Capitolo 14
Introduzione alle strutture di dati

Cay S. Horstmann
Concetti di informatica e fondamenti di Java
quarta edizione

Obiettivi del capitolo

- Capire come utilizzare le liste concatenate presenti nella libreria standard
- Saper usare gli iteratori per scandire le liste concatenate
- Analizzare la realizzazione di liste concatenate
- Comprendere la differenza tra tipi di dati astratti e concreti
- Conoscere l'efficienza delle operazioni fondamentali per le liste e gli array
- Acquisire familiarità con pile e code

Utilizzare liste concatenate

- Una lista concatenata è composta da un certo numero di nodi, ciascuno dei quali contiene un riferimento al nodo successivo
- Aggiungere e rimuovere elementi in una lista concatenata è un'operazione efficiente
- Visitare in sequenza gli elementi di una lista concatenata è efficiente, ma accedervi in ordine casuale non lo è

Inserire un elemento in una lista concatenata

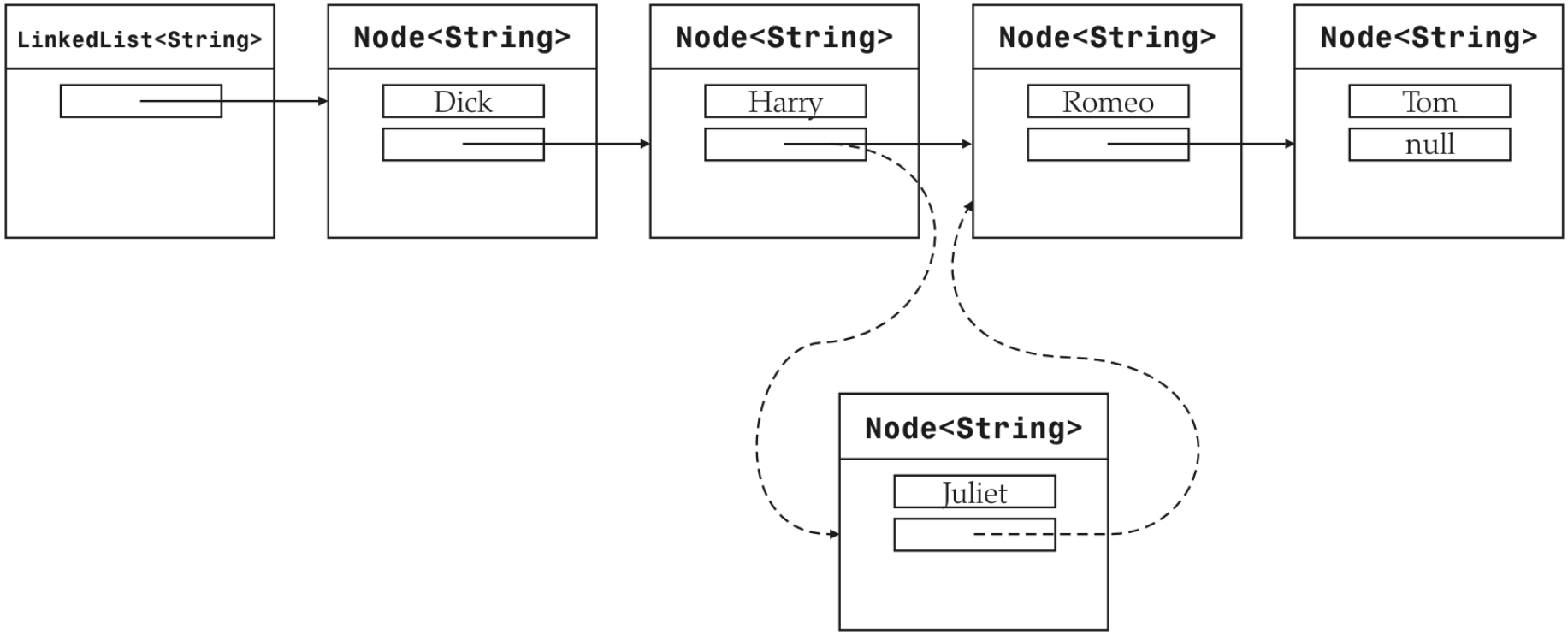

Figura 1

La classe `LinkedList`

- 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 elemento della lista

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

Il metodo `ListIterator`

- 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 concatenata

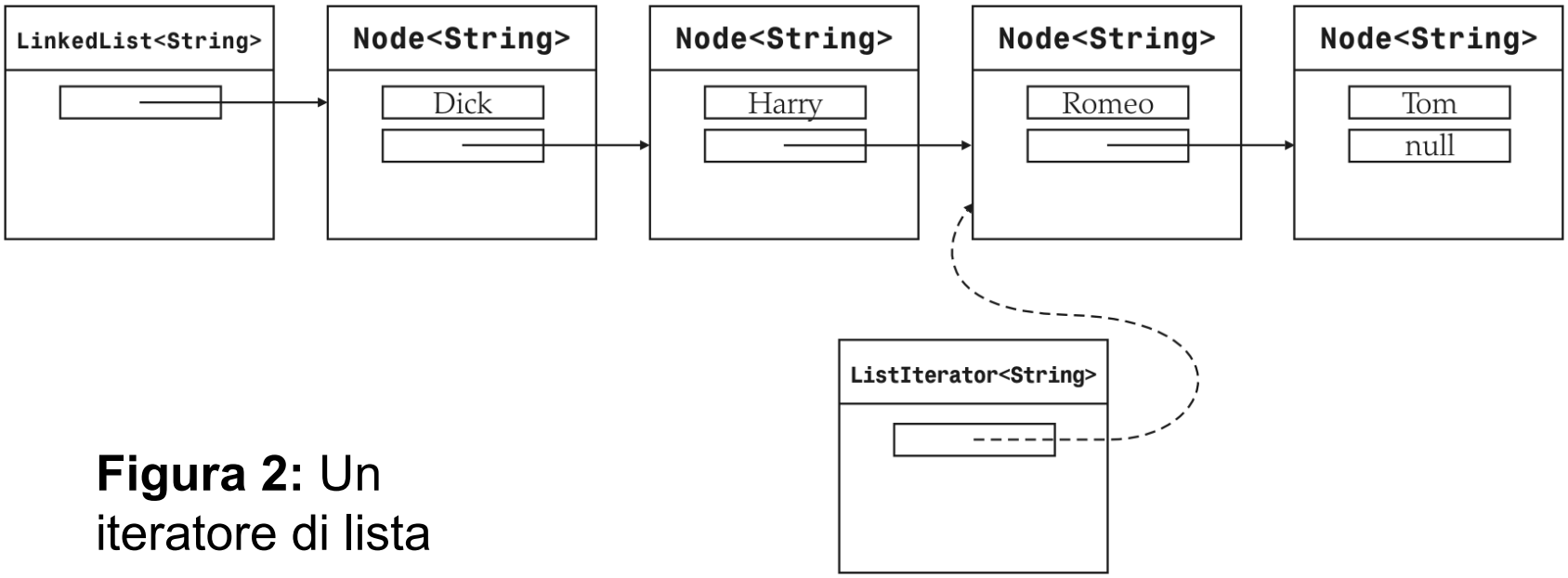

Figura 2: Un iteratore di lista

Una visione astratta dell'iteratore di lista

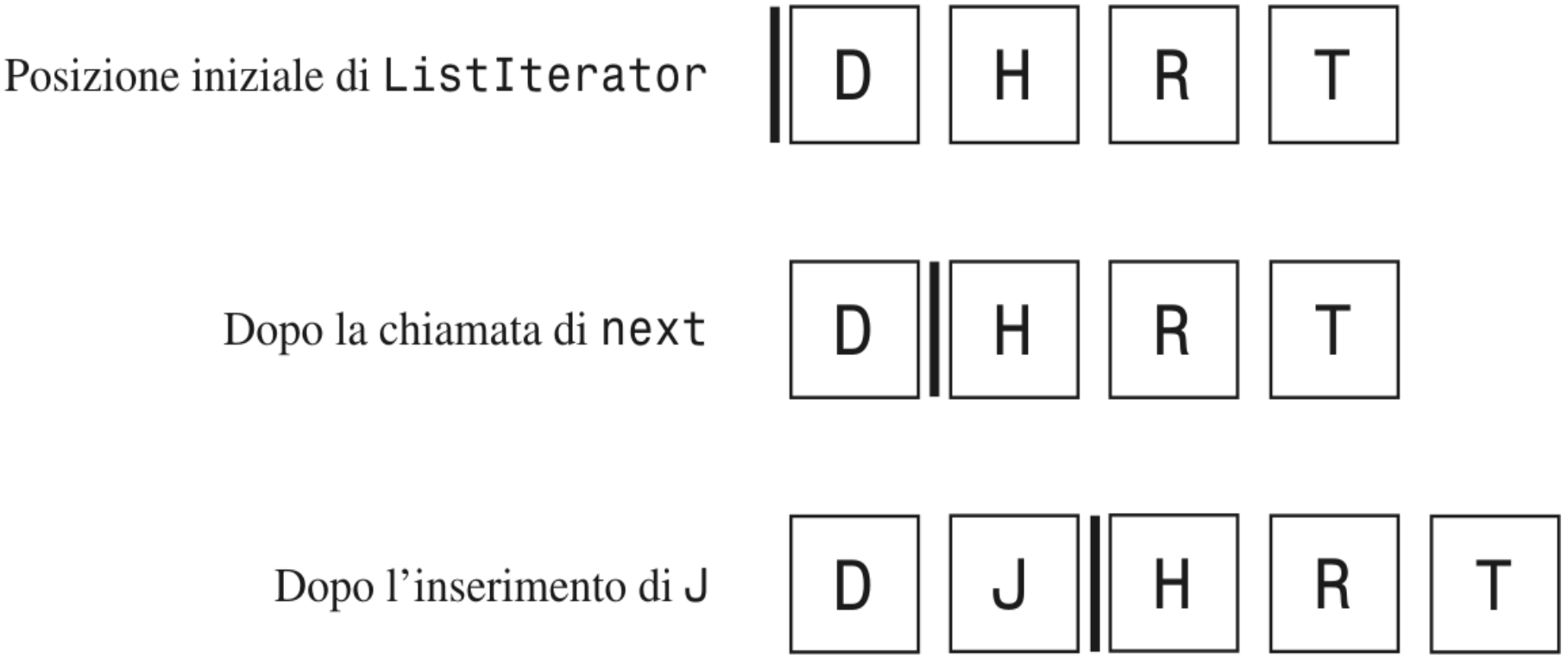

Figura 3

L'iteratore di lista

- Considerate l'iteratore come qualcosa che punta fra due elementi, esattamente come il cursore di un elaboratore di testi punta fra due caratteri
- Per creare un iteratore di lista utilizzate il metodo `listIterator` della classe `LinkedList`:

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

L'iteratore di lista

- All'inizio, l'iteratore punta alla posizione che precede il primo elemento
- Potete spostare la posizione dell'iteratore invocando il metodo `next`

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

- Il metodo `next` lancia l'eccezione `NoSuchElementException` se l'iteratore si trova già oltre la fine della lista.
- `hasNext` restituisce `true` se c'è un elemento successivo.

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

L’iteratore di lista

- Il metodo `next` restituisce l’elemento appena attraversato dall’iteratore

```
while iterator.hasNext()
{
    String name = iterator.next();
    Fa qualcosa con name
}
```

L’iteratore di lista

- Scorciatoia:

```
for (String name : employeeNames)
{
   Fa qualcosa con name
}
```

- Se il vostro ciclo deve semplicemente visitare tutti gli elementi di una lista concatenata, potete usare un ciclo `for` generalizzato

L’iteratore di lista

- Una lista di questo genere è detta *lista doppiamente concatenata* (*doubly linked list*)
- I nodi della classe `LinkedList` contengono due collegamenti:
  - uno verso l’elemento successivo e
  - uno verso il precedente
- Potete usare i metodi `previous` e `hasPrevious` dell’interfaccia `ListIterator` per spostare l’iteratore all’indietro.

Aggiungere e rimuovere da una `LinkedList`

- Il metodo `add`
  - aggiunge un oggetto dopo la posizione attuale dell'iteratore
  - sposta la posizione dell'iteratore dopo il nuovo elemento

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

Aggiungere e rimuovere da una `LinkedList`

