Programmazione II

A.A. 2022-23
Prof. Maria Tortorella

Strutture dati lineari

- LIFO
- FIFO
- Liste

Strutture dati

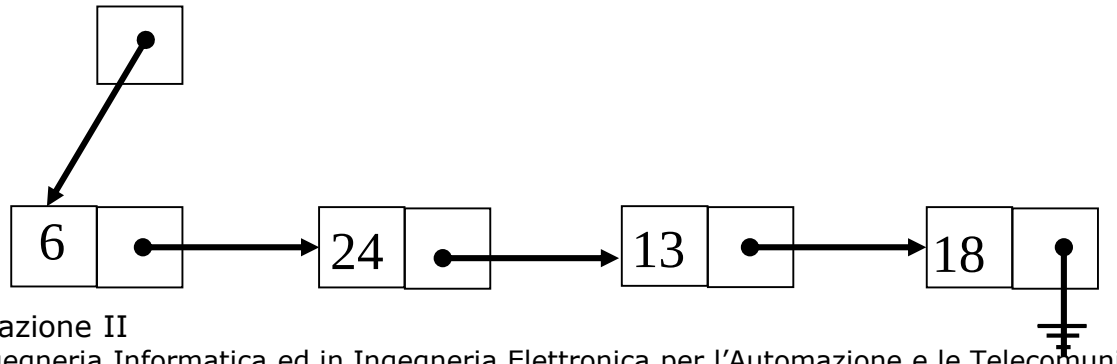
- Un programma object-oriented usa oggetti per memorizzare collezioni di dati
- Fino ad ora abbiamo utilizzato gli ArrayList
- È possibile definire altre strutture dati per organizzare gli oggetti:
  - Strutture dati lineari, alberi, grafi, …

Lista semplice: SimpleList

- Fornisce metodi per l'accesso al primo elemento della lista
  - **`public SimpleList()`**
    - Costruisce la lista vuota
  - **`public Object head()`**
    - Restituisce il primo elemento della lista (definito sulla lista non vuota)
  - **`public void insertHead(Object elem)`**
    - Inserisce elem in testa alla lista
  - **`public void removeHead()`**
    - Elimina il primo elemento della lista (definito sulla lista non vuota)
  - **`public int size()`**
    - Restituisce la lunghezza della lista
  - **`public boolean isEmpty()`**
    - Restituisce true se la lista è vuota, false altrimenti

SimpleList: Realizzazione

- Due alternative
  - Sequenziale
    - Basata su array monodimensionale
    - Bisogna definire una capacità massima e un metodo *full*
    - Realizzarla come esercizio …
  - Collegata
    - Ogni elemento della lista è una coppia denominata Node
    - Node = (Object, Node)



Rappresentazione collegata

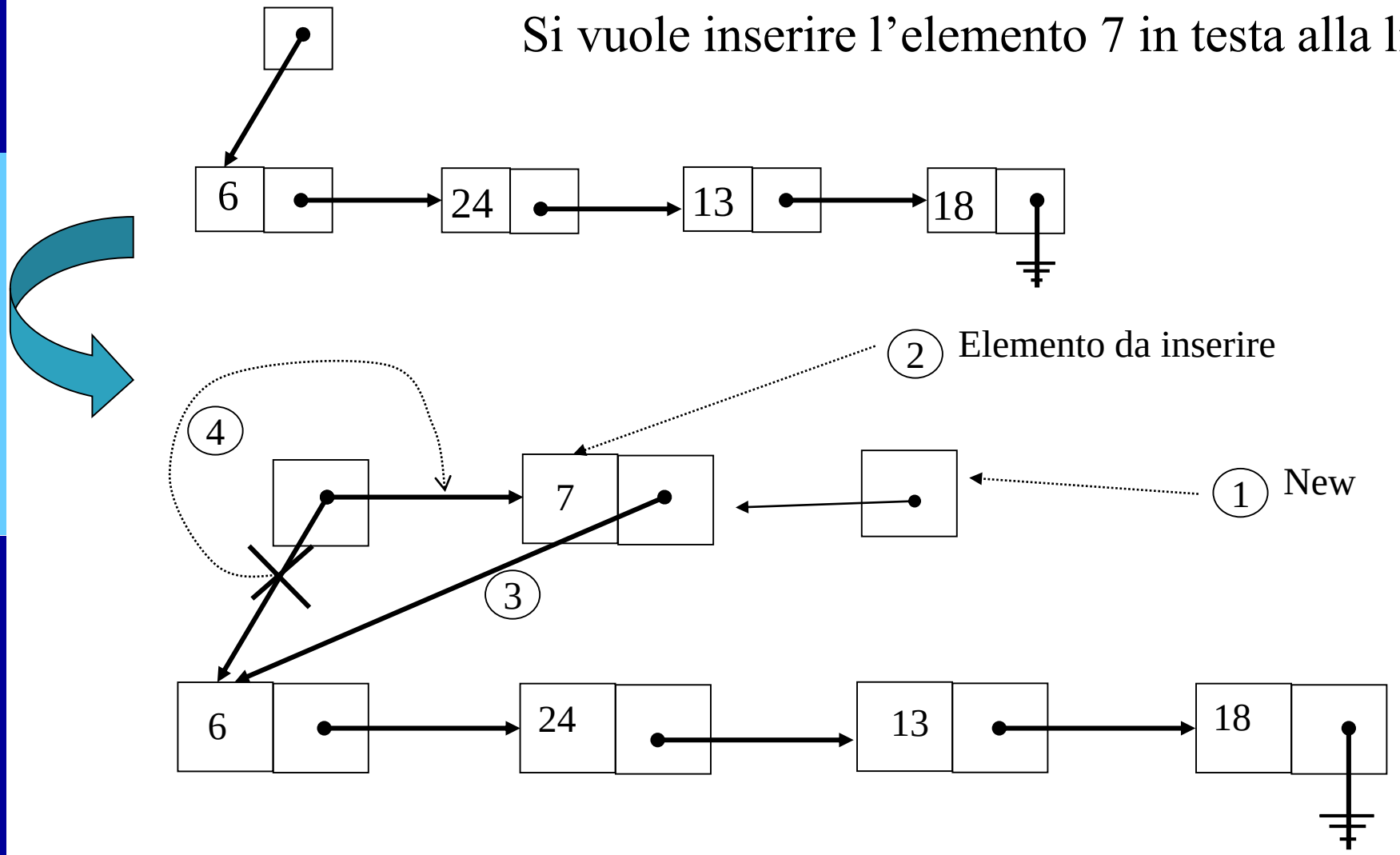
```
import java.io.*;
class SimpleList {
    class Node  {
            public Object elem;
            public Node succ;
    }

   // definizione dei metodi

   private Node node;   // referenza al primo nodo
   private int length;  // mantiene la lunghezza della lista
}
```

SimpleList: metodi

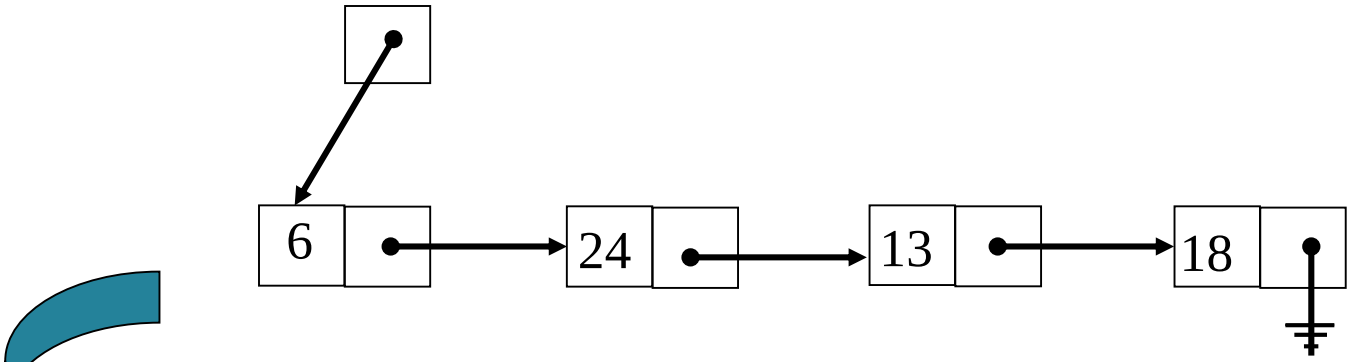
```
public SimpleList() {
      node = null;
      length = 0;
}

public boolean isEmpty() {
  return (node == null);
}

public int size() {
  return length;
}

public Object head() {
      if(node == null)
            return null;
      return node.elem;
}
```

