Capitolo 17

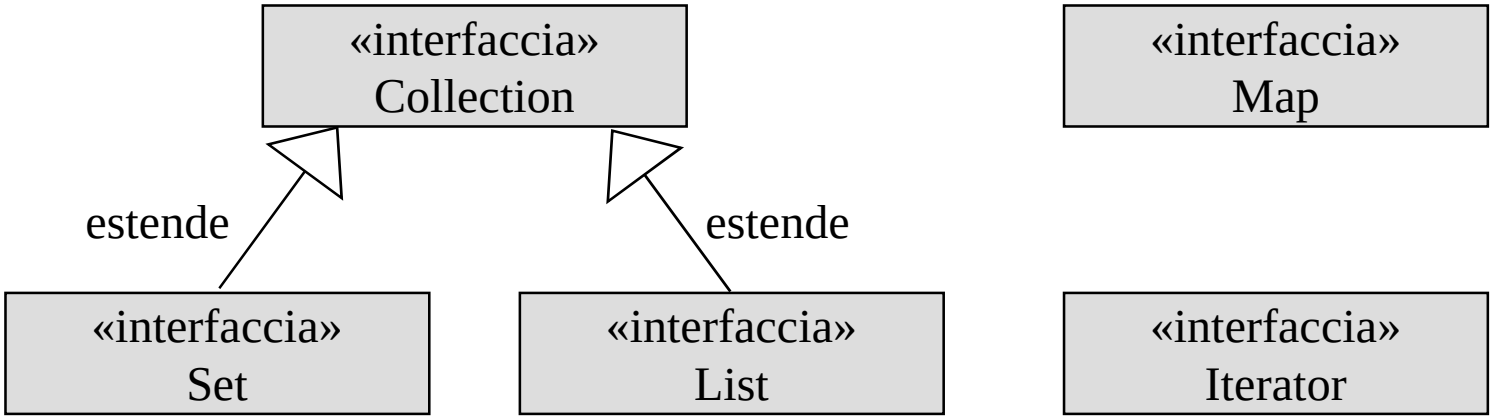
Collezioni

Animated Version

Introduzione al Java Collections Framework

- Una **collezione** (o **contenitore**) consente di organizzare e gestire un gruppo di oggetti
  - collezioni (vere e proprie)
  - mappe
  - implementate dal **Java Collections Framework** (**JFC**)
    - un insieme di interfacce e classi
    - package **java.util**

Le principali interfacce del JCF



- **Collection** rappresenta una generica collezione
- **List** rappresenta una lista
- **Set** rappresenta un insieme
- **Map** rappresenta una mappa
- **Iterator** supporta la visita dei contenitori

Collezioni

- Una **collezione**
  - un gruppo di oggetti
  - interfaccia **Collection**
  - nessuna implementazione diretta

Operazioni sulle collezioni

- Operazioni principali
  - **boolean add(Object o)**
  - **boolean contains(Object o)**
  - **boolean remove(Object o)**
  - **int size()**
  - **boolean isEmpty()**
  - **Iterator iterator()**

Insiemi

- Un **insieme**
  - una collezione di elementi – senza duplicati
  - interfaccia **Set** che estende **Collection**
  - **HashSet** e **TreeSet**

Operazioni sugli insiemi

- Operazioni principali
  - **boolean add(Object o)**
  - **boolean contains(Object o)**
  - **boolean remove(Object o)**
  - **int size()**
  - **boolean isEmpty()**
  - **Iterator iterator()**

Esempio – gestione di un insieme di stringhe


```
import java.util.*;
...
   Set s;    // un insieme di stringhe

   /* crea un nuovo insieme s */
   s = new HashSet();

   /* inserisce alcune stringhe nell'insieme s */
   s.add("uno");
   s.add("due");
   s.add("tre");

   /* accede all'insieme */
   System.out.println( s.size() );              // 3
   System.out.println( s.contains("uno") );     // true
   System.out.println( s.contains("sei") );     // false
   System.out.println( s.toString() );  // [tre, uno, due]
```

Esempio – gestione di un insieme di stringhe


```
/* modifica l'insieme s */
s.add("uno");    // s non cambia
System.out.println( s.size() );              // 3

s.remove("due");
System.out.println( s.size() );              // 2
System.out.println( s.toString() );  // [tre, uno]

s.add("alfa");
s.add("beta");
System.out.println( s.size() );              // 4
```

Iteratori

- Un **iteratore** è un oggetto di supporto usato per accedere gli elementi di una collezione, uno alla volta e in sequenza

- Operazioni del tipo **Iterator**
  - **boolean hasNext()**
  - **Object next()**
  - **void remove()**

Uso di un iteratore


```
/* Visualizza sullo schermo gli elementi dell'insieme s. */
public static void visualizza(Set s) {
    // pre: s!=null
    Iterator i;    // per visitare gli elementi di s
    Object o;      // un elemento di s

    /* visita e visualizza gli elementi di s */
    i = s.iterator();
    while ( i.hasNext() ) {
        o = i.next();
        System.out.println(o.toString());
    }
}
```

Visita di collezioni di oggetti omogenei


```
/* Calcola la stringa più lunga dell'insieme s di stringhe. */
public static String piùLunga(Set s) {
    // pre: s!=null && s.size()>0 &&
    //      gli elementi di s sono tutte stringhe
    Iterator i;      // per visitare gli elementi di s
    String t;        // un elemento di s
    String lunga;    // la stringa più lunga di s

    /* visita s cercando la stringa più lunga */
    i = s.iterator();
    lunga = (String) i.next();
    while ( i.hasNext() ) {
        t = (String) i.next();
        if (t.length()>lunga.length())
            lunga = t;
    }
    return lunga;
}
```

