STRUTTURE DATI: OLTRE GLI ARRAY

- **le strutture dati progettate per ospitare una *collezione di elementi,* sono variazioni di array**
- **Ma l'array ha dimensione fissa anche in Java**
  - **determinata *a priori*, nei linguaggi statici**
  - **determinata *al momento della creazione*, se definiti dinamicamente (Java)**
- **Molti problemi richiedono strutture dati in grado di rappresentare collezioni di elementi *di numero variabile dinamicamente***
- **Efficienza nell'uso delle risorse**

LISTE

- **Una *lista* è una *collezione di elementi* organizzati concettualmente *in sequenza***
- **La dimensione della *lista non* è *prefissata***
- **Gli elementi sono aggiunti su necessità e occupano solo memoria quando necessario**

Progetto in Java

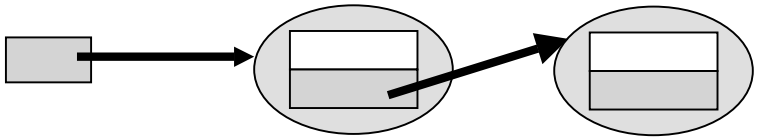
- **Uso della semantica per riferimento**

Decisioni progettuali

- **Una lista *come contenitore* di valori**

LISTE: RAPPRESENTAZIONE

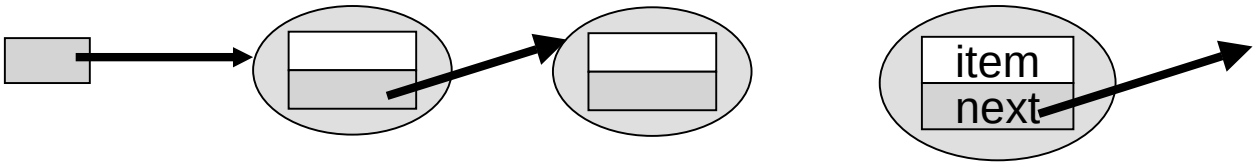
- **Una *lista* è spesso rappresentata sotto forma di *sequenza di nodi concatenati***



- **Ogni *nodo* della lista è fatto di *due parti*: un *valore* e un *riferimento al nodo successivo***
- **La *rappresentazione fisica* può essere:**
  - **basata su puntatori e allocazione dinamica**
  - **basata su oggetti creati dinamicamente**
  - **basata su altre rappresentazioni (file, array, …)**

LISTE

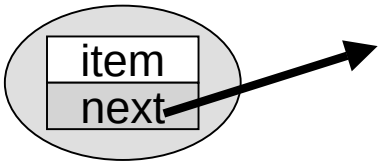
- **Consideriamo un generico elemento**




```
public class ListInt {
  private ListInt next;
  private Integer item;

  public ListInt()
   { next = null; item = null;}
...
}
```

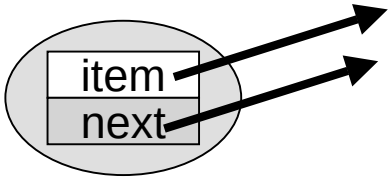
LISTE: funzionalità

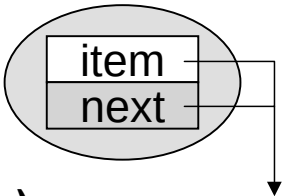
```
public class ListInt {
  private ListInt next;
  private Integer item;

public ListInt() {}
public void  insert (int i) {}
public Integer extract () {}

public boolean isNotEmpty () {}
public void printList () {}
}
```



LISTE: funzionalità

```
public class ListInt {
  private ListInt next;
  private Integer item;

public ListInt() {next = null; item = null;}
}
```

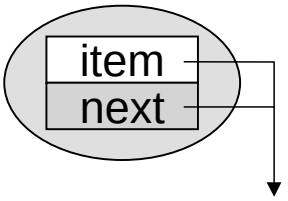


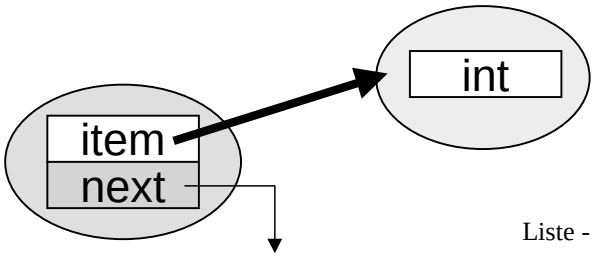


L'elemento creato non riferisce nulla
Si noti che consideriamo di riferire un altro (next) e un wrapper Integer (item)
Inserimenti LIFO