SimpleList: Inserimento

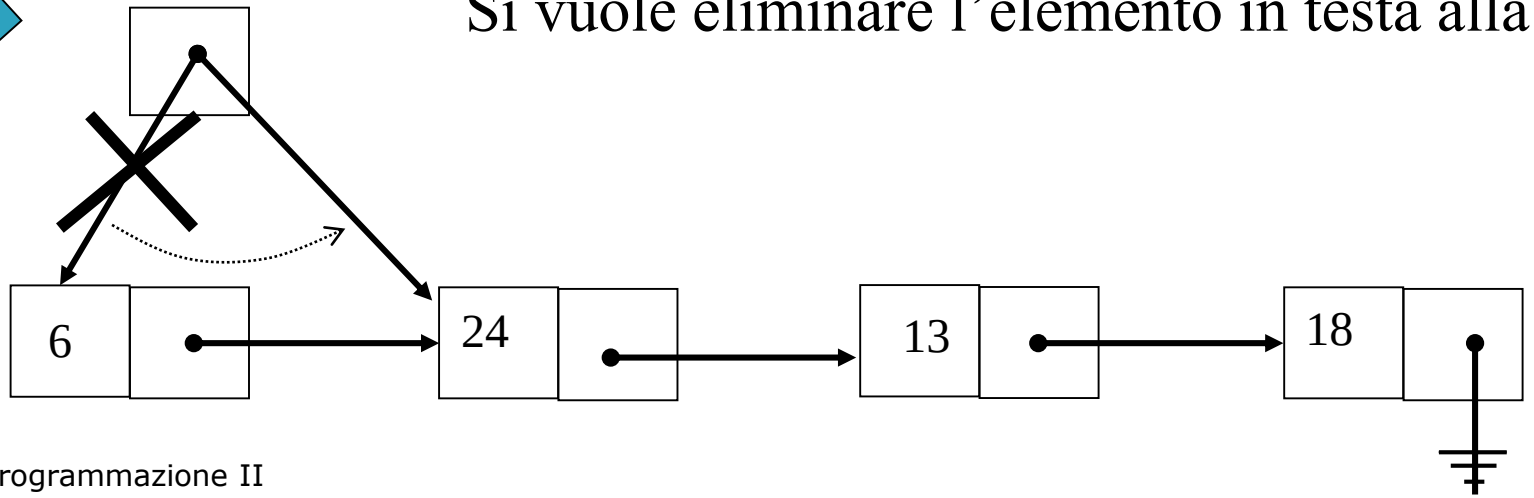
Si vuole inserire l'elemento 7 in testa alla lista



SimpleList: Eliminazione



Si vuole eliminare l'elemento in testa alla lista



SimpleList: metodi

```
public void insertHead(Object elem) {
      Node tempNode = new Node(); // costruttore di default
      tempNode.elem = elem;
      tempNode.succ = node;
      node = tempNode;
      length++;
}
public void removeHead() {
      if(node != null) {
            node = node.succ;
            length--;
       }
}
```

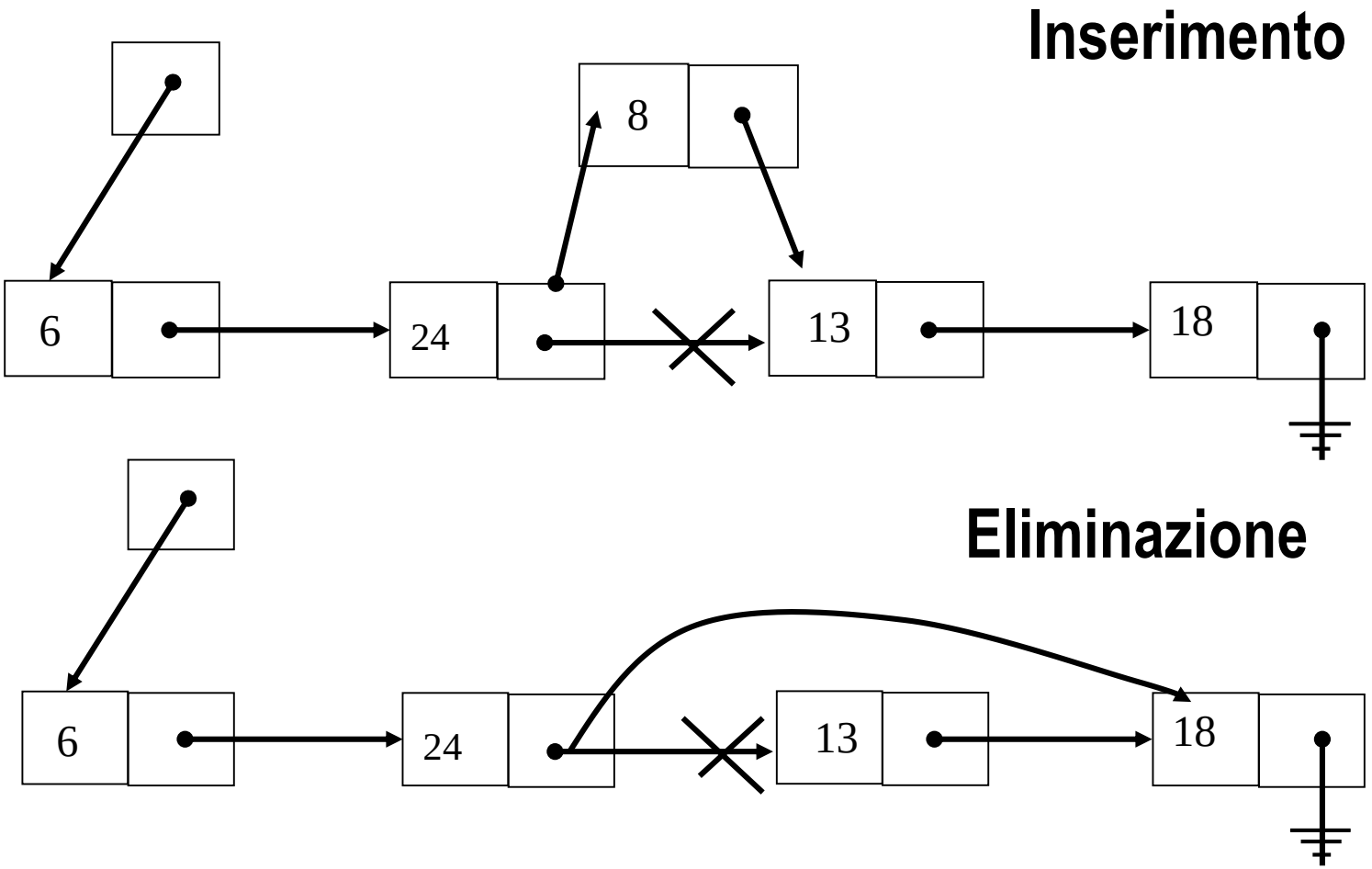
SimpleList: testDriver

```
public static void testDriver() throws Exception {
	Scanner sc = new Scanner(System.in);
	SimpleList list = new SimpleList();
	System.out.println("Stringa: ");
	String line = sc.nextLine();
	while(!line.equals("")) {
		list.insertHead(line);
		System.out.println("Stringa: ");
		String line = sc.nextLine();
	}
	System.out.println("lunghezza lista: " + list.size());
	while(!list.isEmpty()) {
		line = (String) list.head();
		System.out.println(line);
		list.removeHead();
	}
}
```

Liste indicizzate

- Gli operatori della lista semplice costituiscono l'insieme di operatori minimali per la realizzazione di una lista
- Altri operatori possono essere realizzati a partire da questo insieme minimale …
  - … oppure possono estendere questo insieme minimale
- Un'estensione della lista semplice è la lista indicizzata
  - Accesso, inserimento ed eliminazione di elementi possono avvenire in ogni posizione
    - **`elementAt(int pos)`**
    - **`insertElementAt(Object elem, int pos)`**
    - **`removeElementAt(int pos)`**

Inserimento ed eliminazione in posizione qualsiasi



Rappresentazione collegata

```
import java.io.*;
class LinkedList {
    class Node  {
            public Object elem;
            public Node succ;
    }

  // definizione dei metodi

  private Node node;    // referenza al primo nodo
  private int length;   // mantiene la lunghezza della lista
}
```

List: realizzazione di elementAt

```
public Object elementAt(int pos) {
      int i = 0;
      Node tempNode = node;
      while (i < pos && tempNode != null) {
            tempNode = tempNode.succ;
            i++;
       }
      if (i != pos || tempNode == null)
            return null;
      return tempNode.elem;
}
```

List: realizzazione di insertElementAt

```
public void insertElementAt(Object elem, int pos) {
      if(pos == 0)
            insertHead(elem);
      else {
            int i = 1;
            Node predNode = node; // punta al predecessore
            while (i < pos && predNode != null) {
                  predNode = predNode.succ;
                  i++;
             }
            if (i == pos && predNode != null) {
                  Node tempNode = new Node();
                  tempNode.elem = elem;
                  tempNode.succ = predNode.succ;
                  predNode.succ = tempNode;
           // bisogna incrementare la lunghezza della lista
                  length++;
            }
}
```