Liste

- Una **lista**
  - una sequenza di elementi
  - interfaccia **List** che estende **Collection**
  - è possibile l'accesso posizionale
  - **ArrayList** e **LinkedList**

Operazioni sulle liste

- Operazioni principali
  - **boolean add(Object o)**
  - **boolean contains(Object o)**
  - **boolean remove(Object o)**
  - **int size()**
  - **boolean isEmpty()**
  - **Iterator iterator()**

Operazioni di accesso posizionale

- Operazioni principali per l'accesso posizionale
  - **Object get(int i)**
  - **Object set(int i, Object o)**
  - **Object remove(int i)**
  - **void add(int i, Object o)**
  - **int indexOf(Object o)**

Esempio – gestione di una lista di stringhe


```
List l;    // una lista di stringhe

 /* crea una nuova lista l */
 l = new LinkedList();

 /* inserisce alcune stringhe nella lista l */
 l.add("due");
 l.add("quattro");
 l.add("sei");

 /* accede alla lista */
 System.out.println( l.size() );    // 3
 System.out.println( l.get(0) );    // due
 System.out.println( l.toString() );
     // [due, quattro, sei]
```

Esempio – gestione di una lista di stringhe


```
/* modifica la lista l */
l.add(1, "tre");    // tra "due" e "quattro"
System.out.println( l.toString() );
    // [due, tre, quattro, sei]

l.add(0, "uno");    // inserimento in testa
System.out.println( l.toString() );
    // [uno, due, tre, quattro, sei]

l.remove(4);        // cancella "sei"
System.out.println( l.size() );             // 4
System.out.println( l.toString() );
    // [uno, due, tre, quattro]
```

Legge una sequenza di interi e la scrive al contrario


```
List l;         // una sequenza di numeri interi
Integer nn;     // un elemento di l
int n;          // numero letto dalla tastiera
int i;          // per scandire gli elementi di l

/* legge e memorizza la sequenza di numeri */
l = new ArrayList();
while (scanner.hasNextInt()) {
    n = scanner.nextInt();
    nn = new Integer(n);
    l.add(nn);
}
/* scandisce la lista al contrario */
for (i=l.size()-1; i>=0; i--) {
    nn = (Integer) l.get(i);
    n = nn.intValue();
    System.out.print(n + " ");
}
System.out.println();
```

Mappe

- Una **mappa**
  - un insieme di coppie (chiave, valore) – che non contiene chiavi duplicate
  - interfaccia **Map**
  - **HashMap** e **TreeMap**

Operazioni sulle mappe

- Operazioni principali
  - **Object put(Object k, Object v)**
  - **Object get(Object k)**
  - **Object remove(Object k)**
  - **int size()**
  - **boolean isEmpty()**
  - **Set keySet()**

Esempio – dizionario italiano-inglese


```
Map m;    // un insieme di coppie
          // (parola italiana, traduzione inglese)

/* crea una nuova mappa m */
m = new HashMap();
/* inserisce alcune coppie nella mappa m */
m.put("uno", "one");
m.put("due", "two");
m.put("tre", "tree");    // oops...

/* accede alla mappa */
System.out.println( m.size() );         // 3
System.out.println( m.get("uno") );     // one
System.out.println( m.get("sei") );     // null
System.out.println( m.toString() );
    // {tre=tree, uno=one, due=two}

/* modifica la mappa m */
m.put("tre", "three");    // così è meglio
System.out.println( m.size() );         // ancora 3
System.out.println( m.get("tre") );     // three
```

Problemi delle collezioni

- Dal momento che ogni collezione contiene oggetti di tipo **Object**
  - Non è possibile recuperare le informazioni sul tipo una volta inserito un oggetto in un contenitore
  - Questo è dovuto alla generalità delle **Collection**
- Vantaggi:
  - Le **Collection** funzionano con tutti gli oggetti
- Svantaggi:
  - Non esistono limitazioni sul tipo di oggetto che può essere inserito in una collezione
  - Il tipo dell'oggetto restituito da **List::get(int i)** e **Iterator::next()** è **Object**
  - Per utilizzare propriamente gli oggetti è necessario effettuare un cast esplicito (problemi?)

Talvolta funziona comunque…


```
/* Le cose funzionano senza cast: magia? */
public static void main(String[] args) {
    List studenti=new ArrayList();

    /* inserisce 3 elementi nella lista */
    for(int i=0; i<3; i++)
        studenti.add(new Studente("xyz", i));
    for(int i=0; i< studenti.size(); i++)
    /* non è necessario alcun cast
     * chiamata automatica a toString() */
        System.out.println("Studente " + i + ": " +
                studenti.get(i));
}
```

Creazione di collezioni omogenee

- Una soluzione per la creazione di collezioni che accettano un solo tipo è possibile.

```
import java.util.*;
public class StudentsList {
     private ArrayList list = new ArrayList();
     public void add(Studente s) {
            list.add(s);
     }
     public Studente get(int index) {
            return (Studente)list.get(index);
     }
     public int size() { return list.size(); }
}
```

Utilizzo della classe **StudentsList**


```
public class StudentsListTest {
   public static void main(String[] args) {
        StudentsList list = new StudentsList ();

        for(int i=0; i<3; i++)
               studenti.add(new Studente("xyz", i));
        for(int i=0; i<studenti.size(); i++) {
        /* non è necessario alcun cast */
               Studente s=studenti.get(i));
               ... // altre elaborazioni
        }
}
```

- … ma è necessario replicare il codice per ciascuna collezione!