LISTE: inserimento

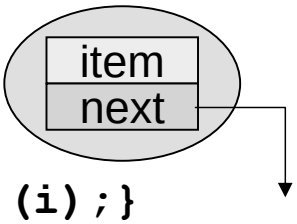
```
public class ListInt {
  private ListInt next;
  private Integer item;
public void  insert (int i) {
 if (item==null){item = new Integer (i);}
 else {
  ListInt temp = next;
  next = new ListInt();
  next.item = item; item = new Integer (i);
  next.next = temp;
     }
}
```

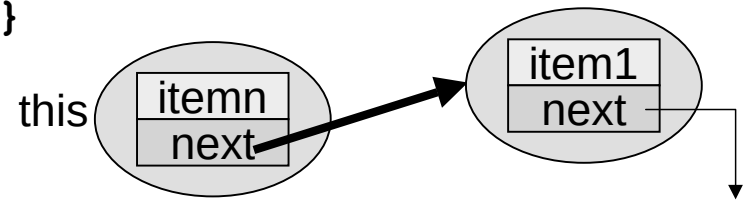


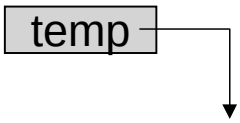


LISTE: inserimento

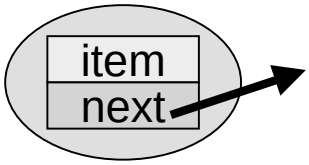
```
public class ListInt {
public void  insert (int i) {
 if (item==null){item = new Integer (i);}
 else { // item non nullo
  ListInt temp = next; // temp e next null
  next = new ListInt();
  next.item = item; item = new Integer (i);
  next.next = temp;
     }
}
```



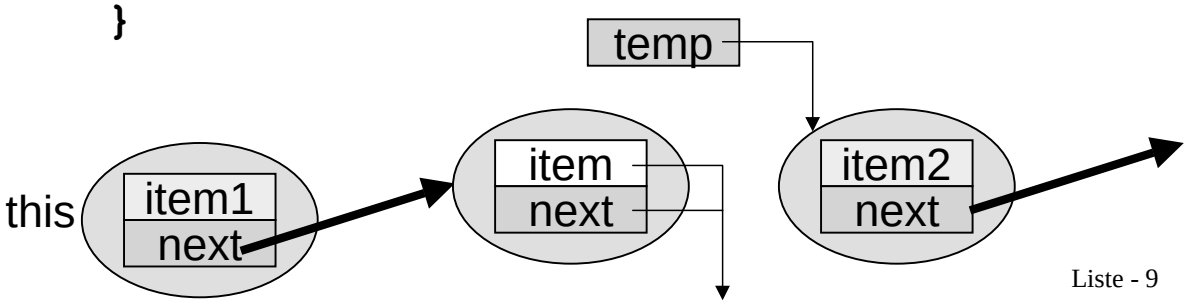




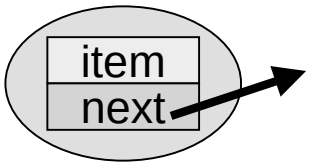
LISTE: inserimento



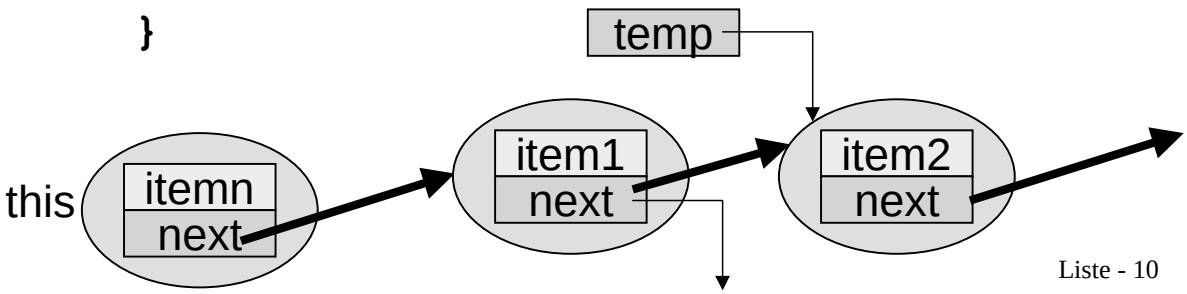

```
public class ListInt {
public void  insert (int i) {
 if (item==null){item = new Integer (i);}
 else { // item non nullo
  ListInt temp = next; // temp non null
  next = new ListInt();
  next.item = item; item = new Integer (i);
  next.next = temp;
      }
}
```