- Il metodo `remove`

  - elimina l'oggetto che era stato restituito dall'ultima invocazione di `next` o `previous`

```
//Elimina tutti gli oggetti che soddisfano una determinata condizione
while (iterator.hasNext())
{
   String name = iterator.next();
   if (name soddisfa la condizione)
      iterator.remove();
}
```

Aggiungere e rimuovere da una `LinkedList`

- Dovete prestare molta attenzione quando invocate `remove`:
  - Può essere chiamato una sola volta dopo aver invocato `next` o `previous`
  - Non potete chiamarlo immediatamente dopo un'invocazione di `add`
  - Se chiamato in modo improprio, il metodo lancia una eccezione `IllegalStateException`

File `ListTester.java`

- Programma dimostrativo che

  - inserisce stringhe in una lista

  - la percorre iterativamente, aggiungendo ed eliminando elementi

  - stampa l'intera lista

File ListTester.java

```
01: import java.util.LinkedList;
02: import java.util.ListIterator;
03:
04: /**
05:    Programma che illustra il funzionamento della classe LinkedList
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:       // il segno | nei commenti indica la posizione dell'iteratore
18:
```

File `ListTester.java`

```
19:         ListIterator<String> iterator
20:               = staff.listIterator(); // |DHRT
21:         iterator.next(); // D|HRT
22:         iterator.next(); // DH|RT
23:
24:         // aggiunge altri elementi dopo il secondo
25:
26:         iterator.add("Juliet"); // DHJ|RT
27:         iterator.add("Nina"); // DHJN|RT
28:
29:         iterator.next(); // DHJNR|T
30:
31:         // toglie l'ultimo elemento attraversato
32:
33:         iterator.remove(); // DHJN|T
34:
```

File ListTester.java

```
35:         // Stampa tutti gli elementi
36:
37:         for (String name : staff)
38:            System.out.println(name);
39:         System.out.println();
40:         System.out.println("Expected: Dick Harry Juliet Nina Tom");

41:     }
42: }
```

File `ListTester.java`

- Visualizza

```
Dick Harry Juliet Nina Tom
Expected: Dick Harry Juliet Nina Tom
```

Realizzare liste concatenate

- Avete visto come si utilizza la classe “lista concatenata”
- Ora esamineremo la realizzazione di una versione semplificata di questa classe
- In questo modo potrete vedere come le operazioni di lista manipolano i collegamenti mentre la lista viene modificata
- Per non complicare il codice realizzeremo soltanto una lista semplicemente concatenata

  - la classe fornirà l’accesso diretto soltanto al primo elemento della lista, non all’ultimo
- La nostra classe non sarà parametrica

  - memorizzeremo valori di tipo `Object` e inseriremo i cast opportuni quando li recupereremo dal contenitore

Realizzare liste concatenate

- Un oggetto `Node` (un nodo della catena) memorizza un oggetto e un riferimento al nodo successivo
- Sia i metodi della classe lista concatenata sia quelli della classe iteratore accedono di frequente alle variabili di esemplare di `Node`
- Per semplificarne l'uso

  - non definiamo come private tali variabili

  - rendiamo `Node` una classe interna privata della classe `LinkedList`

  - Dato che nessun metodo della lista restituisce un oggetto di tipo `Node`, non ci sono rischi a lasciare pubbliche le variabili di esemplare di quest'ultima classe.

