Scope e visibilità per classi

- **Packages**
- **Classi interne nelle loro diverse forme**
  - Interne / statiche / locali
  - Utilizzo congiunto con interfacce
  - Implementazione di iteratori
  - Gestione di eventi

Packages

- **Package: insieme di classi e interfacce in relazione**
- **Per formare un package basta inserire la direttiva**

```
package packageName;
```

  come prima istruzione nel file sorgente

- **Una sola direttiva per file**
- **Classi contenute in file che non dichiarano packages vengono incluse in un package “anonimo”**
  - package anonimo OK solo per micro applicazioni, o in fase di sviluppo

Continua…

Packages

Package	Finalità	Classe Tipica
java.lang	Supporto al linguaggio	Math, String
java.util	Utilities	Random
java.io	Input e Output	PrintStream
Java.awt	Abstract Window Toolkit	Color
Java.applet	Applets	Applet
Java.net	Networking	Socket
Java.sql	Accesso a database	ResultSet
Java.swing	Ingerfaccia utente Swing	JButton
…	…	…

Accesso agli elementi di un package

- Per accedere ai tipi di un package utilizziamo il nome “qualificato”

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

- Uso dei nomi qualificati verboso
- `import` permette sintesi

```
import java.util.Scanner;
. . .
Scanner in = new Scanner(System.in)
```

Import

- di una classe

```
import java.util.Scanner;
. . .
Scanner in = new Scanner(System.in)
```

- di tutte le classi di un package

```
import java.util.*;
```

Continua…

Import

- **Packages non formano gerarchie**

```
// import dei tipi di java.awt.color
import java.awt.color.*;
// import dei tipi di java.awt (non del package color!)
import java.awt.*;
```

- **Static import**
  - delle costanti e metodi statici dei tipi di un package

```
import static java.lang.Math.PI
import static java.lang.Math.*;.
```

Nomi di package

- **Packages utili anche come "*namespaces*" per evitare conflitti di nomi (per classi/interfacce)**
- **Esempio, Java ha due classi `Timer`**

```
java.util.Timer vs. javax.swing.Timer
```

- **Nomi di package devono essere univoci**
  - Convenzione: utilizziamo come prefissi domini internet, oppure indirizzi e-mail (in ordine inverso)

```
it.unive.dais
it.unive.dais.po
```

Continua…

Localizzazione di package

- Nomi di package devono essere consistenti con i path della directory che li contengono

```
it.unive.dais.po.banking
```

- Deve essere contenuto in un folder/directory localizzato nel path corrispondente

```
UNIX: <base directory>/it/unive/dais/po/banking
```

```
WINDOWS: <base directory>\it\unive\dais\po\banking
```

Continua…

Localizzazione di package

- **CLASSPATH: definisce le directory base dove localizzare i packages**
- **Spesso utili due directory base**
  - per file sorgenti (.java)
  - per file compilati (.class)

```
UNIX:
export CLASSPATH=/home/po/java/src:/home/po/java/classes:.

WINDOWS:
set CLASSPATH=c:\home\po\java\src;\home\po\java\classes;.
```

Classi interne

- Una classe può essere dichiarata all'interno di una classe, come i campi ed i metodi
- E' a tutti gli effetti un elemento della classe che la racchiude (come campi e metodi).
- E' riferibile da qualunque punto (e metodo) della classe in cui è dichiarata
- Può essere dichiarata
  - `private` / `public / protected`
  - `static / non static`

Classi interne

- **Se `private` , una classe interna è visibile solo all'interno della classe che la dichiara**
- **La classe interna ha accesso a tutti i campi della classe che la contiene**
  - se **`static`** solo ai campi **`static`**

Classi interne: static / non static

- **Vale sempre la stessa regola**
  - Definiamo static tutte le classi interni non hanno necessità di riferire campi (non static) della classe che le include
- **Ricordiamo**
  - static inner classes sono più efficienti
  - possiamo comunque ottenere l'accesso ai campi della classe esterna passandoli esplicitamente nel costruttore (in modo da poterli riferire nella classe interna).