Soluzione: i generics (da Java 1.5)

- I generics permettono di specificare, a tempo di compilazione, il tipo degli oggetti che verranno inseriti nella collezione.
  - In questo modo, quando si inseriscono e recuperano gli elementi dalla collezione, questa sa già di che tipo sono tali elementi ed il cast esplicito non è più necessario
  - In caso di inserimento di un elemento del tipo errato, l'errore viene segnalato in fase di **compilazione** (cfr. con il caso precedente)
- Esempio:

```
import java.util.*;
public class First {
   public static void main(String args[]) {
        List<Integer> myList = new ArrayList<Integer>(10);
        myList.add(10);  // OK ???
        myList.add("Hello, World"); // OK ???
   }
}
```

Esempio (errore in fase di compilazione)


```
import java.util.*;
public class First {
   public static void main(String args[]) {
        List<Integer> myList = new ArrayList<Integer>(10);
        myList.add(10);        // automatic conversion from
                               // the int type to an Integer
        myList.add("Hello, World");
   }
}

First.java:7: cannot find symbol symbol :
   method add(java.lang.String)
   location: interface java.util.List<java.lang.Integer>
   myList.add("Hello, World");
              ^ 1 error
```

Risultato

- Ora possiamo creare tutte le collezioni **type-safe** che vogliamo
- Problema:
  - Abbiamo spostato semplicemente il cast dall'estrazione alla dichiarazione?
- NO! Ora abbiamo anche il vantaggio del controllo sul tipo a tempo di compilazione (e non più a run-time come in precedenza)
- Vantaggio:
  - Le nostre applicazioni sono più sicure (è più difficile ottenere delle run-time exceptions)

Un altro vantaggio: un **for** più semplice!

- L'uso degli iteratori continua a valere

```
Set<Integer> s;
Iterator<Integer> i;       // per visitare gli elementi di s
Integer o;          // un elemento di s

/* visita e visualizza gli elementi di s */
i = s.iterator();
while ( i.hasNext() ) {
     o = i.next();
     System.out.println(o.toString());
}
```

- Ma si può usare anche un **for** più semplice (senza iteratore)!

```
for (Integer i: s)
     System.out.println(i.toString());
```

… e se continuiamo ad utilizzare le collezioni come prima?


```
import java.util.*;
public class Second {
     public static void main(String args[]) {
            List list = new ArrayList();
            list.add(10);
     }
}
```

- Compilando il programma otteniamo un warning:
  - Note: Second.java uses unchecked or unsafe operations.
  - Note: Recompile with -Xlint:unchecked for details.
- Per risolvere:

```
import java.util.*;
public class Second {
     public static void main(String args[]) {
            List<Object> list = new ArrayList();
            list.add(10);
     }
}
```

Perché funziona?

- Tutto è sottoclasse di **Object**, quindi tutto può andare in **list**
  - Questo può essere utile per creare collezioni che contengano oggetti di classi e sottoclassi diverse
- Esempio:

```
public abstract class Shape {
     public abstract void draw();
}
public class Circle extends Shape {
     private int x, y, radius;
     public void draw() { ... }
}
public class Rectangle extends Shape {
     private int x, y, width, height;
     public void draw() { ... }
}
public class Third {
     public static void main(String args[]) {
            List<Shape> shapes;
            for (Shape s: shapes) { s.draw(this);
     }
```

Tutto qui?

- Ovviamente no!
  - Esistono molte altre implicazioni, ma richiedono una conoscenza del linguaggio più profonda
  - Ad esempio:
    - uso di generics come tipi di ritorno di metodi
    - sub-typing di generics
    - identificazione a run-time della classe,
    - ecc.
- Una precisazione importante:
  - I generics sono diversi dai template C++
  - In particolare, non c'è duplicazione di codice (la classe **List** è unica, indipendentemente dalle realizzazioni)

Differenze tra ArrayList e LinkedList

- **ArrayList** è:
  - Veloce nell'accesso agli elementi (tramite **get(int i)**)
  - Lento nelle operazioni di inserimento e rimozione
- **LinkedList** è:
  - Lento nell'accesso agli elementi (tramite **get(int i)**)
  - Veloce nelle operazioni di inserimento e rimozione
  - Dispone, inoltre, dei metodi:
    - **void addFirst(Object o)**: aggiunge in prima posizione
    - **void addLast(Object o)**: aggiunge in ultima posizione
    - **Object getFirst()**: restituisce l'oggetto in prima posizione
    - **Object getLast()**: restituisce l'oggetto in ultima posizione
    - **Object removeFirst()**: rimuove l'oggetto in prima posizione
    - **Object removeLast()**: rimuove l'oggetto in ultima posizione

Implementazione con array

- La rappresentazione più “banale” utilizza un vettore in cui sono inseriti gli elementi della lista in maniera sequenziale (rappresentazione sequenziale).
- Per motivi di efficienza, si mantiene una variabile che indica da quanti elementi è composta la lista.

```
public class MyArrayList {
    private Object[] array;
    private int size;
    ... // altri metodi
}
```

Creazione di una ArrayList

- Il primo problema da affrontare è:
  - Quanti elementi deve contenere la mia lista?
  - Tale numero può essere noto a priori?
- Il problema viene risolto ridimensionando (ad es., raddoppiando) l'array ogni qualvolta questo non sia in grado di ospitare un nuovo oggetto, e copiando gli elementi nel nuovo array ridimensionato.

```
public MyArrayList() {
     array=new Object[1];
     size=0;
}
private void expand() {
     newarray=new Object[array.length*2];
     for(int i=0; i<size; i++) newarray[i]=array[i];
     array=newarray;
}
```

