Marco Faella

Iteratori, teoria e pratica

Lezione n. 7

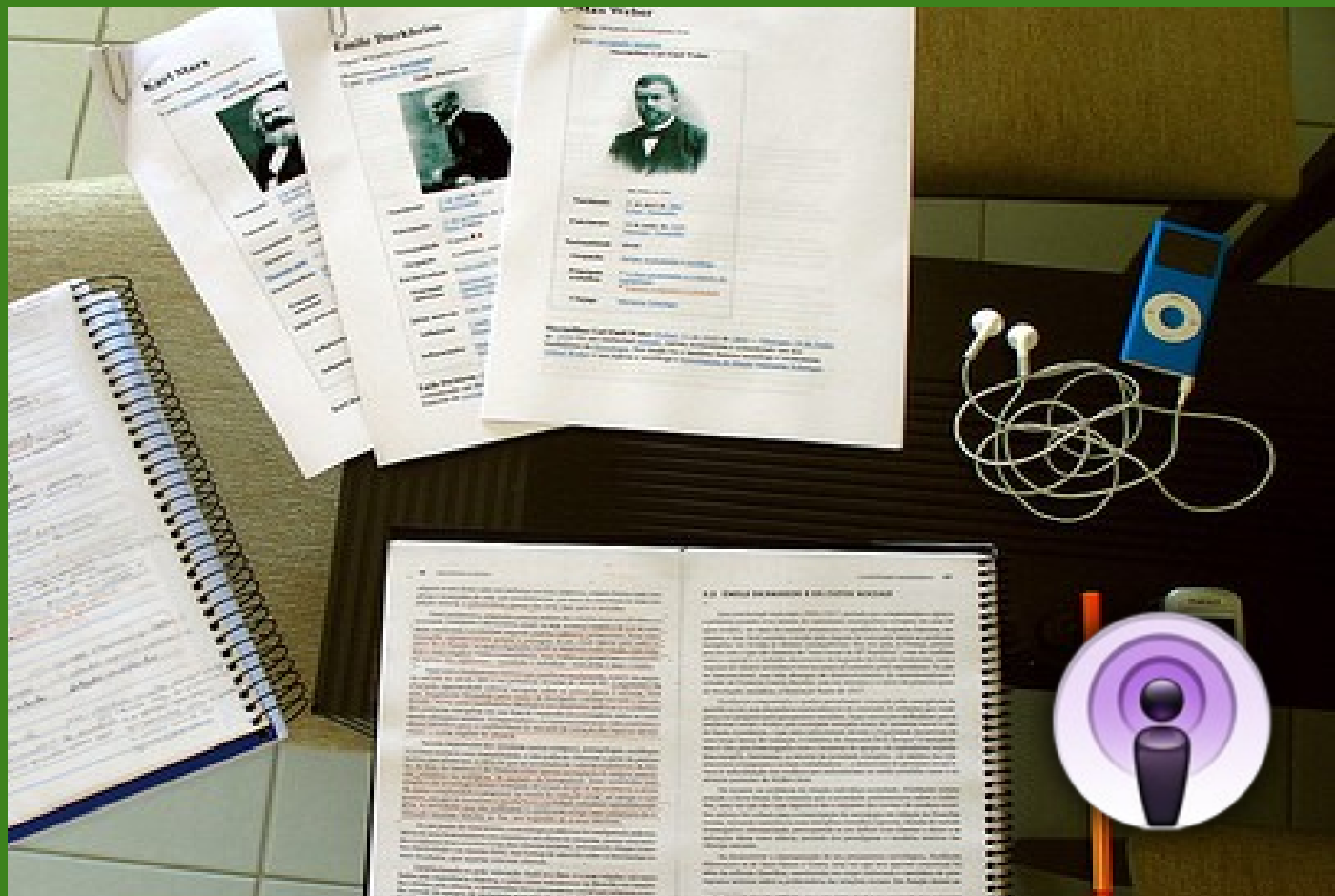
Parole chiave:
Java

Corso di Laurea:
Informatica

Insegnamento:
Linguaggi di Programmazione II

Email Docente:
faella.didattica@gmail.com

A.A. 2009-2010



- In questa lezione sarà illustrato il primo “design pattern” del corso
- I design pattern (letteralmente, schemi di progettazione) presentano soluzioni standard per situazioni tipiche di progettazione orientata agli oggetti
- Nell'informatica, l'uso di questo termine, e l'individuazione dei primi pattern si devono al testo “Design Patterns”, di Gamma, Helm, Johnson e Vlissides (la cosiddetta *gang of four*), del 1994

