati avviene da un'unica estremità.



Pila

Diagramma di classe:



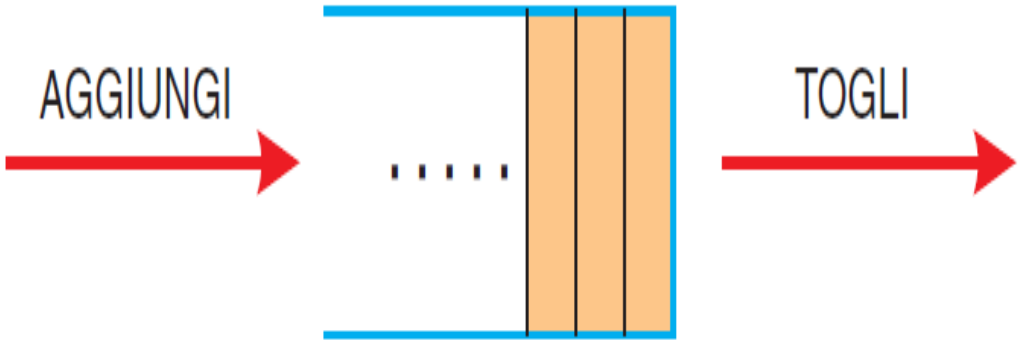
Classe Pila in java

```
public class Pila{
      private Vector elementi;
//costruttore
public Pila() {
      elementi = new Vector();
}
//metodi
public void push (Object obj {
      elementi.addElement(obj);
}
public Object pop() {
      Object obj = null;
      int size = elementi.size();
      if (size > 0){
      obj = elementi.elementAt(size-1);
      elementi.removeElementAt(size-1);
      }
return obj;
}
```

```
public Object top(){
      Object obj = null;
      int size = elementi.size();
      if (size > 0)
          obj = elementi.elementAt(size-1);
      return obj;
      }
public boolean vuota(){
      if (elementi.size() > 0)
            return false;
      else
       return true;
     }
public int size(){
      return elementi.size();
     }
}
```

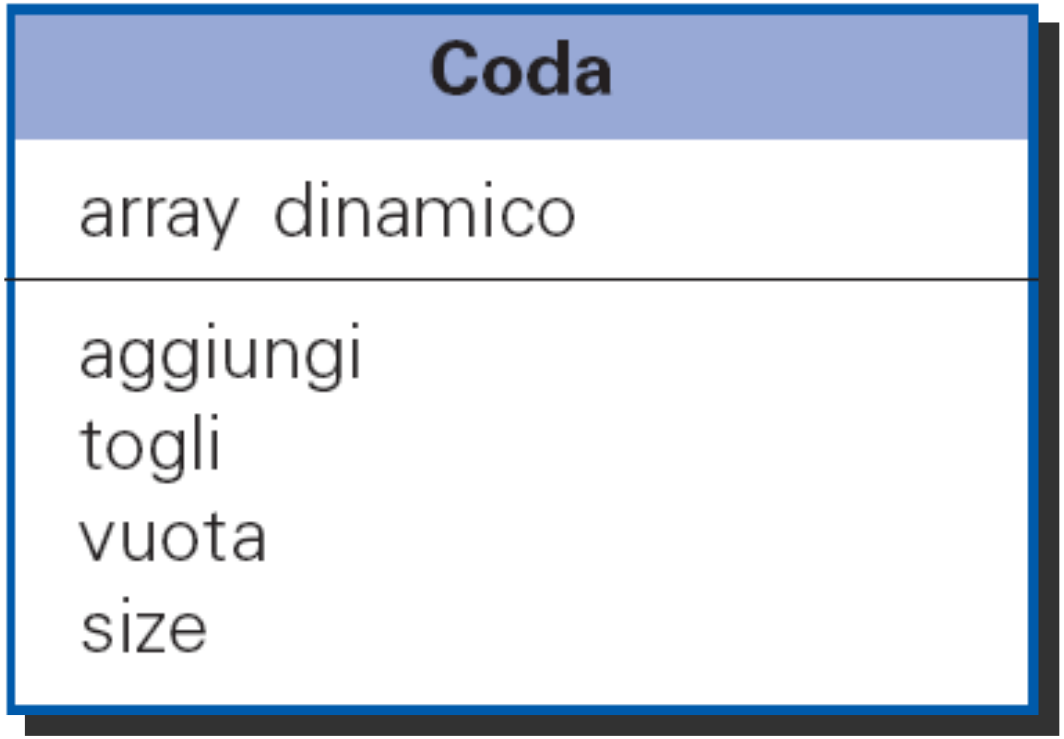
Coda (Queue)

La **coda** (**queue**) è una struttura dinamica di dati gestita usando la modalità **FIFO** (First-In First-Out): l'inserimento avviene da un'estremità e l'estrazione dall'altra.