Realizzare liste concatenate

```
public class LinkedList
   {
      . . .
      private class Node
      {
         public Object data;
         public Node next;
      }
   }
```

Realizzare liste concatenate

- La classe `LinkedList` contiene un riferimento `first` al primo nodo

```
public class LinkedList
{
   public LinkedList()
   {
      first = null;
   }
   public Object getFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      return first.data;
   }
   . . .
   private Node first;

}
```

Aggiungere un nuovo nodo

- Quando viene aggiunto alla lista un nuovo nodo:
  - diventa la testa della lista
  - il nodo che in precedenza era la testa della lista diventa il suo successivo

Aggiungere un nuovo nodo

- Il metodo `addFirst`

```
public class LinkedList
{
   . . .
   public void addFirst(Object obj)
   {

      Node newNode = new Node();
      newNode.data = element;
      newNode.next = first;

      first = newNode;
   }
   . . .
}
```

Aggiungere un nodo all'inizio di una lista concatenata

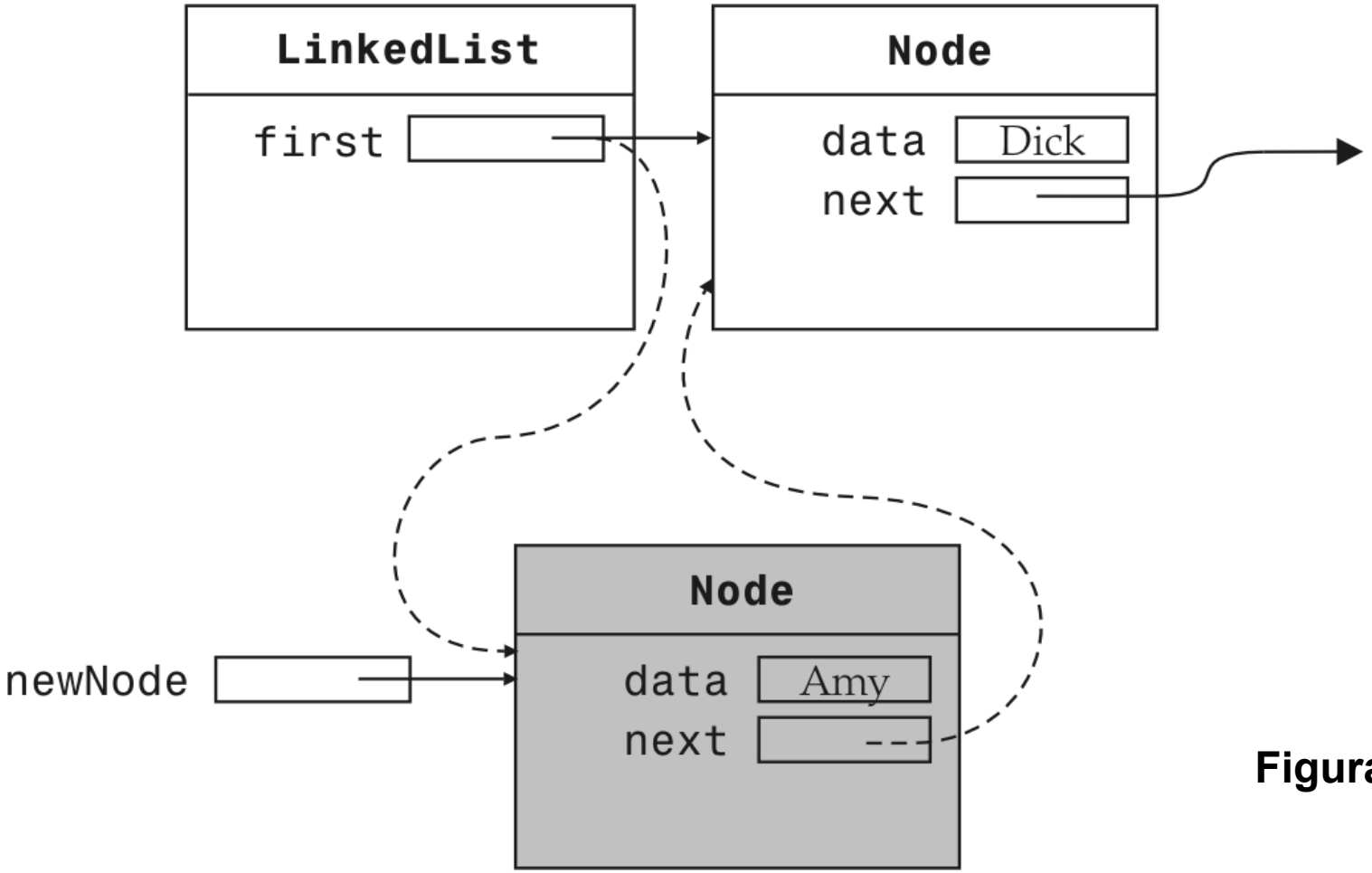

Figura 4

Togliere il primo elemento

- L'eliminazione del primo elemento della lista funziona nel modo seguente:

  - Il dato presente nel primo nodo viene memorizzato e successivamente restituito come risultato del metodo.

  - Il successore del primo nodo diventa il primo nodo della lista accorciata

  - Non vi sono ulteriori riferimenti al vecchio nodo e il *garbage collector* lo riciclerà al momento opportuno

Togliere il primo elemento

- Il metodo `removeFirst`

```
public class LinkedList
{
   . . .
   public Object removeFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      Object element = first.data;
      first = first.next;
      return element;
   }
   . . .
}
```