LISTE: inserimento




```
public class ListInt {
public void  insert (int i) {
 if (item==null){item = new Integer (i);}
 else { // item non nullo
  ListInt temp = next; // temp non null
  next = new ListInt();
  next.item = item; item = new Integer (i);
  next.next = temp;
      }
}
```



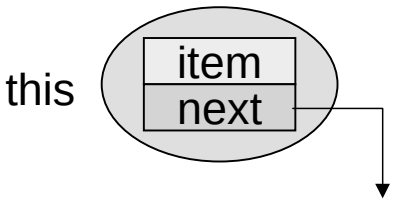
LISTE: estrazione

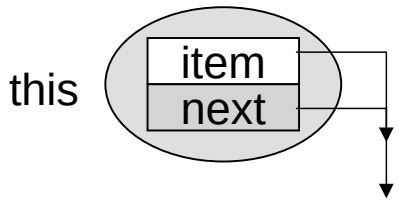
```
public Integer extract () { Integer iI;
 if (item == null) {return null;}
 else
 { iI = item;
   if (next == null) item = null;
   else {item = next.item;
         next = next.next;}
   return iI;
}
```

Si noti il ruolo di Integer (wrapper) per indicare che non si restituisce alcun valore

LISTE: estrazione

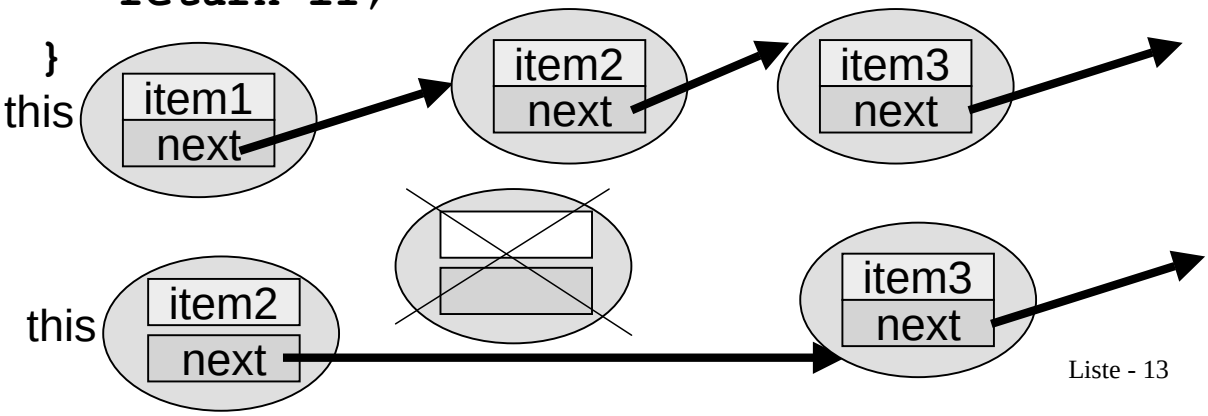
```
public Integer extract () { Integer iI;
 … else
 { iI = item;
   if (next == null) item = null;
   else {item = next.item;
         next = next.next;}
   return iI;
}
```





LISTE: estrazione

```
public Integer extract () { Integer iI;
 … else
 { iI = item;
   if (next == null) item = null;
   else {item = next.item;
         next = next.next;}
   return iI;
}
```



LISTE: funzionalità

```
public class ListInt {
 private ListInt next; private Integer item;
public ListInt() {...}
public void  insert (int i) {...}
public Integer extract () {...}

public boolean isNotEmpty ()
   {return (item != null);}
public void printList () {}
// stampa la lista in formato esterno
}
```

LISTE: printList

```
public void printList () {
ListInt temp = next; int i = 1;
if (item == null) System.out.println (" Lista vuota ");
else
{System.out.println(" Elem N.1 vale "+ item);
 while (temp != null)
  { i++;
    System. out. println (" Elem N." + i +
                " vale " + temp.item);
    temp = temp. next;
  }
}}
```


LISTE: toString

```
public String toString () {
ListInt temp = next; int i = 1; String s;
if (item == null) return " Lista vuota \n";
else
{s = "\n Lista con elementi " + item;
 while (temp != null)
 { s = s + "\t" + temp.item;
   temp = temp.next;  i++; }
 return  s = s + "\n In totale  sono "
               + i + " elementi\n";
}
}
```