Classi locali

- **Classi che hanno interesse locale, possono essere dichiarate all'interno di un metodo**
- **Se una classe è dichiarata localmente di un metodo può essere riferita solo nel corpo del metodo**

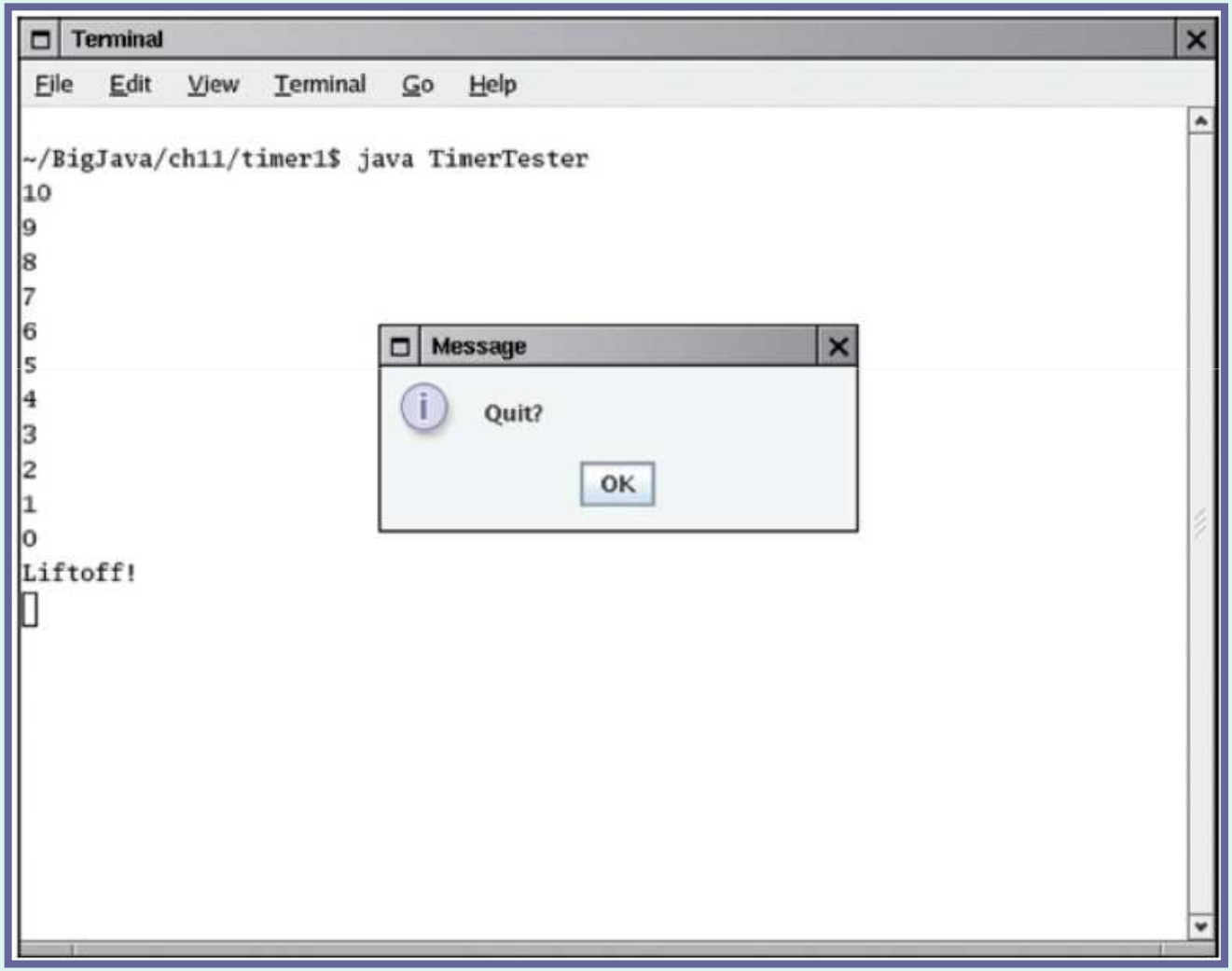
Continua…

Classi locali: accesso a variabili esterne

- **Da una classe locale possiamo riferire**
  - campi e metodi della classe esterna
  - variabili locali final del metodo che la dichiara

- **Da una classe locale a metodi static possiamo riferire**
  - campi e metodi static della classe esterna
  - variabili locali final del metodo che la dichiara

Esempio: timer *countdown*



CountDownListener

```
class CountDownListener implements ActionListener
{
   public CountDownListener(int initialCount)
   {
       count = initialCount;
   }

   public void actionPerformed(ActionEvent event)
   {
       if (count >= 0) System.out.println(count);
       if (count == 0) System.out.println("Liftoff!");
       count--;
   }
   private int count;
}
```

CountDownListener

- **Soluzione con classi top-level**

```
public class CountDownTester
{
   public static void main(String[] args)
   {
     CountDownListener listener = new CountDownListener(10);
     // Millisecondi tra due tick
     final int DELAY = 1000;
     Timer t = new Timer(DELAY, listener);
     t.start();

     JOptionPane.showMessageDialog(null, "Quit?");
     System.exit(0);
   }
}
```