Togliere un nodo dalla prima posizione di una lista concatenata

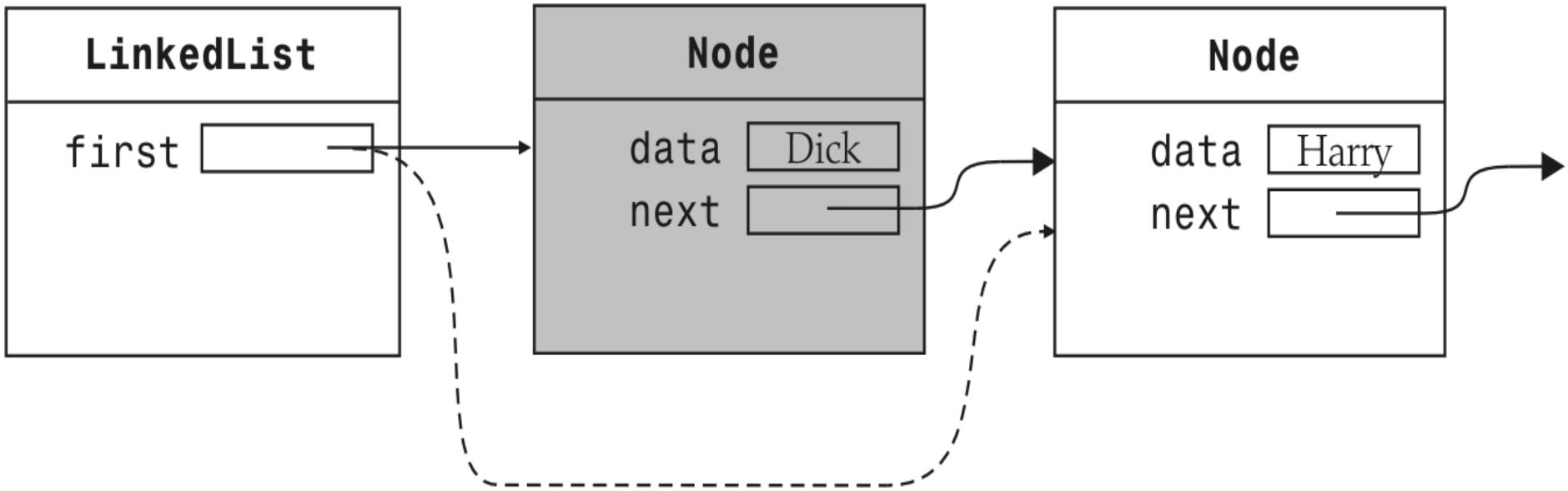

Figura 5

La classe iteratore

- La classe `LinkedList` definisce una classe interna privata `LinkedListIterator`
- Realizza l'interfaccia `ListIterator` semplificata.
- Ha accesso alle caratteristiche private della classe `LinkedList`, tra cui il campo first e la classe privata `Node`.
- I programmi che utilizzano la classe `LinkedList` non hanno bisogno di conoscere il nome della classe iteratore: è sufficiente sapere che è una classe che realizza l'interfaccia `ListIterator`

LinkedListIterator

- La classe `LinkedListIterator`

```
public class LinkedList
{
   . . .
   public ListIterator listIterator()
   {
      return new LinkedListIterator();
   }
   private class LinkedListIterator implements ListIterator
   {
      public LinkedListIterator()
      {
         position = null;
         previous = null;
      }
```

Continua…

LinkedListIterator

```
    . . .
      private Node position;
      private Node previous;
    }
    . . .
}
```

Il metodo `next`

- Ciascun oggetto iteratore ha un riferimento `position` all'ultimo nodo visitato
- Memorizziamo anche un riferimento al penultimo nodo
- Il riferimento `position` viene fatto progredire fino a `position.next`
- La vecchia posizione viene memorizzata in `previous`
- Se l'iteratore è nella posizione che precede il primo nodo della lista, la vecchia `position` è `null` e deve essere impostata su `first`

Il metodo `next`

```
private class LinkedListIterator

implements ListIterator

{
   . . .
   public Object next()
   {
      if (!hasNext())
         throw new NoSuchElementException();
      previous = position; // memorizza per remove
      if (position == null)
         position = first;
      else
         position = position.next;
      return position.data;
   }
   . . .

}
```

Il metodo `hasNext`

- È previsto che il metodo `next` venga invocato soltanto quando l'iteratore non è ancora arrivato alla fine della lista
- L'iteratore è alla fine

  - se la lista è vuota (vale a dire, se `first == null`)

  - oppure se non vi è alcun elemento dopo la posizione corrente (`position.next == null`)

Il metodo `hasNext`

```
private class LinkedListIterator

implements ListIterator
   {
      . . .
      public boolean hasNext()
      {
         if (position == null)
            return first != null;
         else
            return position.next != null;
      }
      . . .
   }
```

Il metodo `remove`

- Se l'elemento che deve essere rimosso è il primo nodo, invochiamo semplicemente `removeFirst`
- In caso contrario, il nodo da rimuovere si trova all'interno della lista, per cui si dovrà aggiornare il riferimento `next` del nodo che lo precede in modo da saltare l'elemento rimosso
- Se il riferimento `previous` è uguale a `position`, allora

  - questa invocazione di `remove` non segue immediatamente un'invocazione di `next`

  - lanciamo un'eccezione di tipo `IllegalStateException`
- Non è lecito invocare `remove` due volte di seguito

  - il metodo `remove` rende il riferimento `previous` uguale a `position`

Il metodo `remove`

```
private class LinkedListIterator

implements ListIterator

{

   public void remove()
   {
      if (previous == position)
         throw new IllegalStateException();
      if (position == first)
      {
         removeFirst();
      }
      else
      {

         previous.next = position.next;

      }

      position = previous;
   }
```