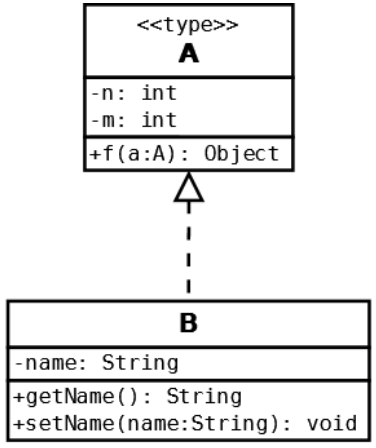
- Ciascun pattern è diviso in 4 parti:
  - un **nome**, breve e significativo
  - la descrizione del **contesto** cui il pattern si applica
  - la descrizione della **soluzione** che il pattern suggerisce
  - la descrizione delle **conseguenze** che l'applicazione del pattern comporta

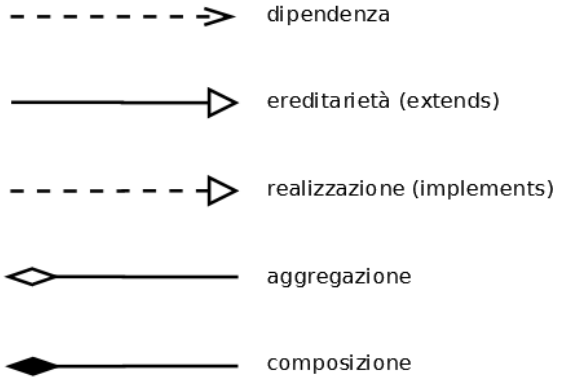
- In queste slide, se un pattern prevede l'uso di più classi o interfacce in relazione tra loro, presenteremo anche il diagramma UML delle classi che illustra queste relazioni

Il diagramma delle classi UML

- Ricordiamo brevemente i principali elementi di un diagramma delle classi UML
- Le classi sono rappresentate da rettangoli
- E' possibile mostrare campi e metodi dividendo in tre parti il rettangolo
- I segni + e – indicano visibilità pubblica e privata, rispettivamente

- Ad esempio, nella figura a destra:
  - l'annotazione "type" indica che A è un'interfaccia
  - A contiene un metodo "f", che accetta un riferimento di tipo A e restituisce un Object
  - la classe B implementa l'interfaccia A
  - B contiene un campo privato di tipo stringa e due metodi pubblici
  - non è necessario elencare esplicitamente anche la presenza del metodo f in B