CountDownListener

- Con una classe interna

```
class CountDownListener implements ActionListener
{
   . . .
}
public class CountDownTester

   public static void main(String[] args)
   {
     CountDownListener listener = new CountDownListener(10);
     Timer t = new Timer(DELAY, listener);
     . . .
   }
}
```

Continua…

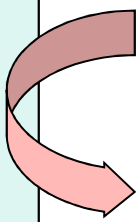
CountDownListener

- Con una classe interna (anche static?)

```
public class CountDownTester
{
   class CountDownListener implements ActionListener
   {
      . . .
   }
   public static void main(String[] args)
   {
     CountDownListener listener = new CountDownListener(10);
     Timer t = new Timer(DELAY, listener);
     . . .
   }
}
```

CountDownListener

- Con una classe locale



```
public class CountDownTester
{
   class CountDownListener implements ActionListener
   {
      . . .
   }
   public static void main(String[] args)
   {
     CountDownListener listener = new CountDownListener(10);
     Timer t = new Timer(DELAY, listener);
     . . .
   }
}
```

CountDownListener

- Con una classe locale

```
public class CountDownTester
{
   public static void main(String[] args)
   {
     class CountDownListener implements ActionListener
     {
        . . .
     }
     CountDownListener listener = new CountDownListener(10);
     Timer t = new Timer(DELAY, listener);
     . . .
   }
}
```

Classi interne/locali anonime

- **In molti casi le classi interne sono utilizzate per creare una sola istanza di una interfaccia**
- **Possono essere rese anonime … se**
  - Implementano una interfaccia
  - Hanno un default constructor

Continua…

Classi interne/locali

```
public class CountDownTester
{
   public static void main(String[] args)
   {
     CountDownListener listener = new CountDownListener(10);
     class CountDownListener implements ActionListener
     {
      public CountDownListener(int initialCount)  { . . .  }
      public void actionPerformed(ActionEvent event){. . . }
      private int count = 10;
     }
     . . .
     Timer t = new Timer(DELAY, listener);
     t.start();
     . . .
    }
}
```

Rendiamo anonimizzabile

Classi interne/locali

```
public class CountDownTester
{
   public static void main(String[] args)
   {
     CountDownListener listener = new CountDownListener();
     class CountDownListener implements ActionListener
     {
      public void actionPerformed(ActionEvent event){. . . }
      private int count = 10;
     }
     . . .
     Timer t = new Timer(DELAY, listener);
     t.start();
     . . .
    }
}
```

Classi interne/locali anonime

```
public class CountDownTester
{
   public static void main(String[] args)
   {
     CountDownListener listener = new CountDownListener(10);
     class CountDownListener implements ActionListener()
     {
       public void actionPerformed(ActionEvent event){. . . }
      private int count = 10;
     }
     . . .
     Timer t = new Timer(DELAY, listener);
     t.start();
     . . .
    }
}
```

Classi interne/locali anonime

```
public class CountDownTester
{
   public static void main(String[] args)
   {
     ActionListener listener = new ActionListener()
     {
      public void actionPerformed(ActionEvent event){. . . }
      private int count = 10;
     }
     . . .
     Timer t = new Timer(DELAY, listener);
     t.start();
     . . .
    }
}
```

Classi interne/locali anonime

- **Definite nel momento della loro istanzazione**
- **Hanno una sola istanza**
- **Non hanno qualificatori di accesso**
- **Implementano una interfaccia (o estendono una classe)**
  - quella menzionata nel costrutto new che le definisce
- **… (cf. java.sun.com/doc)**

Domanda

- **Con quale altro meccanismo potremmo garantire ad un metodi di una classe interna o locale l’accesso ad una variabile esterna?**

Risposta

- **L’alternativa è passare la variabile esterna come parametro per il costruttore della classe interna/locale, o per il metodo che deve accedere la variabile**

Domanda

- **Perché le variabili locali di un metodo devono essere final se acceduta da una classe locale?**