Metodi accessori

- Lunghezza della lista:

```
public int size() {
     return size;
}
```

- Verifica se la lista è vuota:

```
public boolean isEmpty() {
     return (size==0);
}
```

- Problema:
  - Se la lista non è piena esistono delle celle dell'array vuote
  - La memoria, quindi, non è utilizzata al meglio

Accesso agli elementi

- Accesso in posizione qualunque:

```
public Object get(int index) {
     return array[index];
}
```

- Modifica in posizione qualunque:

```
public Object set(int index, Object o) {
     Object old=array[index];
     array[index]=o;
     return old;
}
```

Aggiunta di un elemento

- Aggiunta in coda (con eventuale espansione):

```
public boolean add(Object o) {
     if(size==array.length) expand();
     array[size]=o;
     size++;
     return true;
}
```

- Aggiunta in posizione qualunque (con eventuale espansione):

```
public void add(int index, Object o) {
     if(size==array.length) expand();
     for(int i=size; i>index; i--) array[i]=array[i-1];
     array[index]=o;
     size++;
}
```

Rimozione di un elemento

- Rimozione in posizione qualunque:

```
public Object remove(int index) {
     Object o=array[index];
     for(int i=index+1; i<size; i++)
            array[i-1]=array[i];
     size--;
     return o;
}
```

- Rimozione di un oggetto:

```
public boolean remove(Object o) {
     for(int i=0; i<size; i++)
            if(array[i].equals(o)) {
                   remove(i);
                   return true;
            }
     return false;
}
```

Ricerca di un elemento

- Ricerca della posizione:

```
public int indexOf(Object o) {
     for(int i=0; i<size; i++)
            if(array[i].equals(o))
                   return i;
     return -1;
}
```

- Ricerca di un oggetto:

```
public boolean contains(Object o) {
     return (indexOf(o)>=0);
}
```

Iteratore per la classe MyArrayList


```
class MyArrayListIterator implements Iterator {
     private MyArrayList list;
     private int index;

     public MyArrayListIterator(MyArrayList l) {
            list=l;
            index=0;
     }

     public boolean hasNext() { return(index<list.size()); }
     public Object next() { return list.get(index++); }
     public void remove() { list.remove(--index); }
}
```

- Per creare l'iteratore (metodo di MyArrayList)

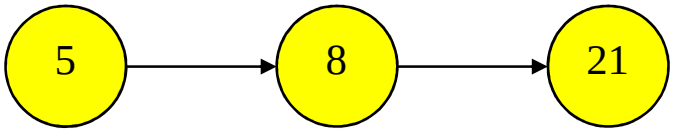
```
public Iterator iterator() {
     return new MyArrayListIterator(this);
}
```

Complessità delle varie operazioni

- Supponendo che esistano *N* elementi nella lista:
  - Creazione: costante, *O*(1)
  - Espansione: copia di tutti gli elementi della lista, *O*(*N*)
  - Calcolo della lunghezza: costante, *O*(1)
  - Verifica se la lista è vuota: costante, *O*(1)
  - Accesso/modifica in posizione qualunque: costante, *O*(1)
  - Aggiunta in coda (senza espansione): costante, *O*(1)
  - Aggiunta in posizione qualunque (senza espansione): shift di elementi, *O*(*N* – i)
  - Rimozione in posizione qualunque: shift di elementi, *O*(*N* – i)
  - Rimozione di un oggetto: ricerca + shift di elementi, *O*(*N*)
  - Ricerca di un elemento: caso peggiore, *O*(*N*)
- In particolare:
  - Aggiunta in testa: *O*(*N*)
  - Rimozione in testa: *O*(*N*)

Implementazione con lista concatenata

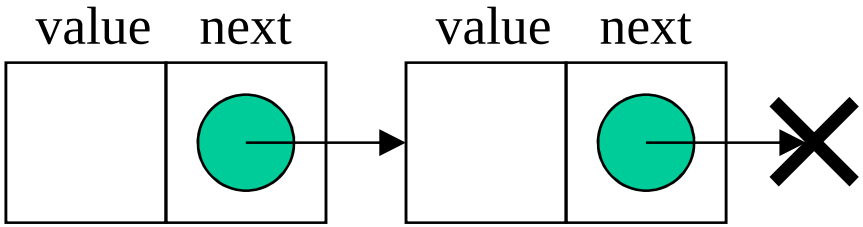
- Si memorizzano gli elementi in variabili dinamiche distinte (nello ***heap***) associando ad ognuno di essi il riferimento alla variabile dinamica in cui è memorizzato l'elemento successivo.
- Esempio:
  - Elementi della lista come nodi (allocati dinamicamente)
  - Riferimenti come archi



- Vantaggio:
  - La memoria utilizzata viene gestita dinamicamente
  - L'occupazione è esattamente proporzionale al numero degli elementi della lista (utilizzo ottimale della memoria)

Rappresentazione

- Ciascun elemento della lista è un record con due campi:
  - un campo rappresenta il valore dell'elemento;
  - un campo "punta" all'elemento successivo nella lista (null, nel caso dell'ultimo elemento).