List: realizzazione di removeElementAt

```
public void removeElementAt(int pos) {
  if(pos == 0){
      removeHead();
  }
  else {
      int i = 1;
      Node predNode = node;   // punta al predecessore
      while (i < pos && predNode != null) {
            predNode = predNode.succ;
            i++;
      }
      if (i == pos && predNode != null
                              && predNode.succ != null) {
            predNode.succ = predNode.succ.succ;
                  // decrementa la lunghezza della lista
            length--;
      }
}
```

Copia di liste e interferenza

- Con l'assegnamento

  l1 = l2;

  le due variabili puntano allo stesso oggetto interferendo tra di loro su successive operazioni di modifica
- Gli oggetti vanno copiati

```
public List copia() {
          List temp = new LinkedList();
          temp.node = this.node;
          temp.length = this.length;
          return temp;
    }
```

- In questo modo posso effettuare l'assegnamento

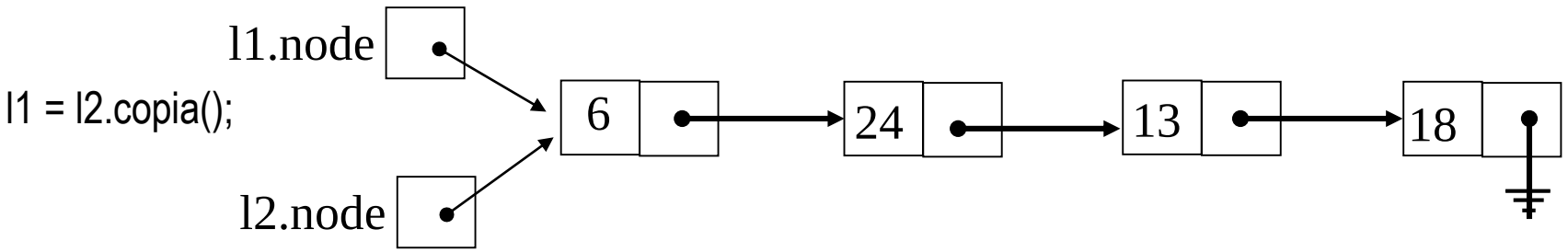
  l1 = l2.copia();

Copia di liste e interferenza

- Ma cosa succede se con l'operatore di copia si eseguono le seguenti operazioni?

```
l1 = l2.copia();
l2.insertElementAt(elem, 4);
l1.removeElementAt(3);
```

- Se la lista di partenza è:

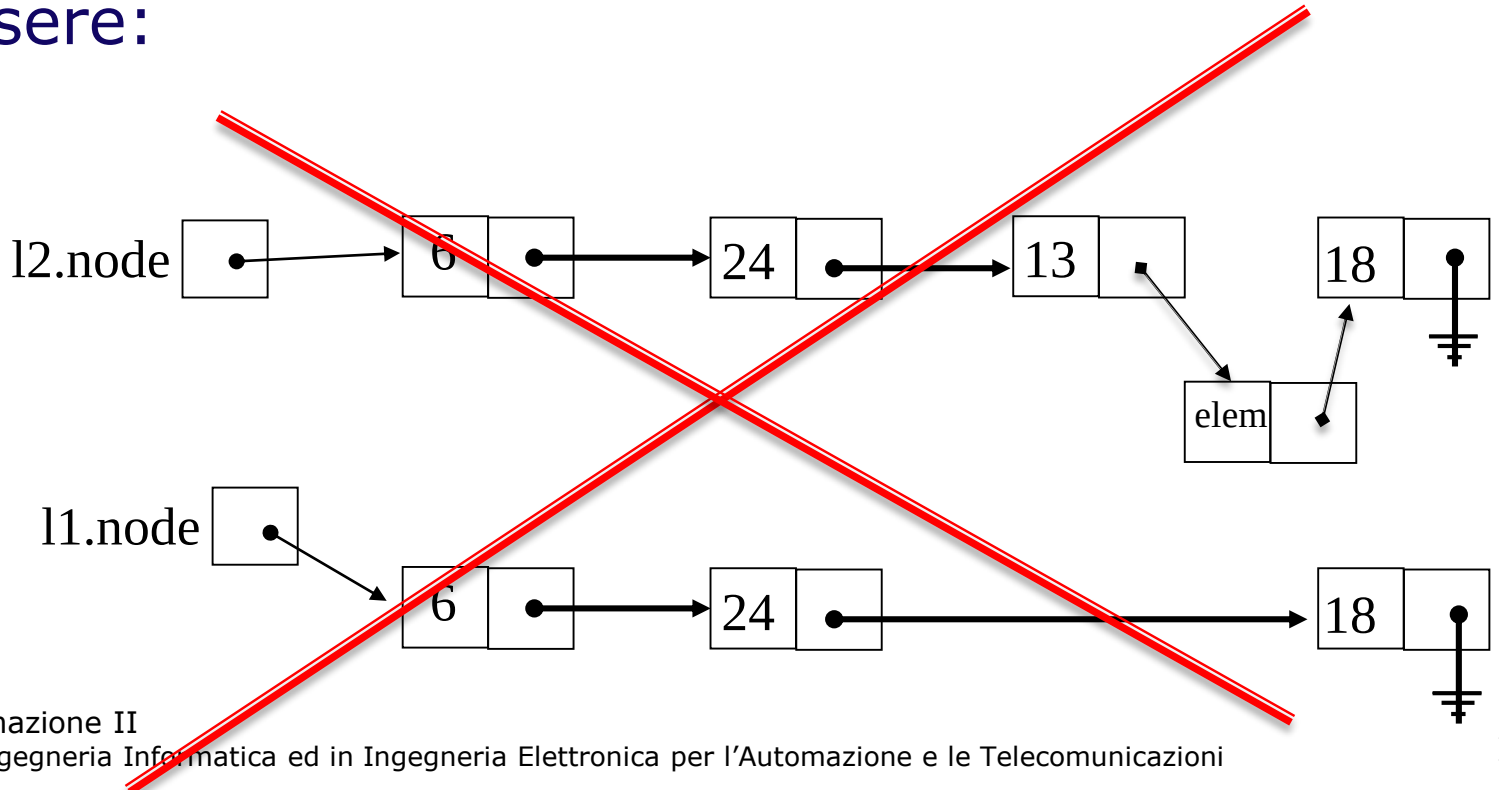


- Il risultato che ci si aspetterebbe …..

Copia di liste e interferenza

```
l1 = l2.copia();
l2.insertElementAt(elem, 4);
l1.removeElementAt(3);
```

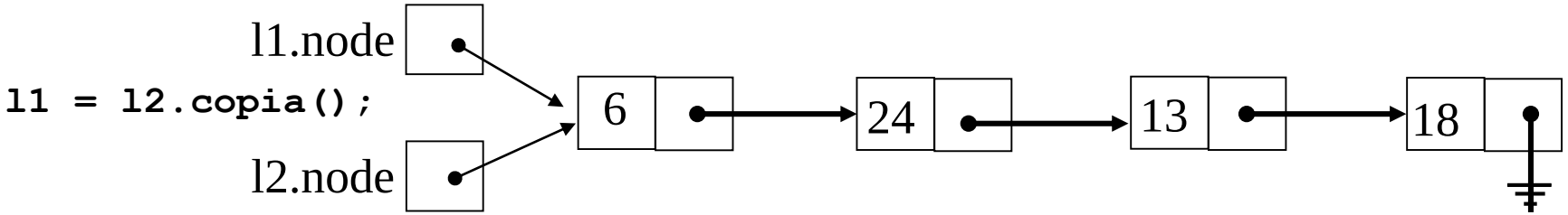
- Il risultato che ci si aspetterebbe dovrebbe essere:

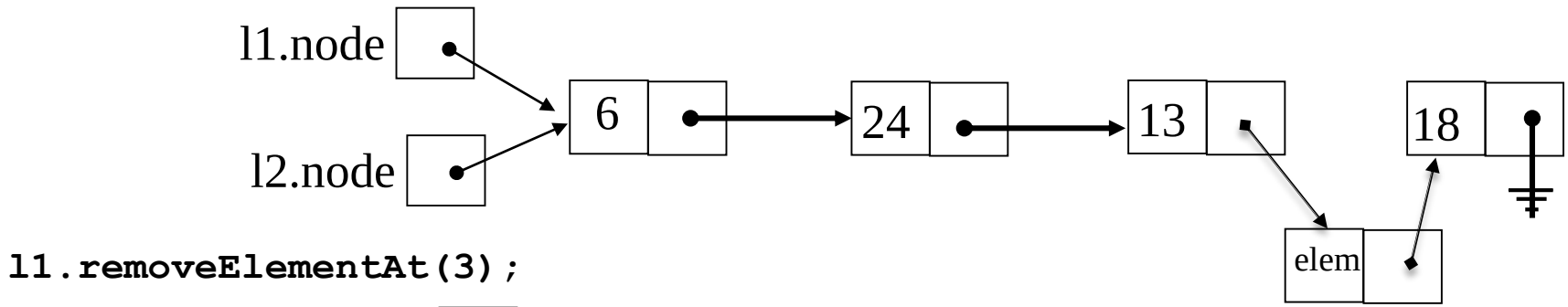


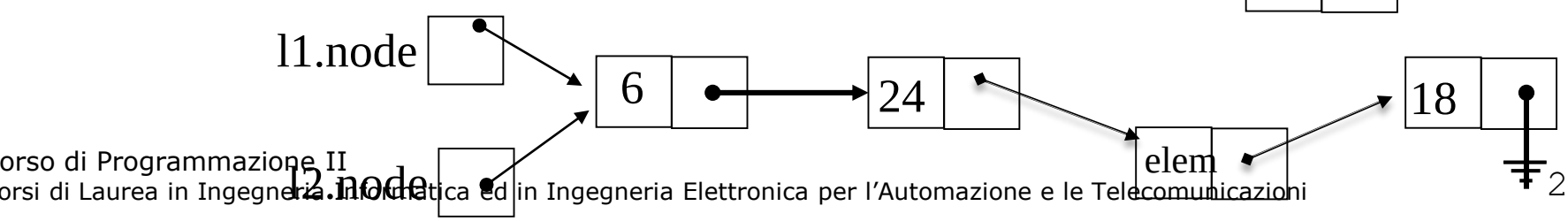
Copia di liste e interferenza

- Ma:

```
l1 = l2.copia();
l2.insertElementAt(elem, 4);
l1.removeElementAt(3);
```







Copia di liste e interferenza

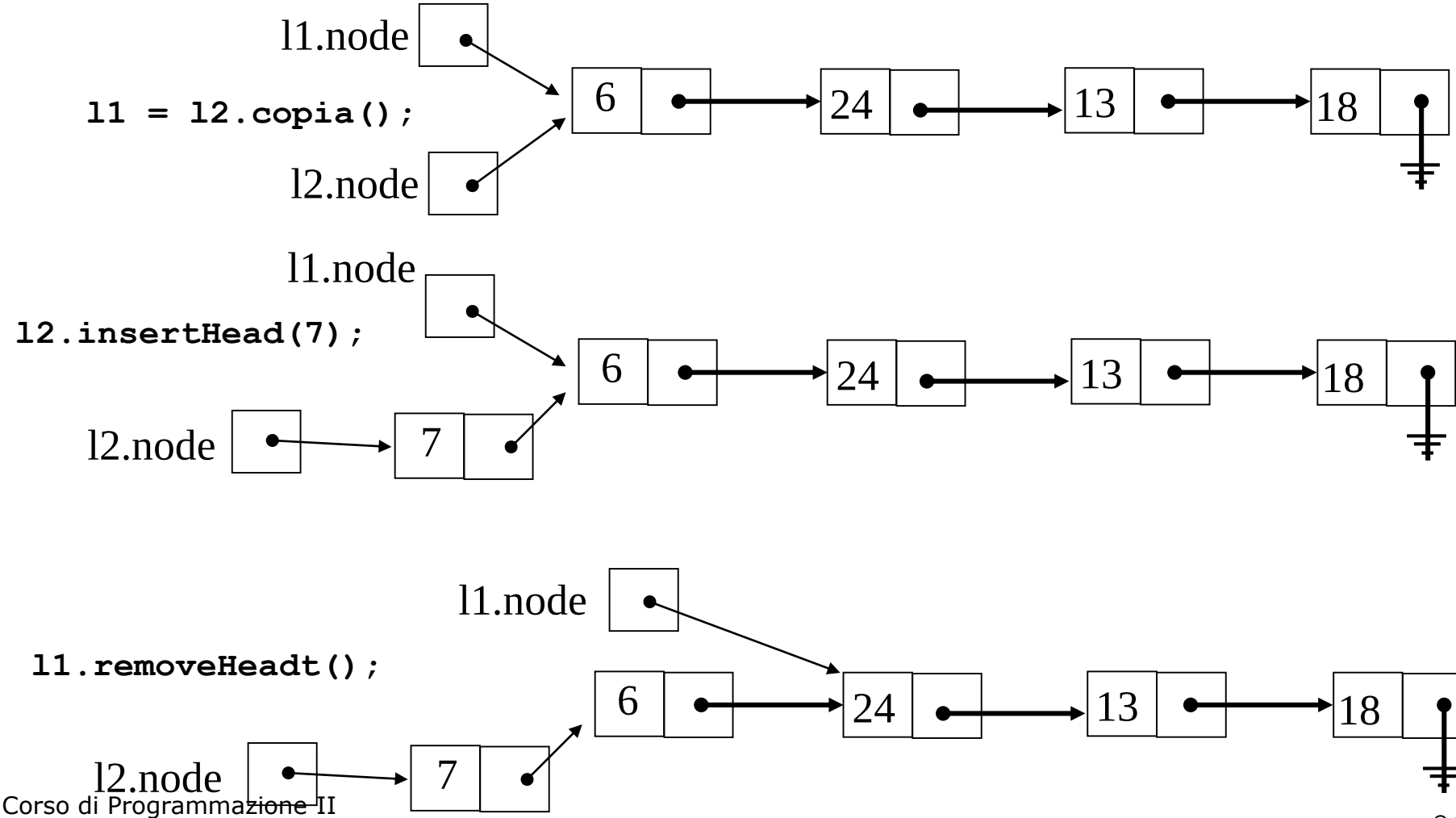
- Le due liste interferiscono l'una con l'altra
  - L'operatore di copia visto prima in questo caso non è sufficiente ad evitare l'interferenza sulle posizioni intermedie della struttura a puntatori
  - Il metodo copia dovrebbe duplicare l'intera struttura (tutti i nodi)

Lista semplice e interferenza

- Da notare che la lista semplice non soffre di questo problema, perché l'accesso è possibile solo in testa:

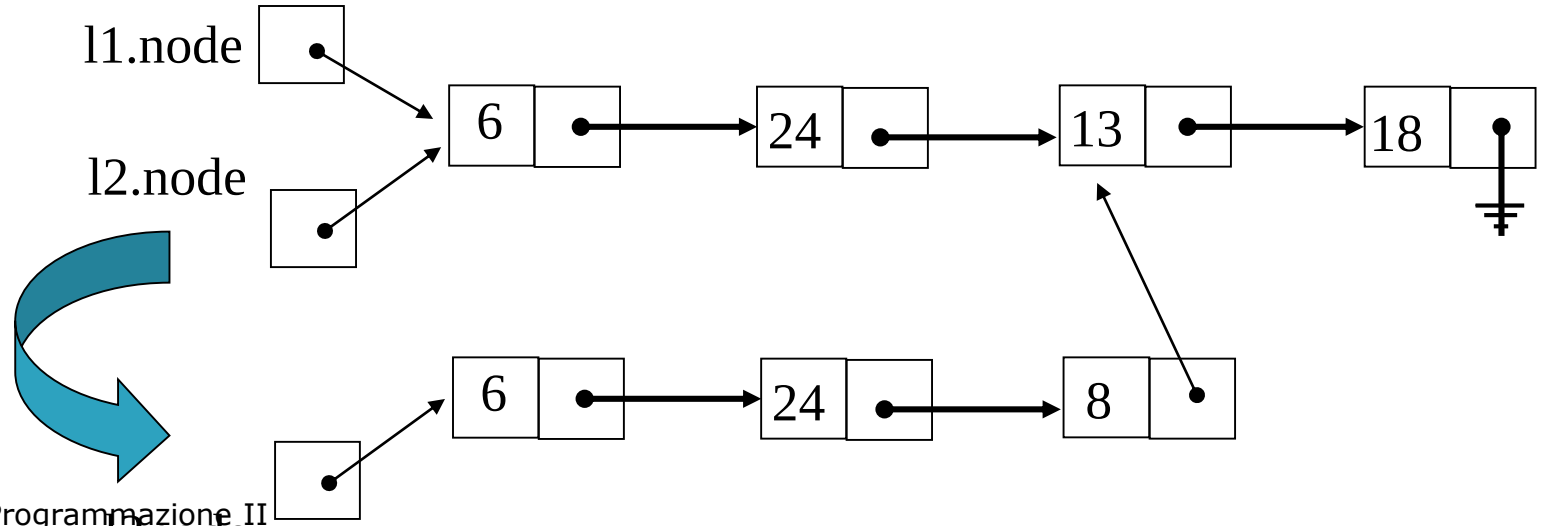
```
l1 = l2.copia();
l2.insertHead(7);
l1.removeHead();
```

La lista semplice è libera da interferenze



Realizzazione senza interferenza

- Per avere una realizzazione libera da interferenza bisogna duplicare la parte precedente alla posizione in cui si vuole operare (inserire/rimuovere)
  - Ad esempio, **`l2.insertElementAt(8, 2)`**