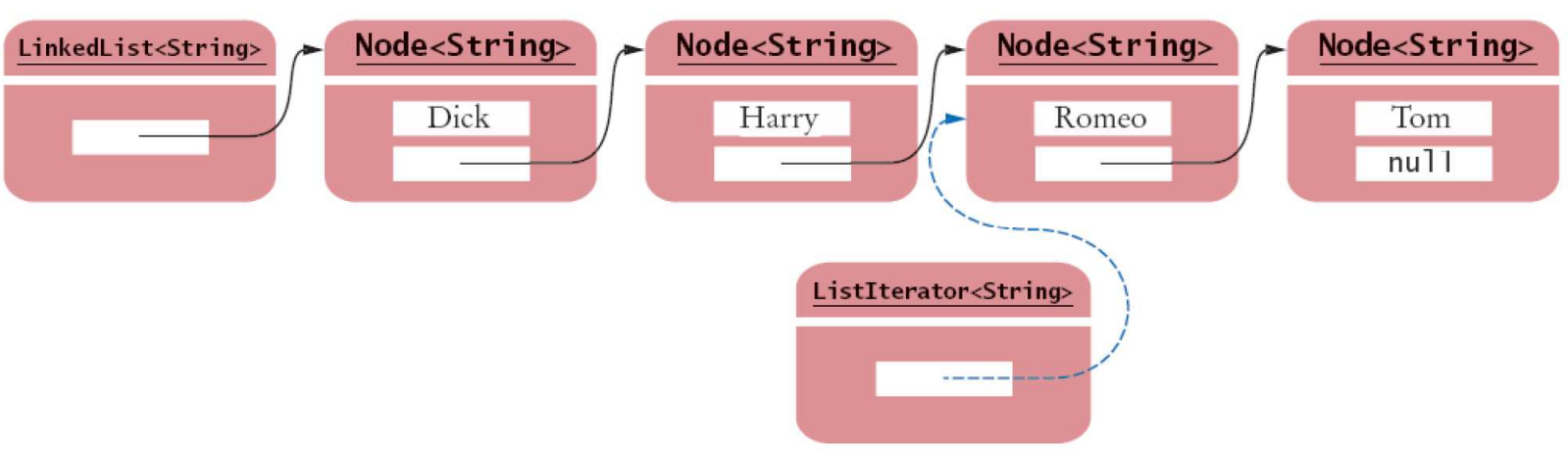
Risposta

- **Perché le variabili locali del metodo sono deallocate al termine al ritorno dal metodo, mentre le istanze della classe interna rimangono attive**

  Per evitare riferimenti pendenti (*dangling references*), il valore della locale viene copiato nella classe interna (e dunque modifiche alla locale non verrebbero percepite dalla classe interna)

Esempio / `LinkedLists`

- **Implementazione standard, con elementi collegati tra loro**
- **La struttura interna è mascherata**
  - gli elementi sono visitabili utilizzando un iteratore



Esempio / `LinkedLists`

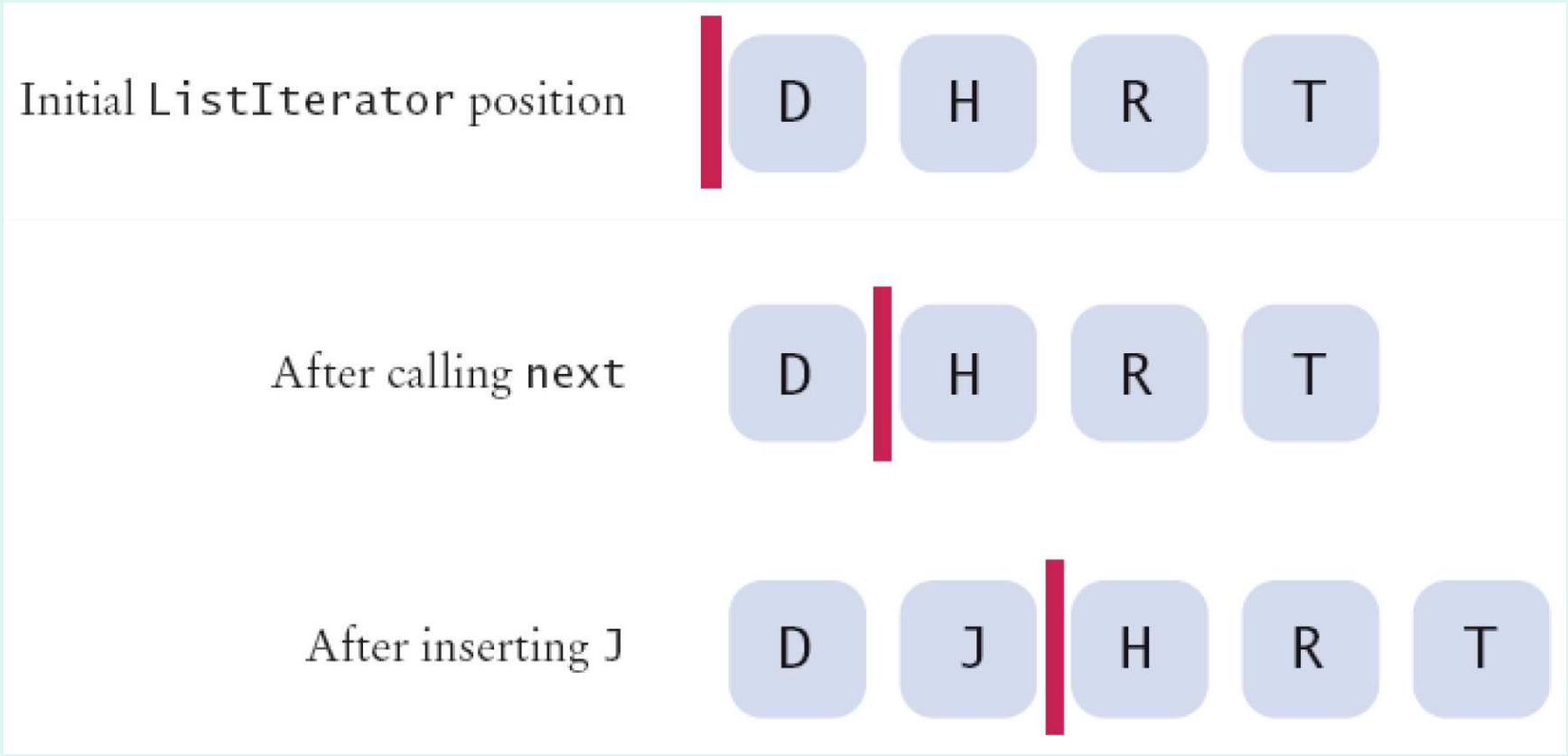
```
public class LinkedList<E> implements List<E>
{
   private Node<E> first;
   . . .

   public ListIterator<E> mkIterator()
   {
      . . .
   }

   . . .
}
```