Implementazione


```
class ListNode {
     Object value;
     ListNode next;

     // costruttore
     public ListNode(Object o, ListNode l) {
            value=o;
            next=l;
     }
     // metodi accessori
     public Object getValue() { return value; }
     public ListNode getNext() { return next; }
     // metodi modificatori
     public void setValue(Object o) { value=o; }
     public void setNext(ListNode l) { next=l; }
}
```

- La definizione è ricorsiva, in quanto uno dei campi del record è un riferimento ad un record dello stesso tipo

Rappresentazione della lista

- La lista altro non contiene che un riferimento al primo elemento della lista (la “testa”)

```
public class MyLinkedList {
     private ListNode head, tail;
     private int size;
     ... // altri metodi
}
```

- Per questioni di efficienza nell’implementazione di alcune operazioni si mantengono anche:
  - il riferimento all’ultimo elemento della lista (la “coda”)
  - il numero di elementi effettivamente presenti nella lista

Costruttore e metodi accessori

- Costruttore:

```
public MyLinkedList() {
     head=tail=null;
     size=0;
}
```

- Lunghezza della lista:

```
public int size() {
     return size;
}
```

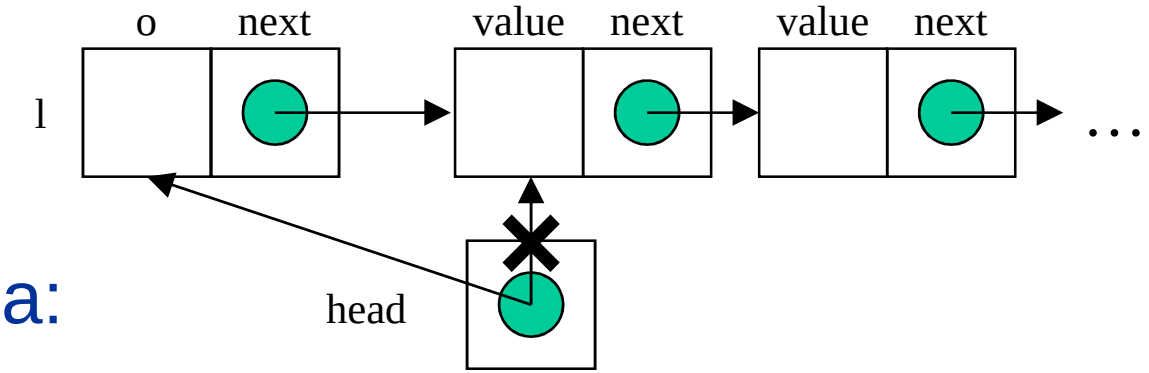
- Verifica se la lista è vuota:

```
public boolean isEmpty() {
     return (size==0);
}
```

Aggiunta di un elemento

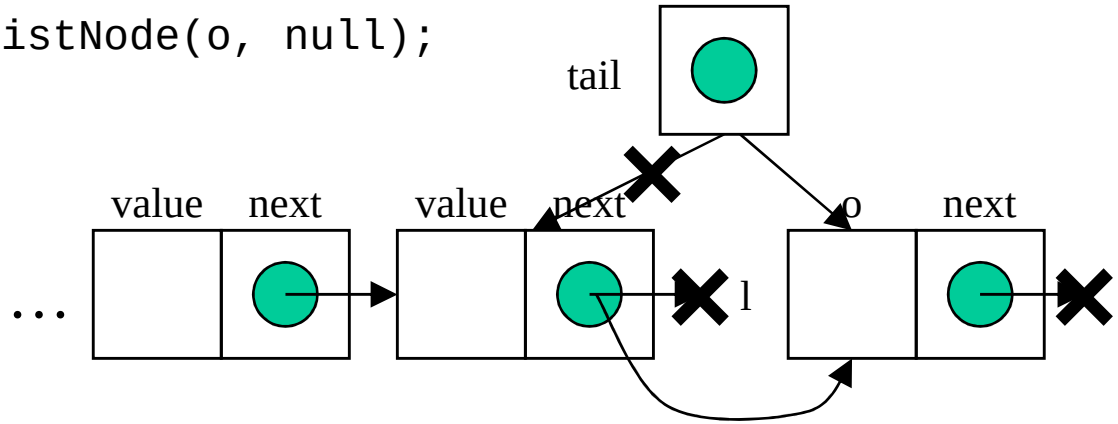
- Aggiunta in testa:

```
public void addFirst(Object o) {
     ListNode l=new ListNode(o, head);

     head=l;
     size++;
}
```



- Aggiunta in coda:

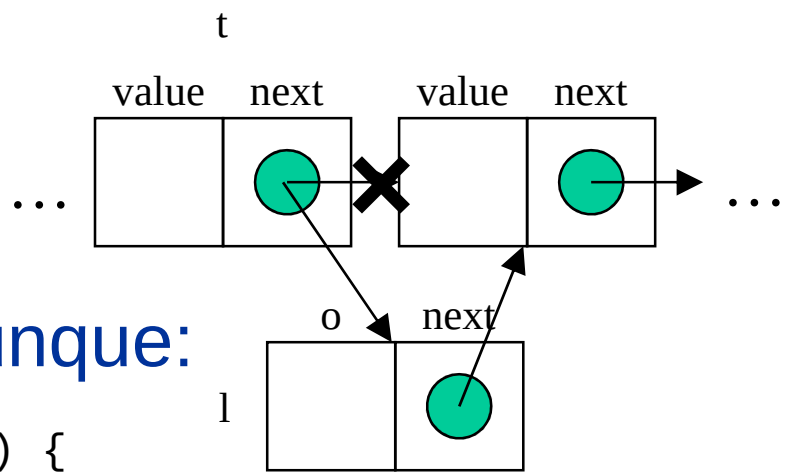
```
public void addLast(Object o) {
     ListNode l=new ListNode(o, null);

     tail.setNext(l);
     tail=l;
     size++;
}
```