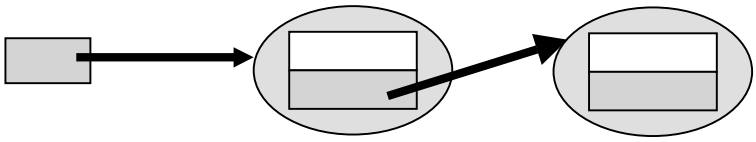
USO da parte di CLIENTI

```
public static void main(String[] args) {
  int i;  Integer iI;
ListInt first = new ListInt ();
first.insert(2); first.insert(6);
first.insert(17);
iI = first.extract();
if (iI != null) { i = iI.intValue(); ...}
first.insert (19);
first.printList ();
while (first.isNotEmpty ())
 { i = first.extract().intValue();}
}
```


PROGETTO DI LISTE IN JAVA

`List`: quale rappresentazione?

- **approccio classico:**
  - **un campo `value` che rappresenta il valore (`Object`)**
  - **un campo `next` che punta al prossimo nodo (`List`)**



- **semplice, ma con alcuni difetti**
  - **la lista non esiste come entità esplicita (coincide con il riferimento al primo nodo)**
  - **difficili variazioni con semantica diversa da quella predefinita (inserimento in testa)**

PROGETTO DI LISTE IN JAVA

`List`: quale

- **approccio**
  - **un campo [illegible] valore (`Object`)**
  - **un campo `next` che punta al prossi[illegible]do (`List`)**

Poiché il valore è un `Object`, **la lista può ospitare solo oggetti** (non variabili di tipi primitivi)

Al posto dei tipi primitivi occorre dunque utilizzare le corrispondenti **classi wrapper** (`int` → `Integer`, `float` → `Float`, etc.)

- **semplice, ma con alcuni difetti**
  - **la lista non esiste come entità esplicita (coincide con il riferimento al primo nodo)**
  - **ciò rende difficile avere semantiche di costruzione diverse da quella predefinita (inserimento in testa)**

ANALISI CRITICA DEL PROGETTO

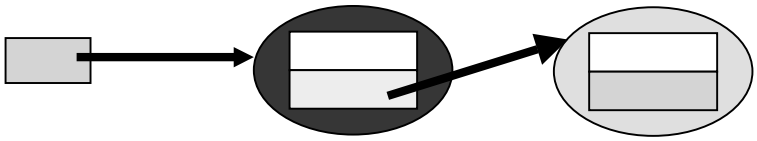
L'approccio classico è soddisfacente?

- **non del tutto, in quanto *la lista non esiste come entità esplicita*: coincide con il riferimento al primo nodo**
  - **la classe `List` in realtà rappresenta *il nodo della lista*, non la lista in quanto tale!**
- **ciò rende impossibile avere semantiche di costruzione *diverse da quella predefinita* (inserimento in testa)**
  - **ad esempio, è impossibile derivare da `List` una classe `SortedList` che costruisca *liste ordinate***

ANALISI CRITICA DEL PROGETTO

Perché?

- **Perché il costruttore di una tale classe `SortedList` *chiamerebbe come prima cosa il costruttore della classe base* `List`, che crea già un nuovo nodo e lo inserisce in testa!**

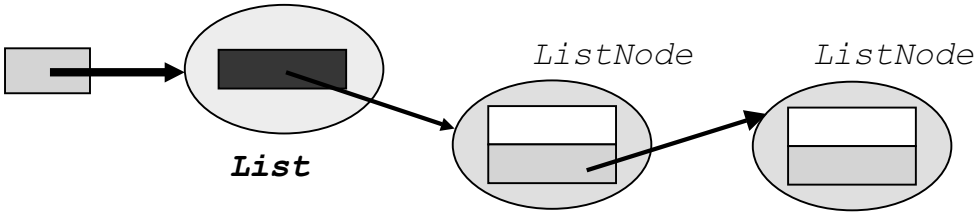


- **Nessuno può revocare quel comportamento, *indipendentemente dal costruttore di* `SortedList`**
- **Il cliente riceverebbe *in ogni caso* il riferimento al nuovo nodo creato in testa**

REVISIONE DEL PROGETTO

Una diversa rappresentazione

- **`List` rappresenta la lista: è un oggetto che contiene *la radice della sequenza di nodi***
  - **tale radice è *un riferimento a un oggetto* `ListNode`**
- **`ListNode` rappresenta il nodo della lista**
  - **contiene il valore e il riferimento al nodo successivo**
  - **è quello che prima chiamavamo impropriamente lista**



REVISIONE DEL PROGETTO

Perché questo approccio risolve il problema?