Coda

Diagramma di classe:



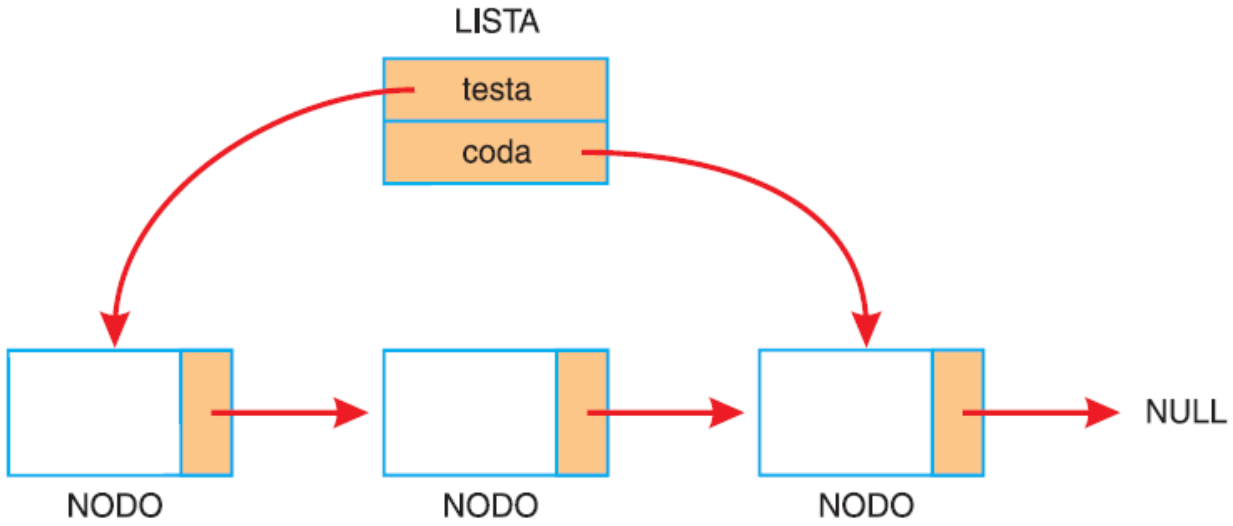
Classe Coda in java

```
class Coda{
private Vector elementi;
public Coda(){ elementi = new Vector();}
public void aggiungi(Object obj){elementi.addElement(obj);}
public Object togli(){
  Object obj = null;
  int size = elementi.size();
  if (size > 0){
      obj = elementi.elementAt(0); elementi.removeElementAt(0);
}
return obj;}
public boolean vuota(){
  if (elementi.size() > 0)
      return false;
  else
      return true;
}
public int size(){ return elementi.size(); }
}
```

Liste concatenate

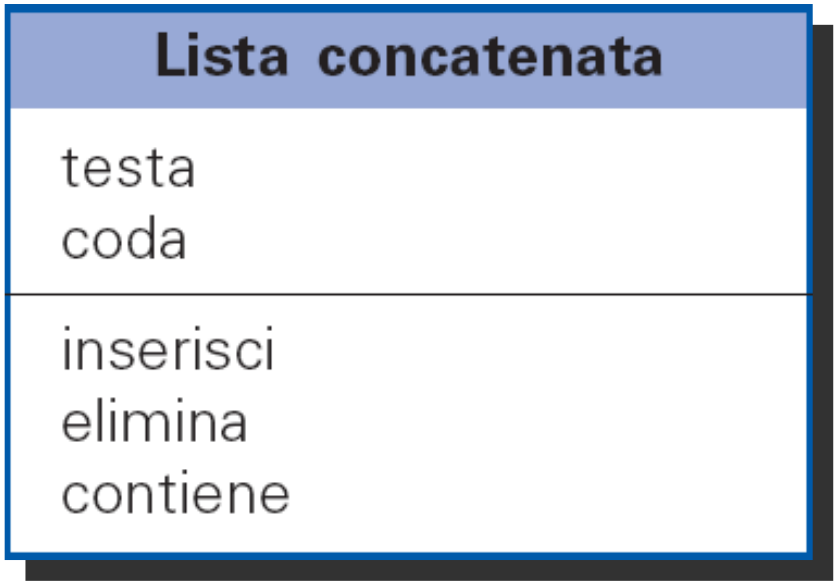
La lista concatenata (**LinkedList**) è una struttura di dati in cui gli elementi sono ordinati linearmente.
Ogni elemento conosce qual è il suo successore:
a partire dal primo elemento è possibile ricostruire tutti gli elementi presenti nella lista.