Aggiunta di un elemento

- Aggiunta (in coda):

```
public boolean add(Object o) {
     addLast(o);
     return true;
}
```

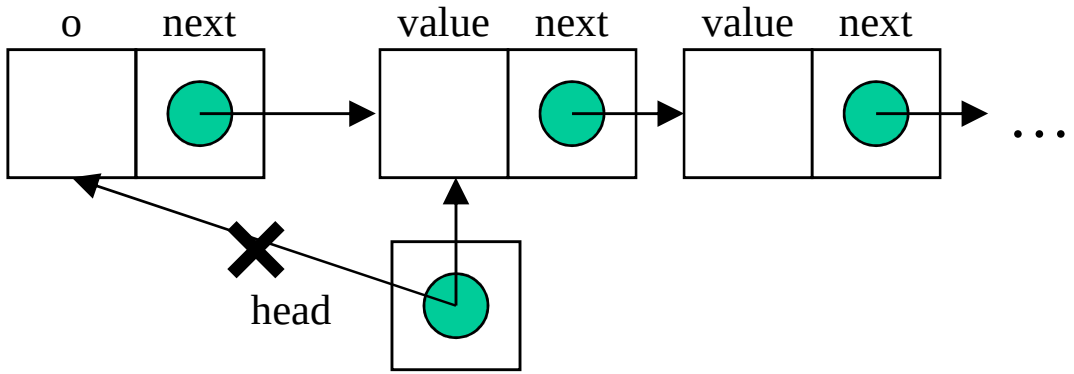


- Aggiunta in posizione qualunque:

```
public void add(int index, Object o) {
     ListNode l, t=head;
     if(index==0) return addFirst();
     for(int i=0; i<index; i++) t=t.getNext();
     l=new ListNode(o, t.getNext());
     t.setNext(l);
     size++;
     if(tail==t) tail=l; // aggiorna la coda
}
```

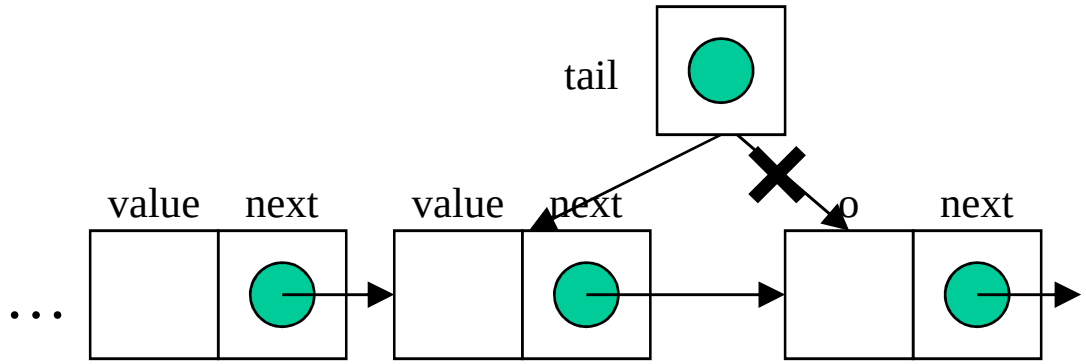
Rimozione in testa

```
public Object removeFirst() {
    Object o=head.getValue();

    head=head.getNext();
    size--;
    if(size==0) tail=null;
    return o;
}
```



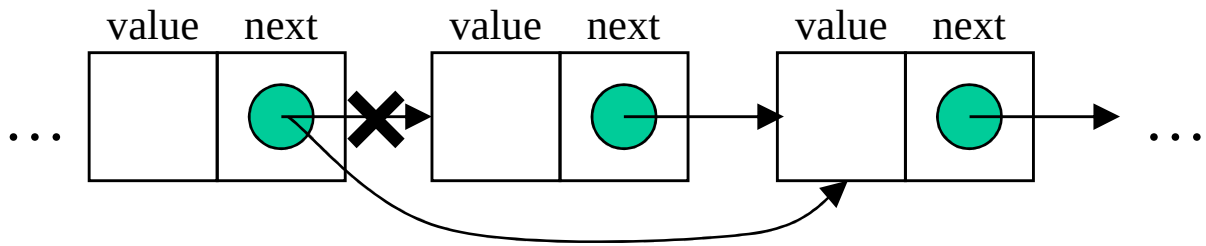
Rimozione in coda

```
public Object removeLast() {
    if(size==1) return removeFirst();
    Object o=tail.getValue();
    ListNode l=head;

    while(l.getNext()!=tail)
        l=l.getNext();
    tail=l;
    size--;
    return o;
}
```



Rimozione di un elemento in posizione qualunque


```
public Object remove(int index) {
    ListNode t=head, next;

    if(index==0) return removeFirst();
    for(int i=0; i<index-1; i++) t=t.getNext();
    next=t.getNext();
    Object o=next.getValue();

    t.setNext(next.getNext());
    size--;
    if(tail==next) tail=t; // aggiorna la coda
    return o;
}
```



Ricerca di un elemento

- Ricerca della posizione:

```
public int indexOf(Object o) {
     ListNode t=head;
     for(int i=0; i<size; i++)
            if(t.getValue().equals(o))
                   return i;
            else t=t.getNext();
     return -1;
}
```

- Ricerca di un oggetto:

```
public boolean contains(Object o) {
     return (indexOf(o)>=0);
}
```