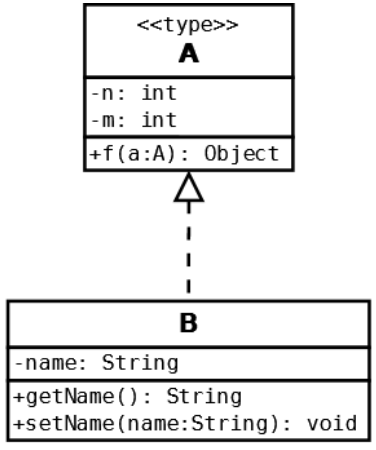


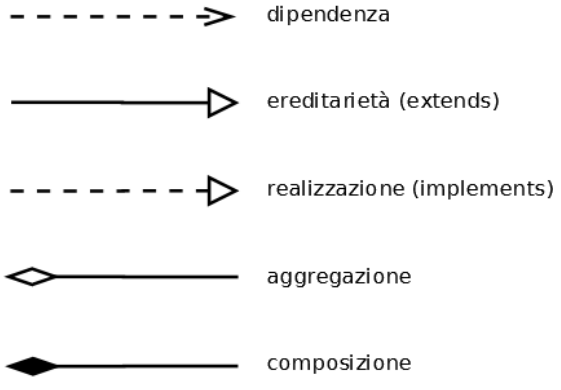


Il diagramma delle classi UML

- Le principali relazioni tra classi sono indicate da frecce con tratti distintivi, come illustrato in figura
- La **dipendenza** si può applicare ogni volta che una classe A ne utilizza in qualunque modo un'altra B
  - ad esempio, quando un metodo di A accetta un parametro di tipo B
  - oppure, quando A chiama un metodo (anche statico) di B
- La punta a triangolo vuoto viene usata per indicare la **generalizzazione** tra due classi o due interfacce (extends) e la realizzazione di un'interfaccia da parte di una classe (implements)
- La relazione di **aggregazione** sussiste quando una classe contiene riferimenti ad un'altra
- La **composizione** è una forma più forte di aggregazione
- Aggregazione e composizione sono solitamente accompagnate da una *cardinalità*

- Per approfondire, si consulti un testo sull'UML, come *"UML for Java Programmers"*, di R.C. Martin





- Il primo pattern che esaminiamo prende il nome di **Iterator**

- Questo pattern riguarda il modo di passare in rassegna gli elementi di una data collezione o insieme di oggetti
- Si suppone, cioè, che un determinato oggetto (chiamato contenitore, o aggregato) ne contenga altri
- Il contenitore vuole permettere agli utenti (client) di passare in rassegna tutti gli oggetti contenuti, senza però esporre la sua struttura interna
- La soluzione proposta consiste essenzialmente nel creare un oggetto, chiamato iteratore, che rappresenta un indice all'interno del contenitore

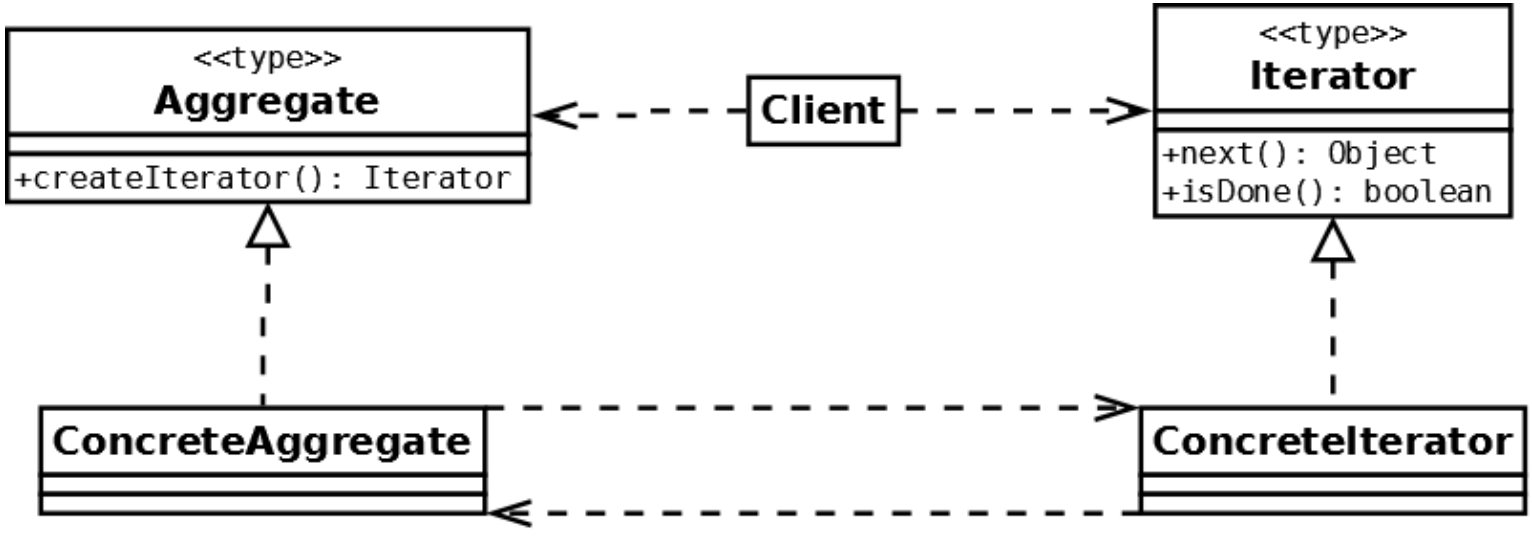
- Secondo la classificazione della gang of four, questo pattern è di categoria **comportamentale** (*behavioral*)