- **Perché ora creare una `List` *non implica più* crearne anche un nodo!**
- **Quindi, il costruttore di `List` può gestire direttamente la creazione del nodo, *inserendolo nella posizione più appropriata***
- **Ridefinendo il proprio costruttore, una sottoclasse `SortedList` può cambiare tale comportamento**
- **Il cliente riceve *in ogni caso* il riferimento alla nuova lista creata, non già a uno specifico nodo!**

Nuovo Progetto Lista Semplice

```
public class ListNode implements Iposition {
// Implementazione dell'interfaccia "IPosition"
protected Object item; protected ListNode next;
public ListNode(Object o) { item = o; next=null;}

public Object getItem(){ return item;}
public Object container() {return null; }
// modificare per riferire la lista cui nodo appartiene

public ListNode getNext() { return next;}
public void setNext(ListNode next){ this.next=next;}
}
```

INTERFACCIA per NODO di LISTA

```
public interface IPosition
{// questa interfaccia per singolo elemento

 public Object getItem ();
 public Object container();
 // questo elemento permette al nodo di puntare
 // alla lista cui il nodo appartiene
 // non implementato: fare per esercizio
}
```

La Interfaccia consente ad un nodo di trovare il prossimo e di riferire una ed una sola lista

INTERFACCIA LISTA

```
public interface IList // interfaccia per la lista
{
// Inserimento oggetto a inizio e a fine lista
public void insert(Object o);
public void append(Object o);
// Rimozione da inizio e fine lista
public Object removeFirst() throws ListEmptyException;
public Object removeLast() throws ListEmptyException;
// Restituire il primo elemento e ultimo lista
public Object first() throws ListEmptyException;
public Object last() throws ListEmptyException;
// Numero elementi e predicato di lista vuota
public int size();        public boolean isEmpty();
```

LISTA SEMPLICE (link in avanti)

```
public class SimpleList implements IList
{private int size; private ListNode first;
// Inserimento oggetto a inizio e a fine lista
public void insert(Object o)
public void append(Object o)
// Rimozione primo e ultimo elemento
public Object removeFirst() throws ListEmptyException
public Object removeLast() throws ListEmptyException
// Restituire il primo elemento e ultimo lista
public Object first() throws ListEmptyException
public Object last() throws ListEmptyException
// Numero elementi e predicato di lista vuota
public int size()          public boolean isEmpty()
```


Inserimento in prima posizione

```
public class SimpleList implements IList
{private int size; private ListNode first;

// Inserimento oggetto all'inizio della lista
public void insert(Object o)
{ ListNode node = new ListNode(o);
  // ListNode creato con puntatori nulli
  node.setNext(first);
  first = node;
  size++; }
```

Inserimento ultima posizione

```
public class SimpleList implements IList
{private int size; private ListNode first;

// Inserimento alla fine della lista
public void append(Object o)
{ ListNode newNode = new ListNode(o);
  ListNode node = first;
  while (node.getNext()!= null)
     node = node.getNext(); // ultimo elemento
  node.setNext(newNode);
  size++;
}
```


Estrazione dalla prima posizione

```
public class SimpleList implements IList
{private int size; private ListNode first;

// Rimozione primo elemento
public Object removeFirst() throws
 ListEmptyException
{ if(isEmpty()) throw new
     ListEmptyException ("lista vuota!");
  ListNode node = first;
  first = node.getNext(); size--;
  return node.getItem();
}
```

Estrazione dalla ultima posizione

```
public Object removeLast() throws ListEmptyException
{if (isEmpty())
     throw new ListEmptyException("lista vuota!");
 if (first.getNext() == null) // un solo elemento
 {Object o = first. getItem(); first = null;
  return o;}
 else // pnode precede, node scorre lista nodi
 {ListNode node=first.getNext(); ListNode pnode=first;
  while (node.getNext() != null)
       {pnode = node; node= node.getNext();}
  Object o=node.getItem ();
  pnode.setNext(null); size--;
  return node.getItem();
 }
}
```

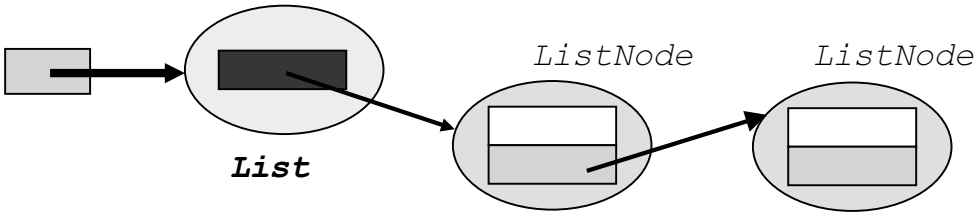

Accessori

```
public Object first() throws ListEmptyException
{if(isEmpty()) throw new ListEmptyException(" vuota!");
 return first.getItem();}

public Object last() throws ListEmptyException
{if(isEmpty()) throw new ListEmptyException("vuota!");
 ListNode node = first;
 while (node.getNext()!= null) node= node.getNext();
 return node.getItem();
}

public int size(){ return size;}
public boolean isEmpty(){ return first==null;}
```