`ListIterator<E>`

- **Fornisce accesso agli elementi contenuti in una `LinkedList<E>`**
- **Tiene traccia di una posizione all'interno della lista**
- **Maschera la struttura interna e allo stesso tempo permette l'accesso**

Visione Concettuale

- **Un cursore che scorre gli elementi della lista**



`ListIterator<E>`

- **Il metodo `mkIterator` associa l'iteratore alla lista**
  - posiziona l'iteratore sul primo elemento della lista
- **Il metodo `next()` sposta il cursore sul prossimo elemento**
  - restituisce l'elemento puntato prima del movimento
  - **`NoSuchElementException`** se non c'è un prossimo elemento
- **Il metodo `hasNext()` restituisce true se esiste un prossimo elemento**

ListIterator<E>

```
LinkedList<String> mailbox = . . .;
ListIterator<String> iterator = mailbox.mkIterator();

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

- **NB: il metodo `next()` restituisce l'elemento a cui l'iteratore punta prima dello spostamento**

`ListIterator<E>`

- **Abbreviazione:**

```
for (String email: mailbox)
{
   Do something with email
}
```

Dietro le quinte, il `for` loop usa un iteratore per visitare tutti gli elementi della lista

ListIterator<E>

```
interface ListIterator<E> extends Iterator<E>
{
   // restituisce il prossimo elemento
   // NoSuchElementException se non esiste prossimo
   E next();

   // true se esiste un prossimo elemento
   public boolean hasNext();

   // aggiunge un elemento prima di next: opzionale
   // UnsupportedException se non implementata
   void add(E e);

   . . .
}
```

`LinkedListIterator<E>`

- **Implementa una versione semplificata dell'interfaccia `ListIterator<E>`**
- **Realizzata come classe interna della classe LinkedList**
- **Ha accesso al campo first della lista ed alla classe interna Nodo**
- **Clienti di `LinkedList` non conoscono il nome della classe che viene mascherata dall'interfaccia**

LinkedListIterator<E>

```
// classe interna a LinkedList<E>

class LinkedListIterator<E> implements ListIterator<E>
{
   private E current = first; // first della classe esterna

   public E next()
   {
      if (!hasNext()) thrown new NoSuchElementException();
      E result = current.info;
      current = current.next;
      return result;
   }

   public boolean hasNext()
   {
       return current != null;
   }
}
```

LinkedList<E>

```
class LinkedList<E> implements List<E>
{
   private Node { E info; Node next; }

   private class LinkedListIterator<E>
                 implements ListIterator<E>
   {
      // vedi slide precedente
   }

   public ListIterator<E> mkIterator()
   {
       return new LinkedListIterator<E>();
   }

   . . .
}
```