Rimozione di un oggetto

- Rimozione di un oggetto:

```
public boolean remove(Object o) {
     ListNode t=head;

     for(int i=0; i<size; i++)
            if(t.getValue().equals(o)) { // cancella l'elemento
                   ListNode next=t.getNext();

                   t.setNext(next.getNext());
                   size--;
                   if(tail=next) tail=t; // aggiorna la coda
                   return true;
            }
            else t=t.getNext();
     return false;
}
```

Accesso agli elementi

- Accesso in testa:

```
public Object getFirst() { return head.getValue(); }
```

- Accesso in coda:

```
public Object getLast() { return tail.getValue(); }
```

- Accesso in posizione qualunque:

```
public Object get(int index) {
     ListNode l=head;
     for(int i=0; i<index; i++) l=l.getNext();
     return l.getValue();
}
```

- Modifica in posizione qualunque:

```
public Object set(int index, Object o) {
     ListNode l=head;
     for(int i=0; i<index; i++) l=l.getNext();
     Object old=l.getValue();
     l.setValue(o);
     return old;
}
```

Iteratore per la classe MyLinkedList


```
class MyLinkedListIterator implements Iterator {
     private MyLinkedList list;
     private ListNode node;

     public MyLinkedListIterator(MyLinkedList l) {
            list=l;
            node=l.head;
     }
     public boolean hasNext() { return(node==null); }
     public Object next() {
            Object o=node.getValue();
            node=node.getNext();
            return o; }
     public void remove() { ... }
}
```

- Per creare l'iteratore (metodo di MyLinkedList)

```
public Iterator iterator() {
     return new MyLinkedListIterator(this);
}
```

Complessità delle varie operazioni

- Supponendo che esistano *N* elementi nella lista:
  - Creazione: costante, *O*(1)
  - Calcolo della lunghezza: costante, *O*(1)
  - Verifica se la lista è vuota: costante, *O*(1)
  - Accesso/modifica in posizione qualunque: scansione, *O*(i)
  - Aggiunta in testa/coda: costante, *O*(1)
  - Aggiunta in posizione qualunque: scansione, *O*(i)
  - Rimozione in testa: costante, *O*(1)
  - Rimozione in coda: scansione, *O*(*N*)
  - Rimozione in posizione qualunque: scansione, *O*(i)
  - Ricerca di un elemento: caso peggiore, *O*(*N*)
  - Rimozione di un oggetto: ricerca, *O*(*N*)

L'ADT pila

- Una pila (stack) è un tipo particolare di lista in cui le operazioni possono avvenire solamente in testa (o solo in coda).
  - è utile per modellare situazioni in cui si memorizzano e si estraggono elementi secondo la logica "Last In, First Out" (LIFO)
  - un esempio di questa situazione è la gestione dei record di attivazione delle funzioni
  - le operazioni di inserimento e rimozione sono chiamate, rispettivamente, "push" e "pop"

- Operazioni sulle pile
  - **void push(Object o)**: aggiunge un oggetto in testa (coda)
  - **Object pop()**: rimuove l'oggetto in testa (coda)
  - **Object top()**: restituisce l'oggetto in testa (coda)

Creare una pila con **LinkedList**


```
public class StackL {
  private LinkedList list;
  public StackL() { list=new LinkedList(); }
  public void push(Object o) { list.addFirst(o); }
  public Object top() { return list.getFirst(); }
  public Object pop() { return list.removeFirst(); }

  public static void main(String[] args) {
    StackL stack = new StackL();
    for(int i = 0; i < 10; i++)
      stack.push(String.valueOf(i));
    System.out.println(stack.top());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.top());
  }
}
```

Creare una pila con **ArrayList**

- Inserire/rimuovere gli elementi in testa sarebbe troppo costoso.
- Di conseguenza, si inseriscono/rimuove gli elementi in coda.

```
public class StackA {
     private ArrayList list;

     public StackA() { list=new ArrayList(); }
     public void push(Object o) { list.add(o); }
     public Object top() { return list.get(list.size()-1); }
     public Object pop() { return list.remove(list.size()-1); }
}
```

Complessità delle operazioni su pile

- Implementazione con **LinkedList**:
  - Inserimento: complessità di addFirst() = $O(1)$
  - Rimozione: complessità di removeFirst() = $O(1)$
- Implementazione con **ArrayList**:
  - Inserimento: complessità di add() = $O(1)$
  - Rimozione: complessità di remove(i) = $O(N - i)$ = $O(N - (N - 1)) = O(1)$

L'ADT coda

- Una coda (queue) è un tipo particolare di lista in cui le operazioni possono avvenire solamente in posizioni predeterminate.
- In particolare:
  - l'inserimento avviene ad una sola estremità (ad es., in coda)
  - il prelievo avviene all'estremità opposta (ad es., dalla testa).
  - è utile per modellare situazioni in cui si memorizzano e si estraggono elementi secondo la logica "First In, First Out" (FIFO)
  - le operazioni di inserimento e rimozione sono chiamate, rispettivamente, "put" e "get" (oppure "qush" e "qoq")

- Operazioni sulle code
  - **void put(Object o)**: aggiunge un oggetto in coda (testa)
  - **Object get()**: rimuove l'oggetto in testa (coda)
  - **Object top()**: restituisce l'oggetto in testa (coda)

Creare una coda con **LinkedList**


```
public class QueueL {
   private LinkedList list;
   public QueueL() { list=new LinkedList(); }
   public void put(Object o) { list.addLast(o); }
   public Object top() { return list.getFirst(); }
   public Object get() { return list.removeFirst(); }

   public static void main(String[] args) {
        QueueL queue = new QueueL();
        for(int i = 0; i < 10; i++)
               stack.push(String.valueOf(i));
        System.out.println(stack.top());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.top());
   }
}
```