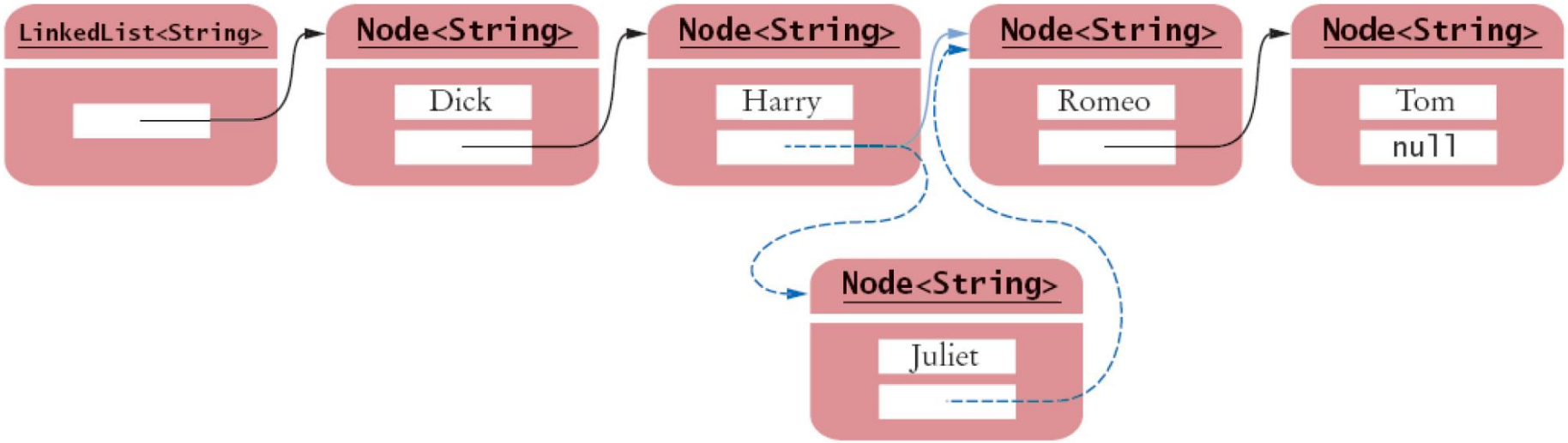


Strutture e libreria standard

- Strutture dati che possono essere utilizzate nei programmi:
  - Linked lists
  - Iterators
  - Stack e Queue
  - Set e map
- Classi ed interface dichiarate nel package java.util

Linked List

- Un insieme di nodi ognuno dei quali ha una referenza al prossimo nodo



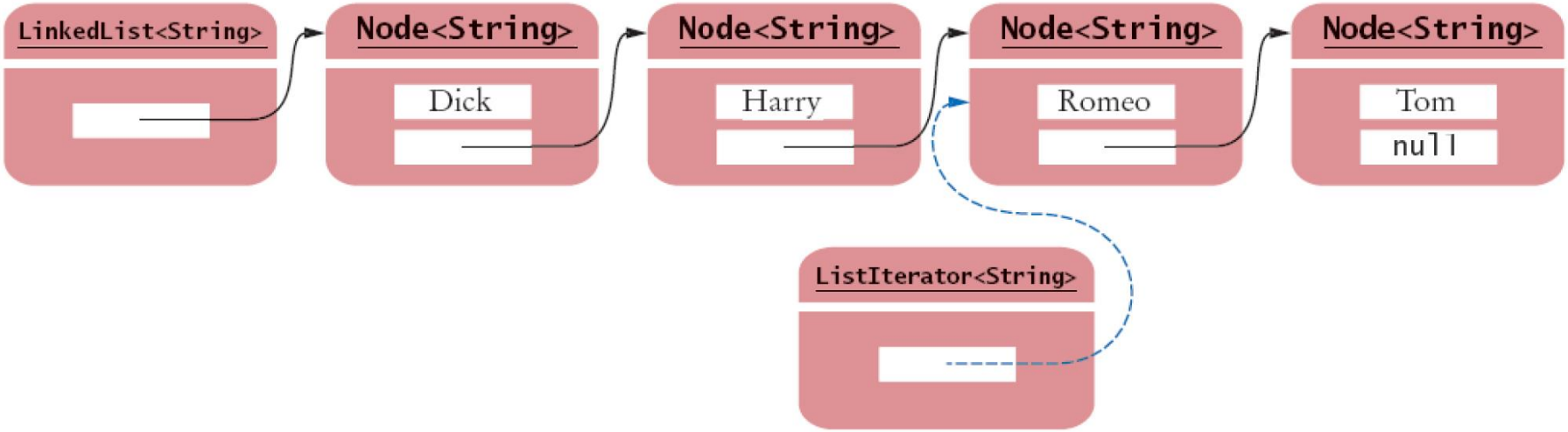
`LinkedList` class di Java

- E' una classe generica
  - E' necessario specificare il tipo degli elementi: **`LinkedList<Product>`**
- Package: **`java.util`**
- Semplicissimo accedere al primo ed all'ultimo elemento

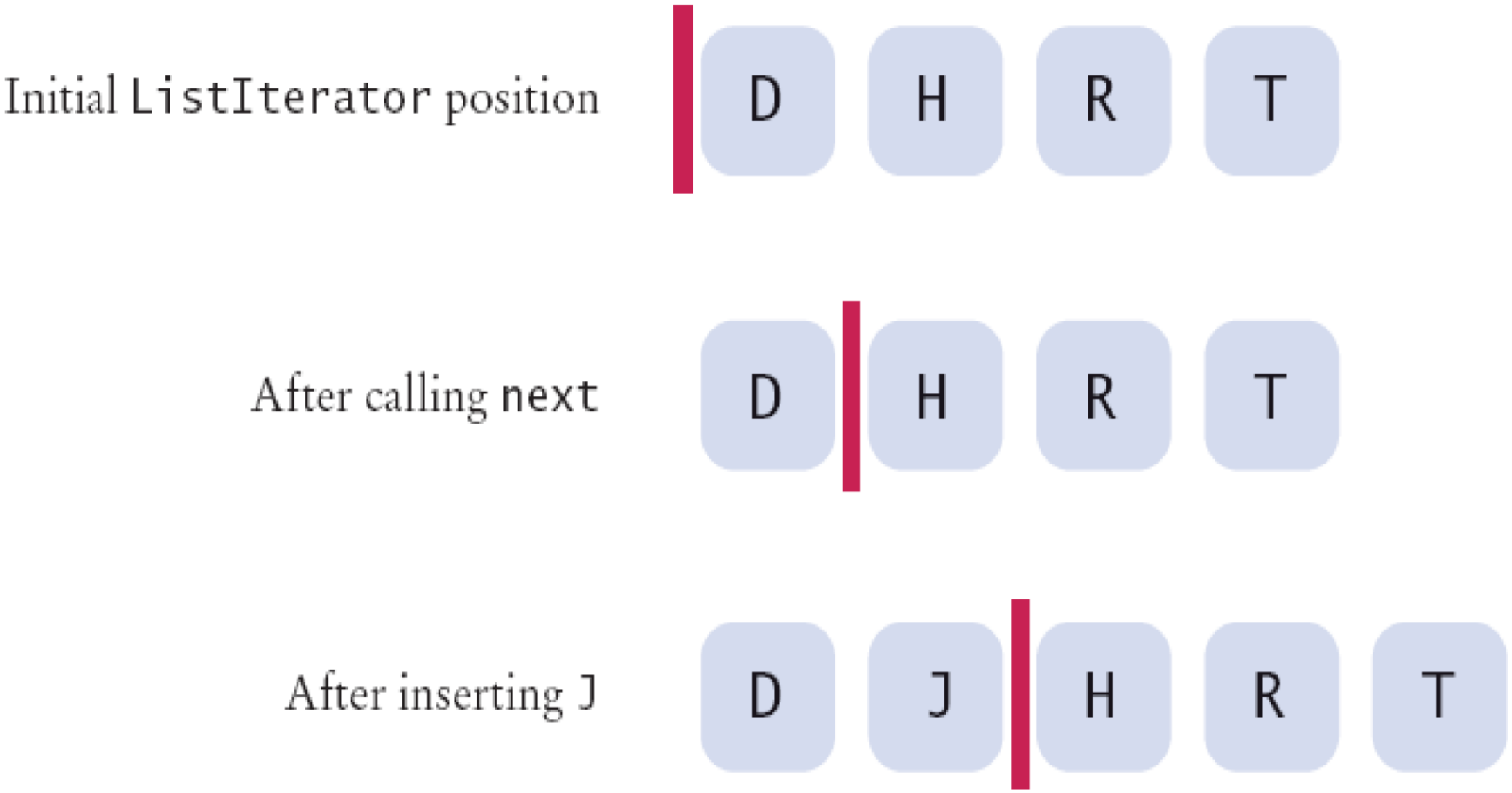
```
void addFirst(E obj)
void addLast(E obj)
E getFirst()
E getLast()
E removeFirst()
E removeLast()
```

List Iterator

- Per poter accedere ad un elemento della LinkedList bisogna utilizzare un **`ListIterator`**
  - Il ListIterator da accesso agli elementi di una lista
    - Incapsula una qualsiasi posizione della lista
    - Permette di gestire la visita e la gestione della lista