In Java vengono utilizzati i **riferimenti agli oggetti:**



Liste concatenate

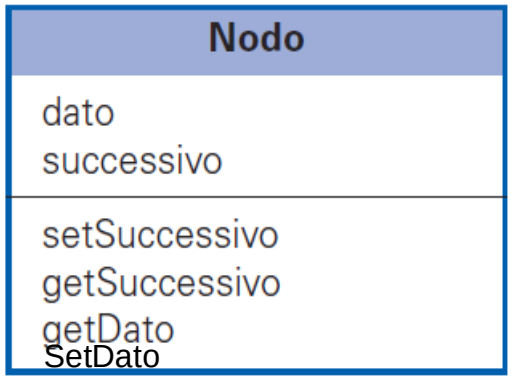
Diagramma di classe:



L'attributo ***testa*** si riferisce al primo nodo della lista.
L'attributo ***coda*** si riferisce invece all'ultimo nodo.
I metodi ***inserisci, elimina*** eseguono le operazioni in base alla scelta dell'utente (operazione in coda, in testa o nell'elemento i-esimo). Il metodo ***contiene*** invece determina la presenza o meno nella lista dell'elemento specifico.

Liste concatenate

Essendo la lista formata da nodi, bisogna implementare la classe Nodo.
Essa è formata dagli attributi dato e successivo che contengono rispettivamente il dato e il riferimento al nodo successivo nella lista.
I metodi xetSuccessivo implementano l'impostazione e la conoscenza del nodo successivo. getDato invece permette di leggere il dato memorizzato nel nodo.



Questo è solo un esempio di codice per la classe nodo,

```
class Nodo {
 private String dato;
 private Nodo successivo;
 public Nodo(){
  dato=null;
  successivo=null;
 }
 public void setSuccessivo(Nodo successivo
{ this.successivo = successivo; }
public Nodo getSuccessivo(){ return successivo; }
public String getDato()  { return dato; }
public void setDato(String dato){ this.dato=dato; }
}
```

LinkedList in java

Classe generica: il tipo degli elementi della lista va specificato tra parentesi angolari, come: LinkedList<String>, LinkedList<Product>

- Fa parte del pacchetto java.util
- Accesso diretto sia al primo sia all'ultimo che a qualsiasi altro elemento della lista

Il metodo ListIterator:

- dà accesso agli elementi all'interno di una lista concatenata
- incapsula il concetto di posizione in un qualsiasi punto entro la lista.