- Nelle prossime slide, presentiamo il problema e la soluzione proposta, nel modo schematico tipico dei design pattern

- **Contesto**:
  1) Un oggetto (*aggregato*) contiene altri oggetti (*elementi*)
  2) I clienti devono poter accedere a tutti gli elementi, uno alla volta
  3) L'aggregato non deve esporre la sua struttura interna
  4) Più clienti devono poter accedere contemporaneamente

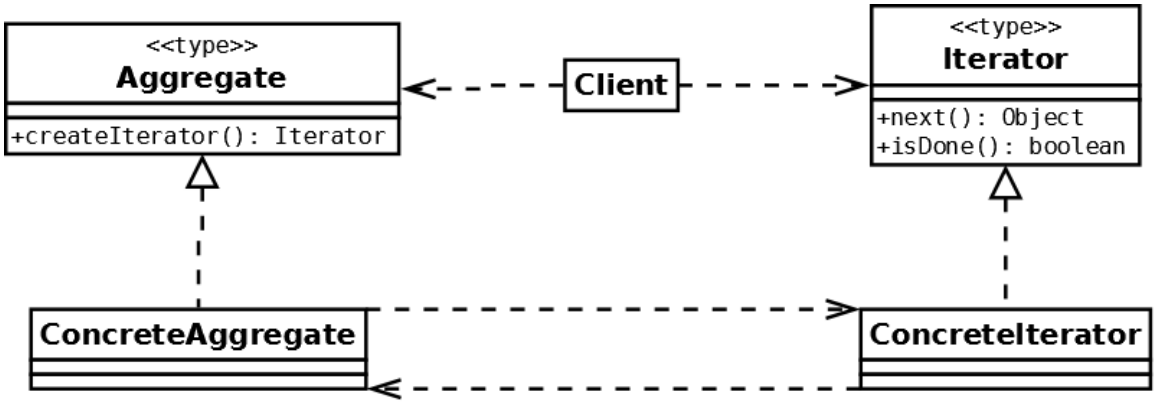
- **Soluzione**:
  1) Definire una classe *iteratore* che recupera un elemento per volta
  2) L'aggregato ha un metodo che restituisce un nuovo iteratore

La soluzione è illustrata meglio nella prossima slide, con l'ausilio di un diagramma UML


<<type>>
Aggregate
+createIterator(): Iterator
Client
<<type>>
Iterator
+next(): Object
+isDone(): boolean
ConcreteAggregate
ConcreteIterator

Il pattern ITERATOR

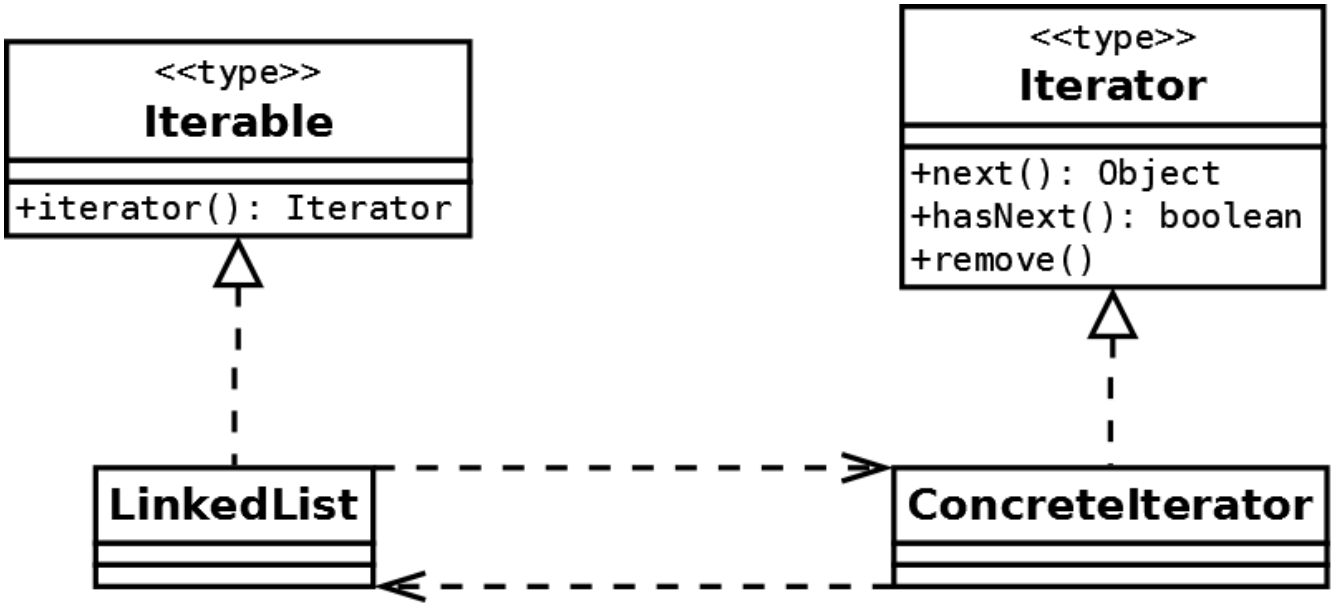
La soluzione proposta è illustrata dal diagramma UML a destra