Una vista concettuale



List Iterator

- Una sorta di puntatore fra due elementi di una lista
  - Come il cursore di editor di testo . . .
- Il metodo **`listIterator`** di **`LinkedList`** crea e restituisce un iterator

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator=employeeNames.listIterator();
```

List Iterator

- Inizialmente posizionato prima del primo elemento
- Il metodo **`next`** sposta l'iterator

```
iterator.next();
```

- **`next`** lancia una eccezione **`NoSuchElementException`** se si è già oltrepassato il limite della lista
- **`hasNext`** è vero se la lista ha ancora elementi

```
if (iterator.hasNext())
   iterator.next();
```

List Iterator

- Otre ad avanzare, il metodo **`next`** restituisce l'elemento che si oltrepassa

```
while (iterator.hasNext())
{
    String name = iterator.next();
    Do something with name
}
```

- Versione breve:

```
for (String name : employeeNames)
{
   Do something with name
}
```

in realtà il for generalizzato usa un iterator

List Iterator

- **`LinkedList`** in realtà è una *doubly linked list*
  - Ci sono due links:
    - Il prossimo
    - Il precedente
- Metodi:
  - **`hasPrevious`**
  - **`previous`**

Aggiungere / rimuovere elementi

- Il metodo **`add`**:
  - Aggiunge un elemento dopo la posizione dell'iterator
  - Muove l'iterator dopo l'elemento inserito

```
iterator.add("Juliet");
```

Aggiungere / rimuovere elementi

- Il metodo **`remove`**
  - Rimuove l'elemento che era stato restituito dall'ultima call a **`next`** o **`previous`**

```
//Remove all names that fulfill a certain condition
while (iterator.hasNext())
{
   String name = iterator.next();
   if (name fulfills condition)
      iterator.remove();
}
```

  - Può essere invocato una sola volta dopo l'invocazione di **`next`** o **`previous`**
  - Non può essere invocato immediatamente dopo una call ad **`add`**
    - Eccezione: **`IllegalStateException`**

Esempio

- **`ListTester`** - un programma che:
  - Inserisce oggetti String in una list
  - Visita la lista, aggiungendo e rimuovendo elementi
  - Stampa la lista

File ListTester.java

```
01: import java.util.LinkedList;
02: import java.util.ListIterator;
03:
04: /**
05:    A program that demonstrates the LinkedList class
06: */
07: public class ListTester
08: {
09:    public static void main(String[] args)
10:    {
11:       LinkedList<String> staff=new LinkedList<String>();
12:       staff.addLast("Dick");
13:       staff.addLast("Harry");
14:       staff.addLast("Romeo");
15:       staff.addLast("Tom");
16:
17:     // | in the comments indicates the iterator position
18:
```

File ListTester.java

```
19:         ListIterator<String> iterator
20:               = staff.listIterator(); // |DHRT
21:         iterator.next(); // D|HRT
22:         iterator.next(); // DH|RT
23:
24:         // Add more elements after second element
25:
26:         iterator.add("Juliet"); // DHJ|RT
27:         iterator.add("Nina"); // DHJN|RT
28:
29:         iterator.next(); // DHJNR|T
30:
31:         // Remove last traversed element
32:
33:         iterator.remove(); // DHJN|T
34:
```

File ListTester.java

```
35:         // Print all elements
36:
37:         for (String name : staff)
38:            System.out.println(name);
39:      }
40: }
```

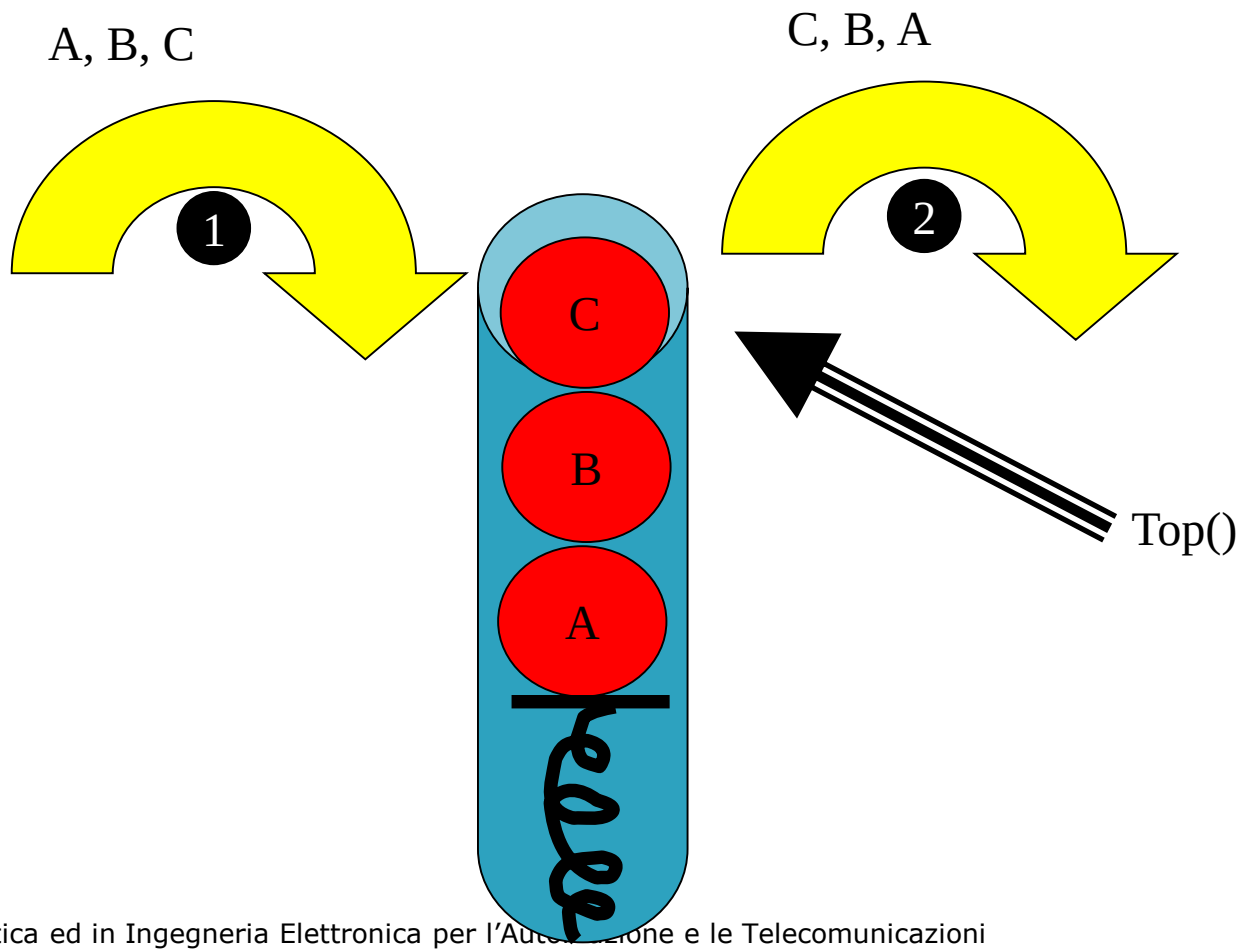
File `ListTester.java`

- Output:

```
Dick
Harry
Juliet
Nina
Tom
```

Stack

Di tipo LIFO
Last In First Out



Stack

- Modella una struttura dati di tipo
  - **`public Stack()`**
    - Costruisce uno stack vuoto
  - **`public void push(Object elem)`**
    - Inserisce elem in testa allo stack
  - **`public Object peek()`**
    - Restituisce l'elemento che sta al top dello stack (definito sullo stack non vuoto)
  - **`public Object pop()`**
    - Elimina l'elemento al top dello stack e lo restituisce (definito sullo stack non vuoto)
  - **`public boolean isEmpty()`**
    - Restituisce true se lo stack è vuoto, false altrimenti
  - **`public int size()`**
    - Restituisce il numero di elementi dello stack

Java library Stack

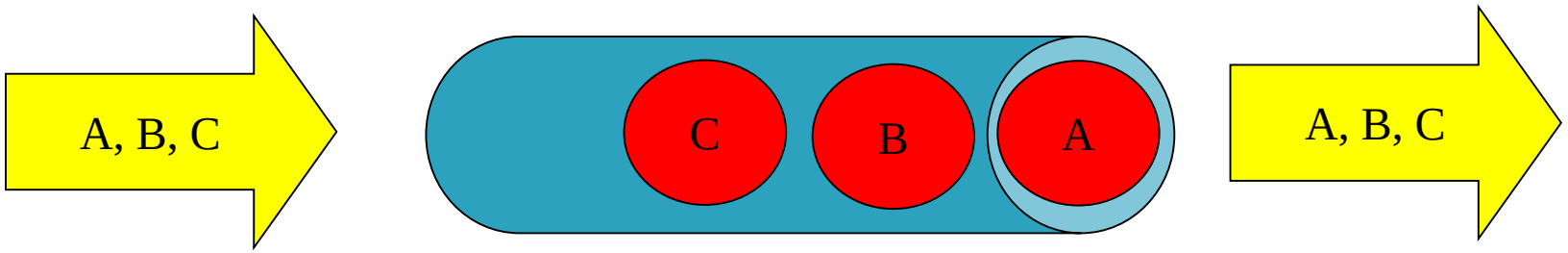
- Stack