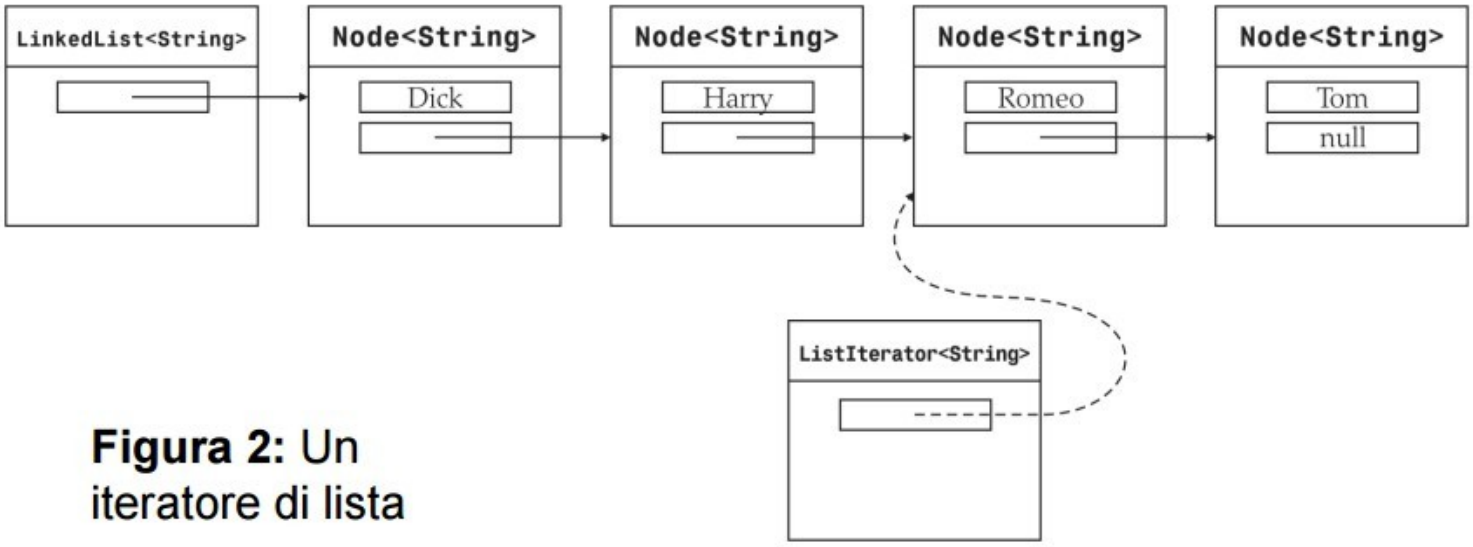


Figura 2: Un iteratore di lista

LinkedList in Java

La LinkedList quindi è una lista concatenata e ordinata in cui la posizione dei singoli elementi è importante, inoltre è doppiamente concatenata. La classe LinkedList appartiene alle collezioni. In una lista concatenata è difficile accedere a posizioni intermedie, per cui bisogna seguire una serie di collegamenti tra gli elementi per scorrere l'intera lista. Si tratta di un processo poco efficiente e per questa motivo viene utilizzato un iteratore, esso contiene i seguenti metodi:
add(Object obj) : aggiunge un oggetto obj in coda alla lista.Restituisce un booleano, nonostante per le LinkedList il risultato sarà sempre true.
contains(Object obj) : restituisce true se la lista contiene un oggetto y tale che obj.equals(y) è vero.
size() : restituisce la dimensione della lista.

LinkedList in Java

addFirst(Object obj) : aggiunge obj in testa alla lista.
addLast(Object obj) : è equivalente di add(obj), ma aggiunge obj in coda.
removeFirst() : rimuove e restituisce la testa della lista. Solleva l'eccezione NoSuchElementException se la lista è vuota.
removeLast() : rimuove l'ultimo elemento e restituisce la coda della lista. Solleva l'eccezione NoSuchElementException se la lista è vuota.
Questi ultimi quattro metodi consentono di realizzare strutture come stack , liste, code, ecc.
Nota: LinkedList eredita da List i metodi get e set, metodi che vengono utilizzati per accedere ad un elemento in posizione determinata nella lista. I metodi fanno ripartire ogni volta l'iterazione dalla posizione iniziale. Per questo motivo, anche se presenti, è fortemente sconsigliato utlizzare i metodi get e set con una LinkedList.

Iterator

La classe Iterator è il modo standard per scorrere gli elementi di una qualunque Collection. Contiene 3 metodi principali: ***hasNext()*** ritorna true se ci sono ancora elementi da scorrere; ***next()*** ritorna il successivo elemento; ***remove()*** elimina dalla collection l'ultimo elemento ritornato. Questo metodo deve essere usato quando si vuole rimuovere un oggetto da una Collection durante un ciclo sugli elementi. Infatti in questi casi chiamare il metodo remove di Collection causa una ConcurrentModificationException al successivo accesso alla lista

Senza iterator una lista potrebbe essere visitata così:

```
public static int sumLessThen(List l, int max) {
    int tot = 0;
    for (int i=0; i<l.size(); i++) {
        Integer tmp = l.get(i);
        if (tmp!=null && tmp.intValue()<max)
            tot+=tmp;
    }
return tot;
}
```

Iterator

Se la lista passata è una LinkedList questo metodo è poco efficiente (per ogni elemento il metodo get riparte dall'inizio).

Usando un iterator il metodo può essere riscritto come:

```
public static int sumLessThen(List l, int max) {
    int tot = 0;
    for (Iterator iterator = l.iterator(); iterator.hasNext();) {
        Integer tmp = iterator.next();
        if (tmp!=null && tmp.intValue()<max)
            tot+=tmp;
    }
return tot;
}
```