IL PROGETTO REVISIONATO

Ora è semplice derivare la classe `SortedList`

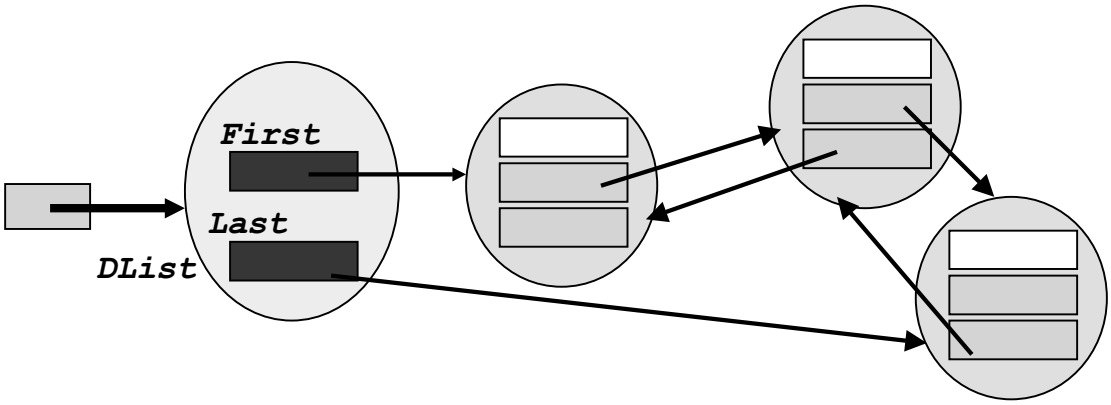
- **una `SortedList` specializza la classe base `SimpleList` nel senso che *il suo costruttore inserisce l'oggetto nella posizione opportuna in base all'ordinamento* anziché sempre in testa**



Consideriamo un Lista Doppia

Possiamo pensare a lista doppiamente linkate

- **ogni nodo è collegato al precedente a al successivo**
- **la lista può accedere anche all'ultimo elemento**



Nodo per Lista Doppia

```
public class DListNode extends ListNode {
// Implementa interfaccia IList
// protected Object item; protected ListNode next;
protected ListNode prev;
public DListNode(Object o)
     { super(o); prev = null;}

public ListNode getPrev() { return prev; }
public void setPrev(ListNode prev)
   { this.prev = prev;}
}
```

Implementa per ereditarietà la interfaccia Ilist e deriva da ListNode i metodi di base

Lista Doppia

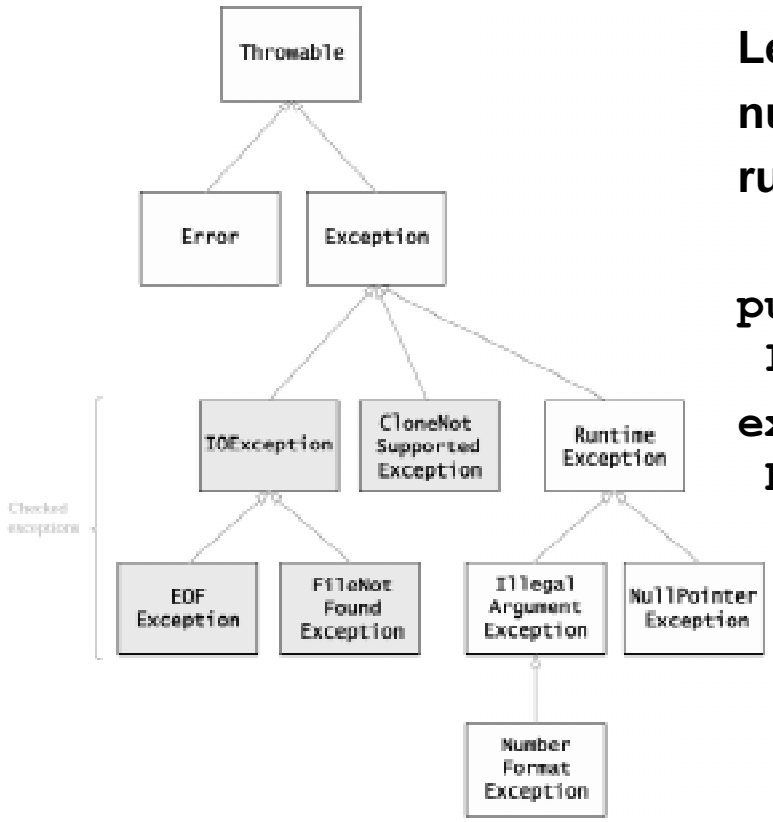
```
class DoubleList implements IList{ // nomi metodi
protected int size;
protected DListNode first, last;
public DoubleList()// costruttore lista vuota

public void insert(Object o)
public void append(Object o)
public Object removeFirst() throws ListEmptyException
public Object removeLast() throws ListEmptyException
public Object first() throws ListEmptyException
public Object last () throws ListEmptyException
public int size()       public boolean isEmpty()
}
```