Togliere un nodo da una posizione interna di una lista concatenata

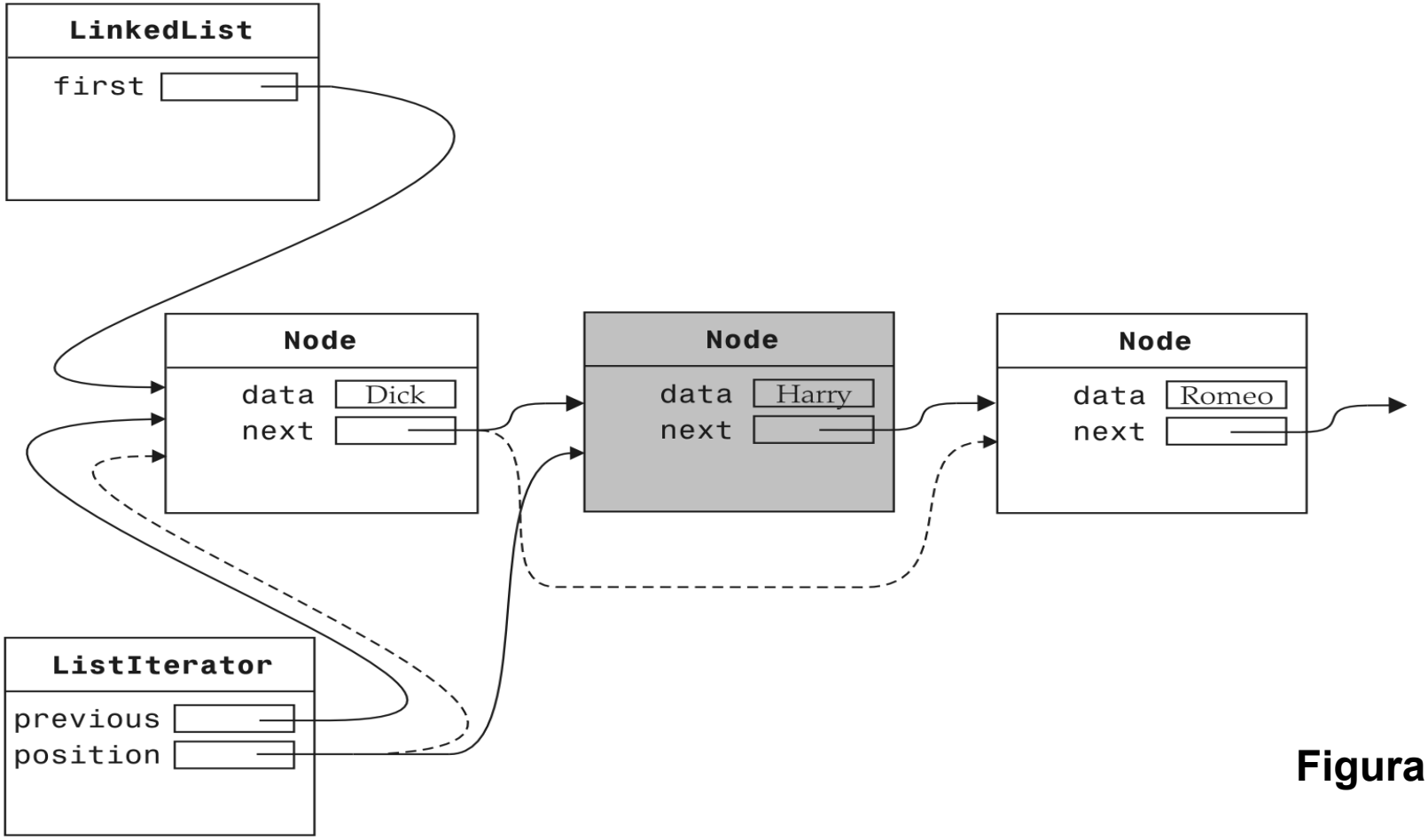

Figura 6

Il metodo `set`

- Modifica il dato memorizzato nell'elemento visitato in precedenza
- Il metodo `set`

```
public void set(Object element)
{
   if (position == null)
      throw new NoSuchElementException();
   position.data = element;
}
```

Il metodo `add`

- L'operazione più complessa è l'aggiunta di un nodo
- Inserire il nuovo nodo dopo la posizione corrente e impostare il successore del nuovo nodo al successore della posizione corrente

Il metodo add

```
public void add(Object obj)
   {
      if (position == null)
      {
         addFirst(element);
         position = first;
      }
      else
      {
         Node newNode = new Node();
         newNode.data = element;
         newNode.next = position.next;
         position.next = newNode;
         position = newNode;
      }

      previous = position;
   }
```

Aggiungere un nodo in una posizione interna di una lista concatenata

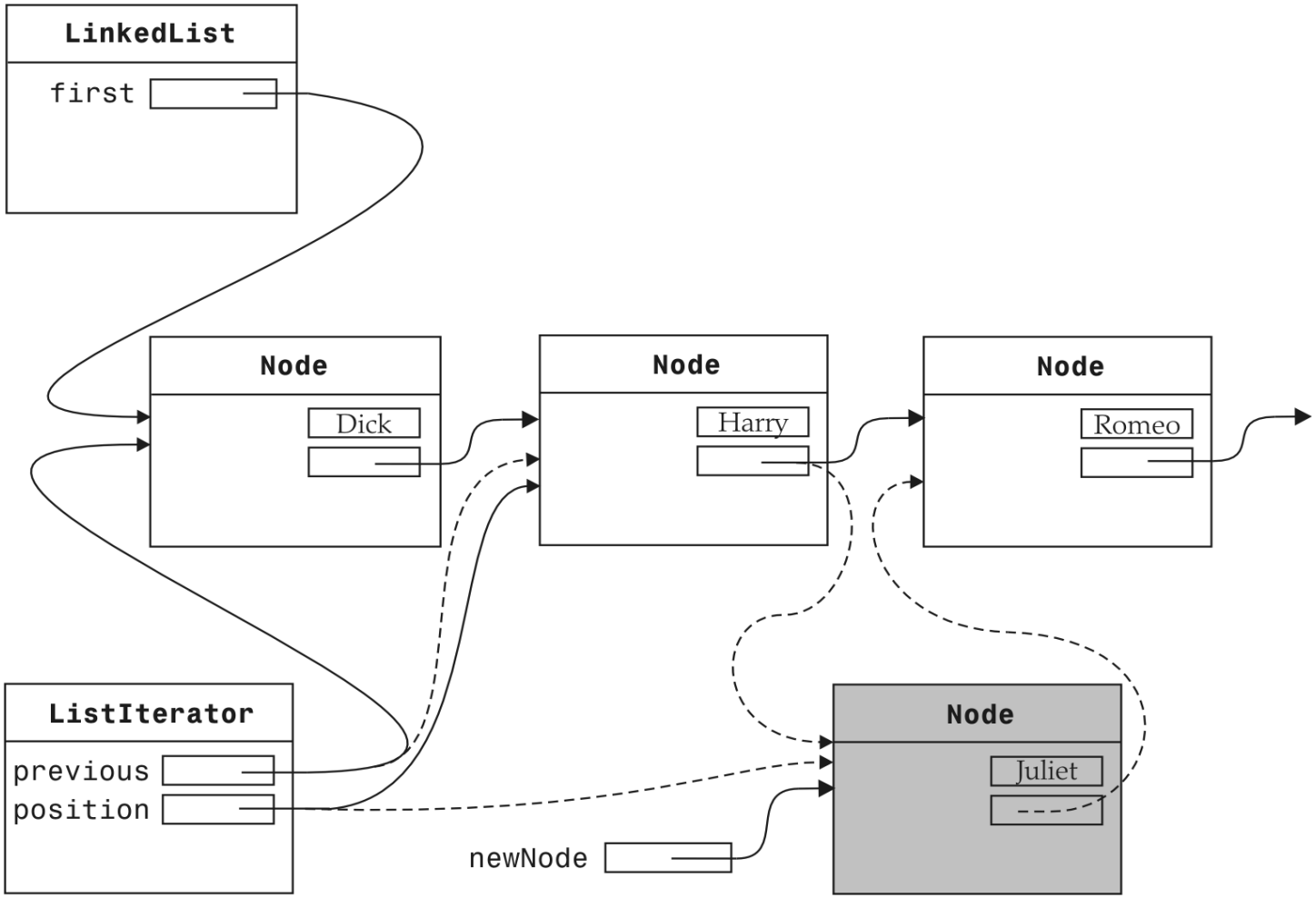

Figura 7

File LinkedList.java

```
001: import java.util.NoSuchElementException;
002:
003: /**
004:    Una lista concatenata è una sequenza di nodi dotata di
005:    efficienti meccanismi di inserimento ed eliminazione.
006:    Questa classe ha un sottoinsieme dei metodi della classe
   standard java.util.LinkedList
007:.
008: */
009: public class LinkedList
010: {
011:    /**
012:       Costruisce una lista concatenata vuota.
013:    */
014:    public LinkedList()
015:    {
016:       first = null;
017:    }
018:
```

Continua…

File LinkedList.java

```
019:    /**
020:    Restituisce il primo elemento della lista concatenata.
021:       @return il primo elemento della lista concatenata
022:    */
023:    public Object getFirst()
024:    {
025:       if (first == null)
026:          throw new NoSuchElementException();
027:       return first.data;
028:    }
029:
030:    /**
031:       Toglie il primo elemento dalla lista concatenata.
032:    @return l'elemento eliminato
033:    */
034:    public Object removeFirst()
035:    {
```

Continua…

File LinkedList.java

```
036:          if (first == null)
037:             throw new NoSuchElementException();
038:          Object element = first.data;
039:          first = first.next;
040:          return element;
041:       }
042:
043:       /**
044:        Aggiunge un elemento in testa alla lista concatenata.
045:       @param element l'elemento da aggiungere
046:       */
047:       public void addFirst(Object element)
048:       {
049:          Node newNode = new Node();
050:          newNode.data = element;
051:          newNode.next = first;
052:          first = newNode;
053:       }
054:
```

Continua…

File `LinkedList.java`

```
055:    /**
056:       Restituisce un iteratore per questa lista.
057:    @return un iteratore per questa lista
058:    */
059:    public ListIterator listIterator()
060:    {
061:       return new LinkedListIterator();
062:    }
063:
064:    private Node first;
065:
066:    private class Node
067:    {
068:       public Object data;
069:       public Node next;
070:    }
071:
```