```
Stack<String> s = new Stack<String>();
s.push("A");
s.push("B");
s.push("C");
// The following loop prints C, B, and A
while (s.size() > 0)
   System.out.println(s.pop());
```

- Implementare lo stack usando un array

Queue

Di tipo FIFO
First **I**n **F**irst **O**ut



Queue: pensata per programmi multithreaded ...

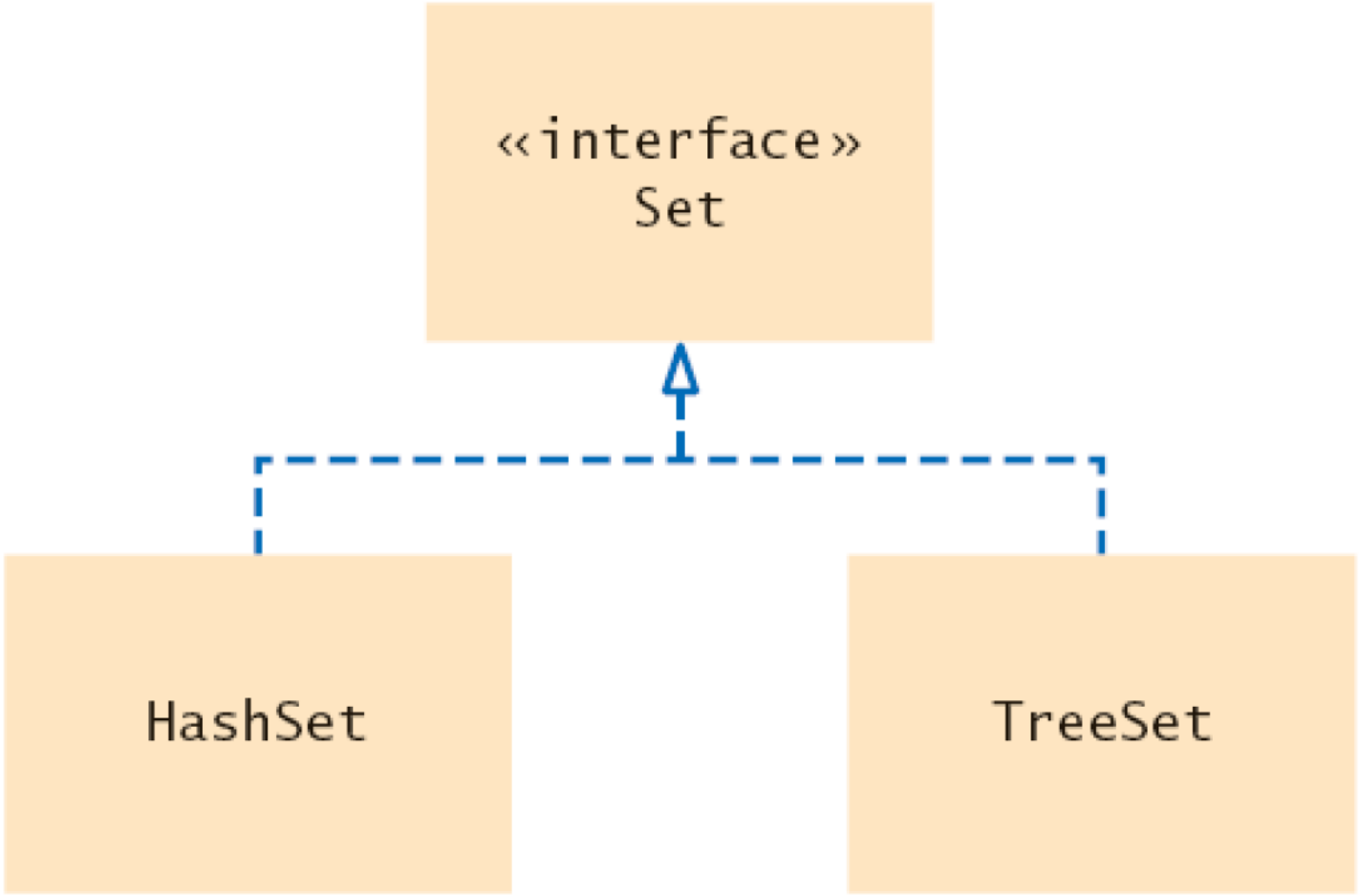
Queue

- Modella una struttura dati di tipo Queue
  - **`public Queue()`**
    - Costruisce una coda vuota
  - **`public void insert(Object elem)`**
    - Inserisce elem al rear alla coda
  - **`public Object element()`**
    - Restituisce l'elemento che sta in front alla coda (definito sulla coda non vuota)
  - **`public Object extract()`**
    - Elimina l'elemento in front alla coda e lo restituisce (definito sulla coda non vuota)
  - **`public boolean isEmpty()`**
    - Restituisce true se la coda è vuota, false altrimenti
  - **`public int size()`**
    - Restituisce il numero di elementi della coda

Sets

- Set: collezione non ordinata di oggetti . . .
- La standard Java library fornisce due implementazioni del set, basate su due strutture diverse
  - HashSet
  - TreeSet
- Entrambe implementano la stessa interfaccia **`Set`**

Set



Metodi del Set

```
//Creating a hash set
Set<String> names = new HashSet<String>();
```

```
//Adding an element names.add("Romeo");
```

```
//Removing an element names.remove("Juliet");
```

```
//Is element in set
if (names.contains("Juliet") { . . .}
```

Iterator

- Per la visita di un set si utilizza un iterator
- L'ordine di visita NON è quello di inserimento

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   Do something with name
}

// Or, using the "for each" loop

for (String name : names)
{
   Do something with name
}
```

File SetTester.java

```
01: import java.util.HashSet;
02: import java.util.Iterator;
03: import java.util.Scanner;
04: import java.util.Set;
05:
06:
07: /**
08:    This program demonstrates a set of strings. The user
09:    can add and remove strings.
10: */
11: public class SetTester
12: {
13:    public static void main(String[] args)
14:    {
15:       Set<String> names = new HashSet<String>();
16:       Scanner in = new Scanner(System.in);
17:
```

File SetTester.java

```
18:         boolean done = false;
19:         while (!done)
20:         {
21:            System.out.print("Add name, Q when done: ");
22:            String input = in.next();
23:            if (input.equalsIgnoreCase("Q"))
24:               done = true;
25:            else
26:            {
27:               names.add(input);
28:               print(names);
29:            }
30:         }
31:
32:         done = false;
33:         while (!done)
34:         {
```

File SetTester.java

```
35:          System.out.println("Remove name, Q when done");
36:          String input = in.next();
37:          if (input.equalsIgnoreCase("Q"))
38:              done = true;
39:          else
40:          {
41:              names.remove(input);
42:              print(names);
43:          }
44:       }
45:    }
46:
47:    /**
48:       Prints the contents of a set of strings.
49:       @param s a set of strings
50:    */
51:    private static void print(Set<String> s)
52:    {
```

File SetTester.java

```
53:         System.out.print("{ ");
54:         for (String element : s)
55:         {
56:            System.out.print(element);
57:            System.out.print(" ");
58:         }
59:         System.out.println("}");
60:     }
61: }
62:
63:
```