Eccezioni per le Liste

```
public class ListEmptyException
                extends RuntimeException
{  public String reason;
// costruttore
 ListEmptyException(String reason)
  {this.reason = reason;}

 public String toString() {return reason;}
}
```

L'eccezione segnala una lista vuota per operazioni di analisi e di estrazione

Eccezioni per le Liste



Le list introducono una nuova eccezione a run-time

```
public class
 ListEmptyException
extends
 RuntimeException
```

Lista Doppia: crea e insert

```
class DoubleList implements IList{ // nomi metodi
protected int size;  protected DListNode first, last;
public DoubleList() { size = 0; first = last = null; }
// Inserimento all'inizio della lista
public void insert(Object o)
{ DListNode node = new DListNode(o);
  node.setNext(first); node.setPrev(null);
  if(first!=null) first.setPrev(node);
  first = node;
  if(last == null) last = node;
  size++;
}
}
```


Lista Doppia: append

```
class DoubleList implements IList{ // nomi metodi
protected int size;  protected DListNode first, last;
// Inserimento alla fine della lista
public void append(Object o)
{ DListNode node = new DListNode(o);
  node.setNext(last); node.setPrev(null);
  if(last!=null) last.setNext(node);
    last = node;
    if(first==null) first = node;
    size++;
}
}
```

Lista Doppia: remove inizio

```
// Estrazione all'inizio della lista
public void removeFirst(Object o)
                throws ListEmptyException
{if(isEmpty()) throw new ListEmptyException
                      ("lista vuota!");
  DListNode node = first;
 first = (DListNode) node.getNext();
 if(first!=null)  first.setPrev(null);
   else last = null;
 size--;
 return node.getItem();
}
```

getNext restituisce un ListNode: necessario il cast

Lista Doppia: remove fine

```
// Estrazione alla fine della lista
public void removeLast(Object o)
                throws ListEmptyException
{if(isEmpty()) throw new ListEmptyException
                      ("lista vuota!");
 DListNode node = last;
 last = (DListNode) node.getPrev();
 if(last!=null)  last.setNext(null);
 else first = null;
 size--;
 return node.getItem();
}
```

Lista Doppia: Accessori

```
// Primo della lista
public Object first()throws ListEmptyException
{if(isEmpty()) throw new ListEmptyException
                    ("lista vuota!");
 return first.getItem(); }
public Object last() throws ListEmptyException
{if(isEmpty()) throw new ListEmptyException
      ("lista vuota!");
 return last.getItem();}
public int size(){ return size;}
public boolean isEmpty(){return first==null;}
```


LISTE COME CONTENITORI

- **Approccio imperativo: una lista è una sequenza di contenitori di elementi, di lunghezza non stabilita a priori**

Conseguenze

- **Una lista è un contenitore**
- **Si può *cambiare* una lista esistente**
- **È un approccio efficiente (non si ricostruisce inutilmente ciò che già esiste)**
- **Ma può essere pericoloso in caso di *structure sharing* e *aliasing* (che quindi vanno evitati)**

LISTE COME VALORI

- **Approccio funzionale: una lista è una coppia formata da un elemento e da un'altra lista**
- **il valore *listavuota* è dato a priori**

Conseguenze

- **Una lista non è un contenitore**
- **Non si può *cambiare* una lista esistente: ogni cambiamento produce una lista nuova, e si possono occupare molte risorse**
- **Approccio sicuro (non si cambia mai niente)**
- **NON Approccio efficiente**

JAVA: LE CLASSI DI UTILITÀ

Il package `java.util` definisce:

- ***interfacce* che rappresentano *tipi di dati astratti di contenitori (+ Classi Astratte)***
  - **`Collection` → `List`**
  - **`Collection` → `Set` → `SortedSet`**
  - **`Iterator` → `ListIterator`**
- ***classi* che forniscono *contenitori concreti***
  - **`LinkedList`, `Vector`, `HashSet`, `TreeSet`, ...**

I metodi principali (`add`, `remove`, `contains`, `isEmpty`, ...) sono introdotti da `Collection` e specializzati dalle varie classi

JAVA: LE LISTE PREDEFINITE

- **L'interfaccia `java.util.List` aggiunge ai metodi introdotti da `Collection` altri metodi, specifici del concetto di lista**
- **La classe `java.util.Vector` implementa `List` definendo un componente che cattura l'idea di *array illimitatamente espandibile***
- **La classe `java.util.LinkedList` implementa `List` definendo l'astrazione *lista doppiamente concatenata***

Si veda la documentazione di Java per i dettagli

Iteratori per le Liste

```
public interface IIterator
{ // i metodi posizionano l'iteratore
public Object first(); // sul primo elemento
public Object next(); //  sul prossimo
  // se siamo sull'ultimo elemento, o null o eccezione
public Object current(); // elemento corrente

public boolean isDone();  // scandito tutta la lista?
public boolean hasMore(); // ci sono altri elementi?
}
```

Un iteratore consente di scorrere la lista facilmente: è creato per una scansione fino alla fine e permette di posizionarsi sulla lista e di variarla

Iteratori per le Liste

Con un iteratore, si fanno operazioni ripetute su una lista

```
...
dsList.append(new String ("pippo"));
dsList.insert(new String ("pluto"));
dsList.insert(new String ("paperino"));
SequenceIterator sIter =
                 new SequenceIterator (dsList);
sIter.first();
while (sIter.hasMore())
{System.out.println("elem. " +sIter. current());
 sIter.next();
}
...
```


INTERFACCIA ISequence

```
public interface ISequence //interfaccia sequenza
{
// accedere all'elemento con posizione
public Object get(IPosition pos);
// inserire elemento in una posizione
public void insert(Object o, IPosition pos);
// rimuovere elemento di posizione
public Object remove(IPosition pos);
public IPosition firstPos();
public IPosition lastPos();
public IPosition after(IPosition pos);
public IPosition before(IPosition pos);
}
```

Accesso da posizione qualunque

Sequenza costruita con lista doppia

```
class DLSequence extends DoubleList
                         implements ISequence
{public DLSequence(){ super();}

// accesso all'elemento di posizione data
public Object get(IPosition pos)
{ // controllare che posizione sia nella lista
  return pos.getItem(); }

// inserimento e rimozione dalla lista
public void insert(Object o, IPosition pos)...
public Object remove(IPosition pos)
// altri metodi della Interfaccia
```


Sequenza costruita con lista doppia

```
public void insert(Object o, IPosition pos)
{DListNode node = new DListNode(o);
 DListNode prec; // nodo che precede il nuovo nodo
  if(pos!=null)
      prec =(DListNode)((DListNode)pos).getPrev();
      else prec = last;
  node.setNext((ListNode)pos);node.setPrev(prec);
  if(pos!=null)((DListNode)pos).setPrev(node);
      else last = node;
  if (pos==first)first=node;
     else prec.setNext(node);
  size++;
}
```

Sequenza costruita con lista doppia

```
public Object remove(IPosition pos) {
DListNode succ; DListNode prec;
if(((DListNode)pos) == null) return null;
 // if(pos != null)
if(((DListNode)pos).getPrev() != null &&
     ((DListNode)pos).getNext() != null){
prec =(DListNode)((DListNode) pos).getPrev();
succ =(DListNode)((DListNode) pos).getNext();
prec.setNext(((DListNode) pos).getNext());
prec.setPrev(((DListNode) pos).getPrev());
}
 else // se c'è da sistemare first e/o last
```


Sequenza costruita con lista doppia

```
public Object remove(IPosition pos)
{// sistema first e/o last della lista
...
else {
if (((DListNode)pos).getPrev() == null)
{first
 =(DListNode)((DListNode)pos).getNext();}
if (((DListNode)pos).getNext() == null)
{last =(DListNode)((DListNode)pos).getPrev();}
}
size--;
return pos.getItem();
}
```

Sequenza costruita con lista doppia

```
class DLSequence extends DoubleList
                    implements ISequence

public IPosition firstPos(){ return first; }
public IPosition lastPos() { return last;}

public IPosition after(IPosition pos)
   { return ((DListNode) pos).getNext();}

public IPosition before(IPosition pos)
   { return ((DListNode) pos).getPrev();}
}
```