Continua…

File LinkedList.java

```
072:    private class LinkedListIterator implements
   ListIterator
073:    {
074:       /**
075:          Costruisce un iteratore che punta alla testa
076:    della lista concatenata.
077:       */
078:       public LinkedListIterator()
079:       {
080:          position = null;
081:          previous = null;
082:       }
083:
084:       /**
085:          Sposta l'iteratore dopo l'elemento successivo.
086:     @return l'elemento attraversato
087:       */
```

Continua…

File LinkedList.java

```
088:       public Object next()
089:       {
090:          if (!hasNext())
091:             throw new NoSuchElementException();
092:          previous = position; // Memorizza per remove
093:
094:          if (position == null)
095:             position = first;
096:          else
097:             position = position.next;
098:
099:          return position.data;
100:       }
101:
102:       /**
103:          Controlla se c'è un elemento dopo la posizione
104:    dell'iteratore.
```

Continua…

File `LinkedList.java`

```
105:          @return true se c'è un elemento dopo la
106:    posizione dell'iteratore
107:       */
108:       public boolean hasNext()
109:       {
110:          if (position == null)
111:             return first != null;
112:          else
113:             return position.next != null;
114:       }
115:
116:       /**
117:          Aggiunge un elemento prima della posizione
    dell'iteratore
118:       e sposta l'iteratore dopo l'elemento inserito.
119:    @param element l'oggetto da aggiungere
120:       */
```

Continua…

File LinkedList.java

```
121:       public void add(Object element)
122:       {
123:          if (position == null)
124:          {
125:             addFirst(element);
126:             position = first;
127:          }
128:          else
129:          {
130:             Node newNode = new Node();
131:             newNode.data = element;
132:             newNode.next = position.next;
133:             position.next = newNode;
134:             position = newNode;
135:          }
136:          previous = position;
137:       }
138:
```

Continua…

File LinkedList.java

```
139:       /**
140:          Toglie l'ultimo elemento attraversato. Questo
   metodo può
141:    essere chiamato soltanto dopo una chiamata al metodo
        next().
142:       */
143:       public void remove()
144:       {
145:          if (previous == position)
146:             throw new IllegalStateException();
147:
148:          if (position == first)
149:          {
150:             removeFirst();
151:          }
152:          else
153:          {
154:             previous.next = position.next;
155:          }
```

Continua…

File `LinkedList.java`

```
156:          position = previous;
157:       }
158:
159:       /**
160:          Imposta il dato presente nell'ultimo elemento
161:     attraversato.
162:    @param element il dato da impostare
163:       */
164:       public void set(Object element)
165:       {
166:          if (position == null)
167:             throw new NoSuchElementException();
168:          position.data = element;
169:       }
170:
171:       private Node position;
172:       private Node previous;
173:    }
174: }
```

File ListIterator.java

```
01: /**
02: Un iteratore di lista consente l'accesso a una posizione in una
03: lista concatenata. Questa interfaccia contiene un sottoinsieme
04: dei metodi dell'interfaccia standard java.util.ListIterator,
05: in quanto i metodi per l'attraversamento all'indietro sono esclusi.
06: */
07: public interface ListIterator
08: {
09:    /**
10:       Sposta l'iteratore dopo l'elemento successivo.
11:       @return l'elemento attraversato
12:    */
13:    Object next();
14:
15:    /**
16: Controlla se c'è un elemento dopo la posizione dell'iteratore.
17:
```

Continua…

File `ListIterator.java`

```
18:          @return true se c'è un elemento dopo la posizione
   dell'iteratore
19:
20:     */
21:     boolean hasNext();
22:
23:     /**
24:     Aggiunge un elemento prima della posizione dell'iteratore
25: e sposta l'iteratore dopo l'elemento inserito.
26: @param element l'oggetto da aggiungere
27:     */
28:     void add(Object element);
29:
30:     /**
31:        Toglie l'ultimo elemento attraversato. Questo metodo può
32:      essere chiamato soltanto dopo un'invocazione di next().
33:     */
```

Continua…

File `ListIterator.java`

```
34:     void remove();
35:
36:     /**
37:     Imposta il dato presente nell'ultimo elemento
38:     attraversato.
39:     @param element il dato da impostare
40:     */
41:     void set(Object element);
42: }
```

Tipi di dati astratti e concreti

- Ci sono due diversi punti di vista dai quali osservare una lista concatenata

  - Uno consiste nel pensare a una realizzazione concreta di tale lista, con i suoi nodi tra loro collegati

  - D'altra parte, si può pensare al concetto *astratto* di lista concatenata: in astratto, una lista concatenata è una sequenza ordinata di dati che può essere attraversata con un iteratore