File SetTester.java

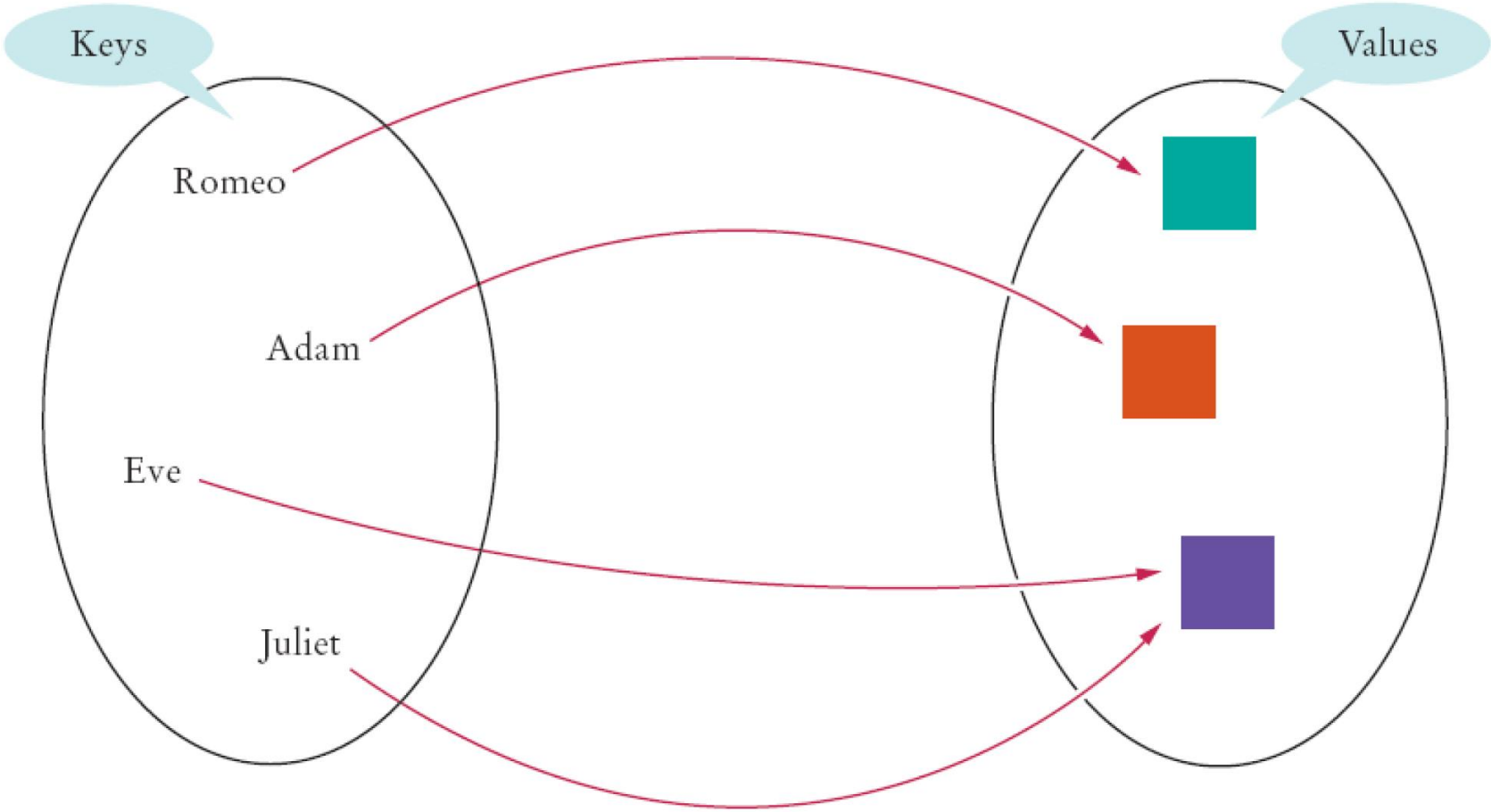
- Output

```
Add name, Q when done: Dick
{ Dick }
Add name, Q when done: Tom
{ Tom Dick }
Add name, Q when done: Harry
{ Harry Tom Dick }
Add name, Q when done: Tom
{ Harry Tom Dick }
Add name, Q when done: Q
Remove name, Q when done: Tom
{ Harry Dick }
Remove name, Q when done: Jerry
{ Harry Dick }
Remove name, Q when done: Q
```

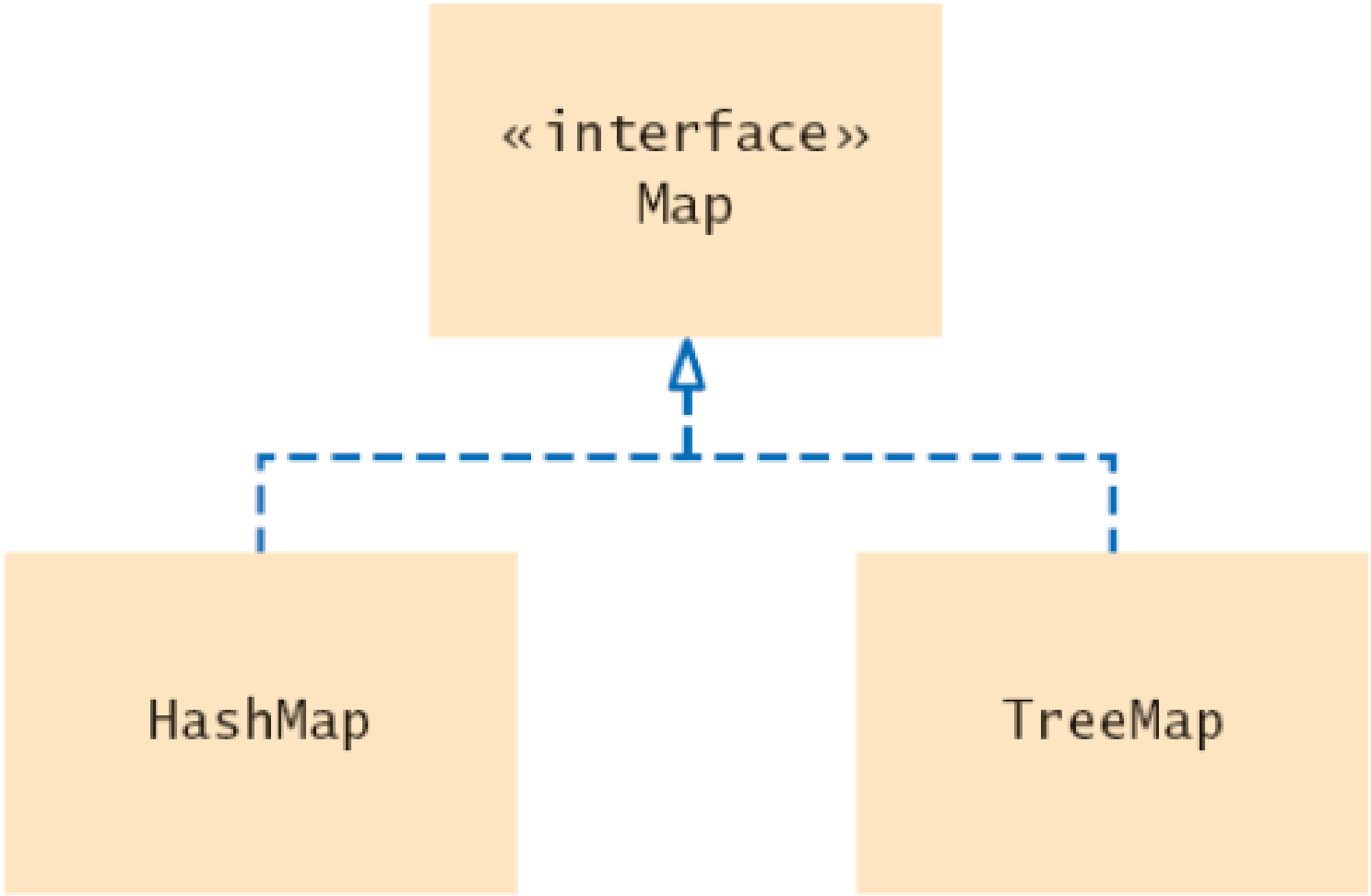
Maps

- Una mappa memorizza coppie (chiave oggetto)
  - Le chiavi sono uniche
- Come per gli insiemi, i Set:
- `Map` interface
  - HashMap
  - TreeMap

Un esempio di una Map



Map Classes and Interfaces



HashMap

```
//Creating a HashMap
Map<String, Color> favoriteColors
   = new HashMap<String, Color>();
```

```
//Adding an association
favoriteColors.put("Juliet", Color.PINK);
```

```
//Changing an existing association
favoriteColor.put("Juliet",Color.RED);
```

HashMap

```
//Getting the value associated with a key
Color julietsFavoriteColor
            = favoriteColors.get("Juliet");
```

```
//Removing a key and its associated value
favoriteColors.remove("Juliet");
```

Stampare tutte le coppie

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
   Color value = m.get(key);
   System.out.println(key + "->" + value);
}
```

File MapTester.java

```
01: import java.awt.Color;
02: import java.util.HashMap;
03: import java.util.Iterator;
04: import java.util.Map;
05: import java.util.Set;
06:
07: /**
08:    This program tests a map that maps names to colors.
09: */
10: public class MapTester
11: {
12:    public static void main(String[] args)
13:    {
14:       Map<String, Color> favoriteColors
15:             = new HashMap<String, Color>();
16:       favoriteColors.put("Juliet", Color.pink);
17:       favoriteColors.put("Romeo", Color.green);
```

File MapTester.java

```
18:         favoriteColors.put("Adam", Color.blue);
19:         favoriteColors.put("Eve", Color.pink);
20:
21:         Set<String> keySet = favoriteColors.keySet();
22:         for (String key : keySet)
23:         {
24:            Color value = favoriteColors.get(key);
25:            System.out.println(key + "->" + value);
26:         }
27:     }
28: }
```

File MapTester.java

- Output

```
Romeo->java.awt.Color[r=0,g=255,b=0]
Eve->java.awt.Color[r=255,g=175,b=175]
Adam->java.awt.Color[r=0,g=0,b=255]
Juliet->java.awt.Color[r=255,g=175,b=175]
```