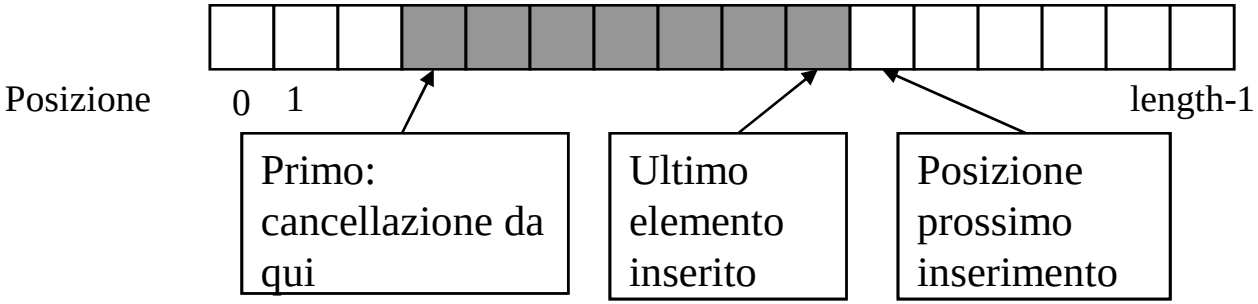
Creare una coda con **ArrayList**

- Si può implementare la coda come caso particolare di lista.
  - In questo caso, però, la cancellazione dalla testa è particolarmente costosa.

- Si può migliorare la complessità delle operazioni su coda osservando il comportamento del vettore:
  - Gli inserimenti avvengono sempre spostando la posizione di inserimento verso destra.
  - Le cancellazioni avvengono sempre sul primo elemento (quindi, logicamente, spostando verso destra la posizione di cancellazione).

Creare una coda con **ArrayList**

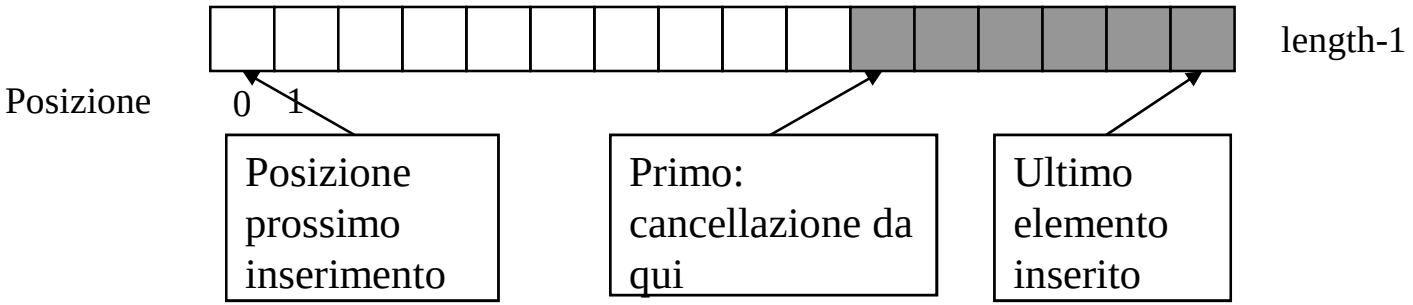
- Se supponiamo di non effettuare lo scorrimento dei dati in cancellazione, si ottiene un pacchetto di dati validi che, per effetto di inserimenti e cancellazioni, si mantiene consecutivo ma si sposta verso destra.



- Per tenere traccia di questo pacchetto è sufficiente mantenere traccia di due posizioni:
  - la prima (primo elemento da estrarre)
  - l'ultima (punto in cui effettuare il prossimo inserimento)

Implementazione come vettore circolare

- Quando la coda raggiunge l'ultima posizione del vettore è possibile ritornare ad utilizzare il primo elemento del vettore (gestione circolare della struttura), in modo che la posizione successiva a length – 1 sia la posizione 0.



- Per far ciò è sufficiente considerare la posizione come:

posizione % length

Quando la lista è piena?

- Utilizzando la tecnica circolare, quando la lista è piena la posizione di inserimento e quella di cancellazione coincidono.
- Il problema è che anche quando la lista è vuota la posizione di inserimento e quella di cancellazione coincidono.
- Per risolvere l'ambiguità si può considerare la proprietà **size()**.

```
...
if(head==tail)&&(size()>0) expand();
...
```

Implementazione


```
public class QueueA {
    private ArrayList list;
    private int head, tail;
    private int size;
    public QueueA() { list=new ArrayList(); head=tail=size=0; }
    public void put(Object o) {
          if(head==tail)&&(size>0) {
                    ArrayList l=new ArrayList();
                    while(!list.isEmpty()) l.add(list.remove(head));
                    head=0;
                    tail=list.length;
                    list=l;
          }
          list.set(tail, o);
          tail=(tail+1)%list.length;
          size++;
    }
    public Object top() { return list.get(head); }
    public Object get() {
          Object o=list.get(head);
          head=(head+1)%list.length;
          size--;
          return o;
    }
}
```

Complessità delle operazioni su code

- Implementazione con **LinkedList**:
  - Inserimento: complessità di addLast() = *O*(1)
  - Rimozione: complessità di removeFirst() = *O*(1)
- Implementazione con **ArrayList**:
  - Inserimento: complessità di set() = *O*(1)
  - Rimozione: complessità di get(i) = *O*(1)