Tipi di dati astratti e concreti

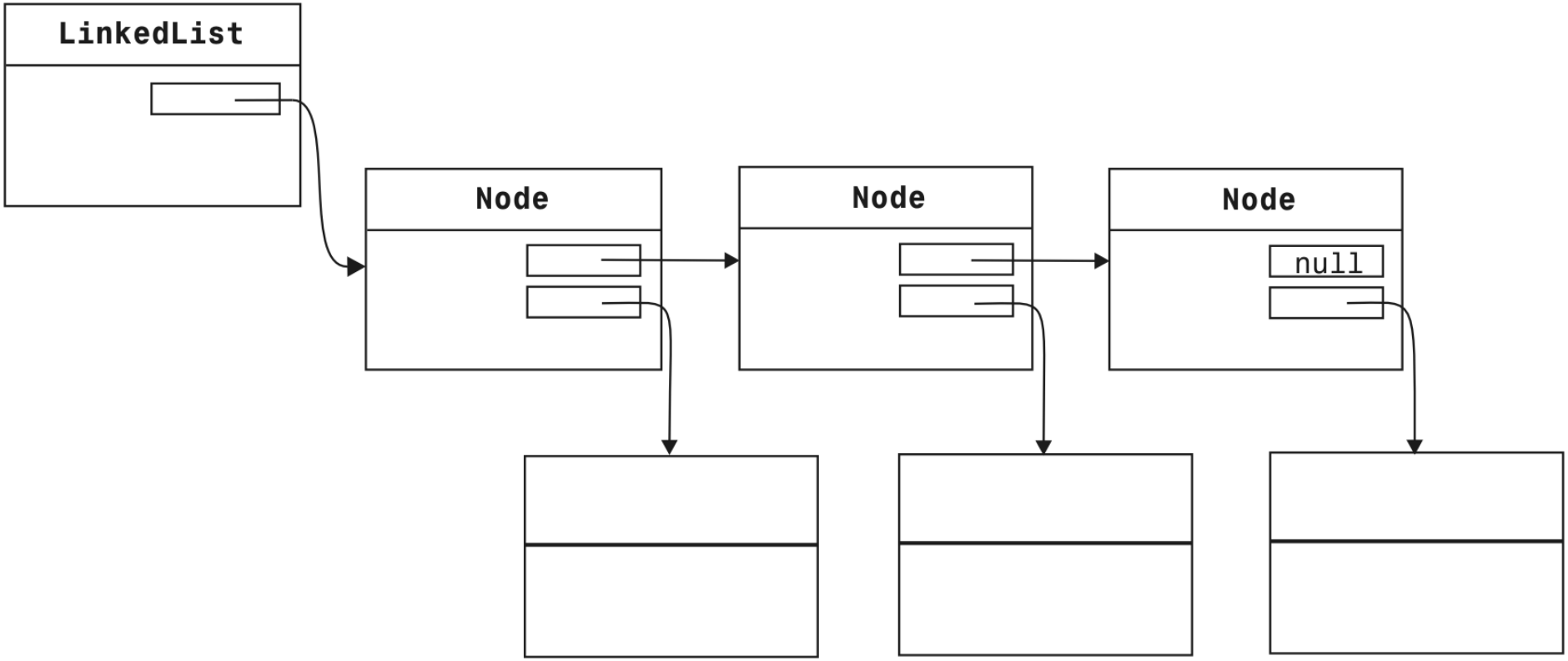

Figura 8: Visione concreta di una lista concatenata

Tipi di dati astratti e concreti

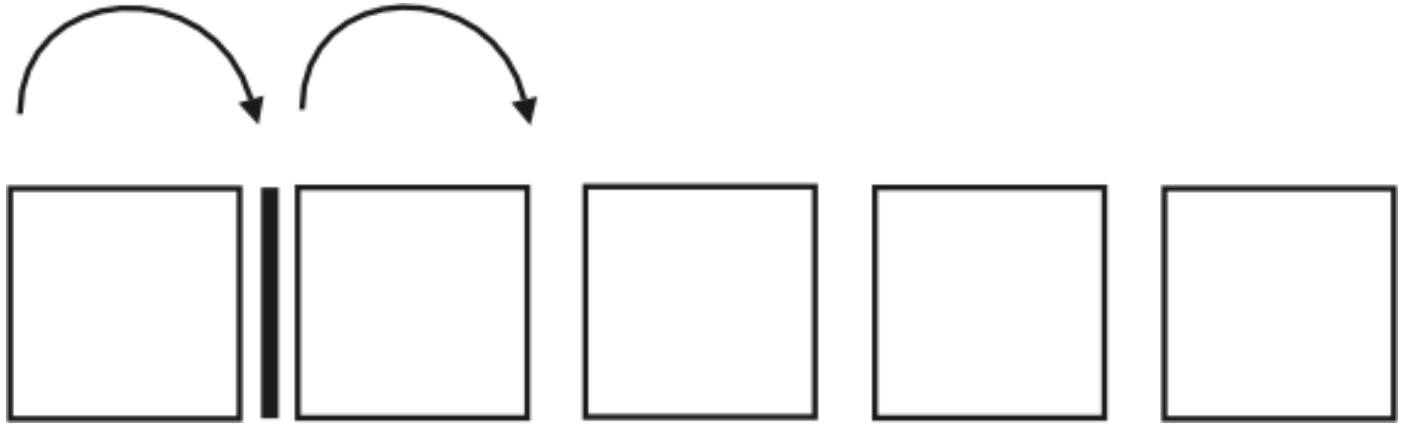

Figura 9: Visione astratta di una lista concatenata

- Un tipo di dati astratto definisce le operazioni fondamentali sui dati ma non ne specifica una implementazione

Tipi di dati astratti e concreti

- Ci sono due modi per considerare un vettore o lista sequenziale (*array list*)
- Un vettore ha una realizzazione concreta: un array di riferimenti a oggetti riempito solo in parte
- Quando si usa un vettore non si pensa alla sua realizzazione concreta, se ne considera l'astrazione: un vettore è una sequenza ordinata di dati, a ciascuno dei quali si può accedere mediante un numero intero

Visione concreta di un vettore

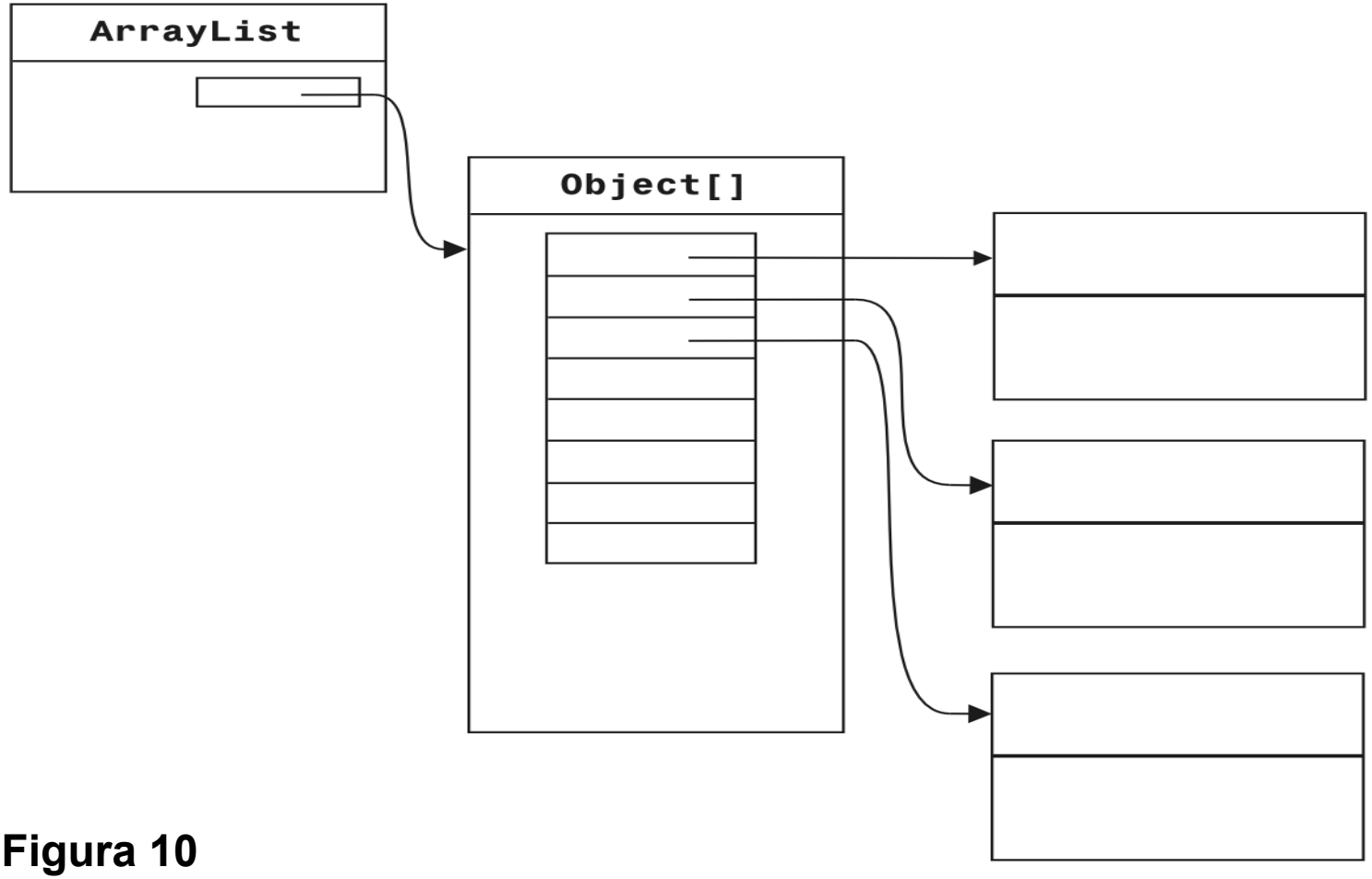

Figura 10

Visione astratta di un vettore

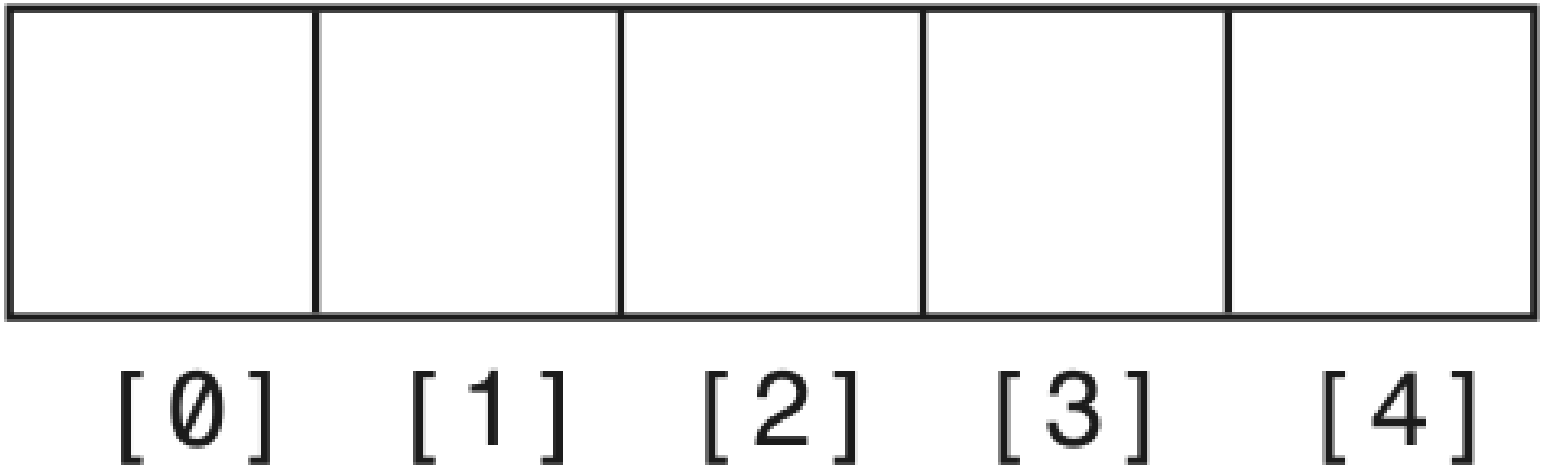

Figura 11

Tipi di dati astratti e concreti

- La realizzazione concreta di una lista concatenata e di un vettore sono abbastanza diverse
- Le loro astrazioni, di converso, sembrano essere a prima vista simili
- Per apprezzarne le differenze, consideriamo le loro interfacce pubbliche, ridotte a ciò che è essenziale

Operazioni fondamentali di un vettore

```
public class ArrayList
   {
      public Object get(int index) { . . . }
      public void set(int index, Object element) { . . . }
      . . .
   }
```

Operazioni fondamentali di una lista concatenata

```
public class LinkedList
{
   public ListIterator listIterator() { . . . }
   . . .
}

public interface ListIterator
{
   Object next();
   boolean hasNext();
   void add(Object element);
   void remove();
   void set(Object element);
   . . .
}
```

Tipi di dati astratti e concreti

- La classe ArrayList combina le interfacce di un array e di una lista
- Sia la classe `ArrayList` sia la classe `LinkedList` realizzano un'interfaccia chiamata `List` che definisce operazioni sia per l'accesso casuale sia per l'accesso sequenziale
- Questa terminologia non è comunemente utilizzata al di fuori della libreria Java
- La libreria Java fornisce le realizzazioni concrete `ArrayList` e `LinkedList` per questi tipi astratti
- Gli array in Java sono un'altra realizzazione del tipo astratto vettore

Efficienza dei vettori e delle liste

- In una lista concatenata
  - è possibile aggiungere o rimuovere un elemento
  - è necessario modificare un numero fisso di riferimenti per aggiungere o rimuovere un nodo, indipendentemente dalla dimensione della lista
  - Usando la notazione O-grande
- Aggiungere o rimuovere un elemento qualsiasi in un vettore
  - richiede un tempo $O(n)$
  - in media devono essere spostati $n/2$ elementi

Efficienza delle operazioni per vettori e liste

Operazione	Vettore	Lista
Accesso casuale	$O(1)$	$O(n)$
Passo di attraversamento lineare	$O(1)$	$O(1)$
Aggiunta/rimozione di un elemento	$O(n)$	$O(1)$

Tipi di dati astratti

- Una lista
- E’ una sequenza ordinata di elementi che può essere scandita con accesso sequenziale
- Consente l’inserimento e la rimozione di elementi in qualsiasi posizione
- Un vettore
- è una sequenza ordinata di elementi che consente l’accesso casuale mediante un indice intero

Pile e code

- Una pila è una raccolta di oggetti che vengono estratti in modo “last in, first out”
- Una coda è una raccolta di oggetti che vengono estratti in modo “first in, first out”

Pile

- Una *pila* (*stack*) consente l'inserimento e la rimozione di elementi a una sola estremità, che viene tradizionalmente chiamata la *cima* (*top*) della pila.
- Nuovi oggetti vengono inseriti in cima alla pila
- Oggetti possono essere rimossi dalla cima della pila
- Gli oggetti vengono rimossi in ordine inverso cioè "l'ultimo inserito è il primo a essere estratto" ("*last in, first out*", LIFO)
- Le operazioni di inserimento e di estrazione vengono tradizionalmente chiamate `push` e `pop`

Code

- Una *coda* (*queue*) è simile a una pila ma si inseriscono elementi a un'estremità della coda (la "fine", *tail*) e li si rimuovono dall'altra estremità (la "testa", *head*)
- Le code conservano gli elementi in modo che "il primo entrato sia il primo a uscire" ("*first in, first out*", FIFO)
- Gli oggetti vengono quindi rimossi nello stesso ordine in cui sono stati inseriti

Pile e code nell'informatica

- Code
  - Il sistema che in Java gestisce l'interfaccia utente grafica conserva in una coda tutti gli eventi
  - Un altro esempio è una coda di stampa
- Pile
  - *la pila di esecuzione* (*run-time stack*) gestita da un processore o da una macchina virtuale per memorizzare le variabili di metodi annidati

La classe `Stack`

- Nella libreria Java è presente una classe `Stack` che realizza il tipo astratto pila

```
Stack<String> s = new Stack<String>();
s.push("A");
s.push("B");
s.push("C");
// Il ciclo seguente stampa C, B, e A
while (s.size() > 0)
   System.out.println(s.pop());
```

La classe `Queue`

- Le realizzazioni di coda presenti nella libreria standard di Java sono state progettate per un utilizzo in programmi `multithreaded`
- Ma è semplice realizzarne una

Le realizzazioni di coda

```
public class LinkedListQueue
{
   /**
      Costruisce una coda vuota che usa una lista concatenata.
   */
   public LinkedListQueue()
   {
      list = new LinkedList();
   }
   /**
      Aggiunge un oggetto alla fine della coda.
      @param x l'oggetto da aggiungere
   */
   public void add(Object x)
   {
      list.addLast(x);
```

Continua…

Le realizzazioni di coda

```
}

/**
    Rimuove un oggetto dalla testa della coda.
   @return l'oggetto rimosso
*/
   public Object remove()
   {
      return list.removeFirst();
   }

   /**
      Restituisce il numero di oggetti presenti nella coda.
      @return la dimensione
   */
   int size()
   {
      return list.size();
   }
   private LinkedList list;
}
```