- L'interfaccia *Iterator* rappresenta l'iteratore, cioè l'indice all'interno del contenitore
- Il suo metodo **next** restituisce il prossimo elemento del contenitore, e contemporaneamente fa avanzare l'indice di una posizione
- Il metodo **isDone** restituisce vero se tutti gli elementi del contenitore sono stati visitati
- Le chiamate a isDone non modificano la posizione corrente dell'indice

- L'interfaccia *Aggregate* rappresenta il contenitore
- Il contenitore offre un metodo createIterator che restituisce un nuovo iteratore

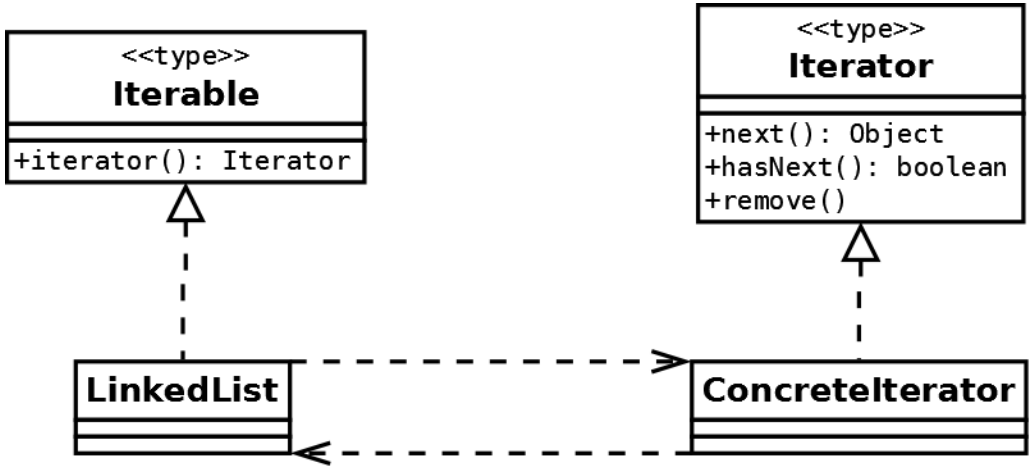
- Naturalmente, ci saranno classi concrete che implementeranno le due interfacce di sopra
- Tuttavia, il client potrebbe anche non conoscere tali classi concrete e limitarsi ad utilizzare le interfacce



- Il metodo **remove**, non previsto dal pattern, elimina dal contenitore *l'ultimo elemento che è stato restituito da next*
- remove può essere chiamato una sola volta dopo ogni chiamata a next
- remove è un'operazione facoltativa: gli iteratori concreti sono liberi di non supportarla
  - in questo caso, remove deve lanciare l'eccezione (non verificata) *UnsupportedOperationException*

La figura a destra illustra come è stato implementato il pattern ITERATOR nella libreria standard Java

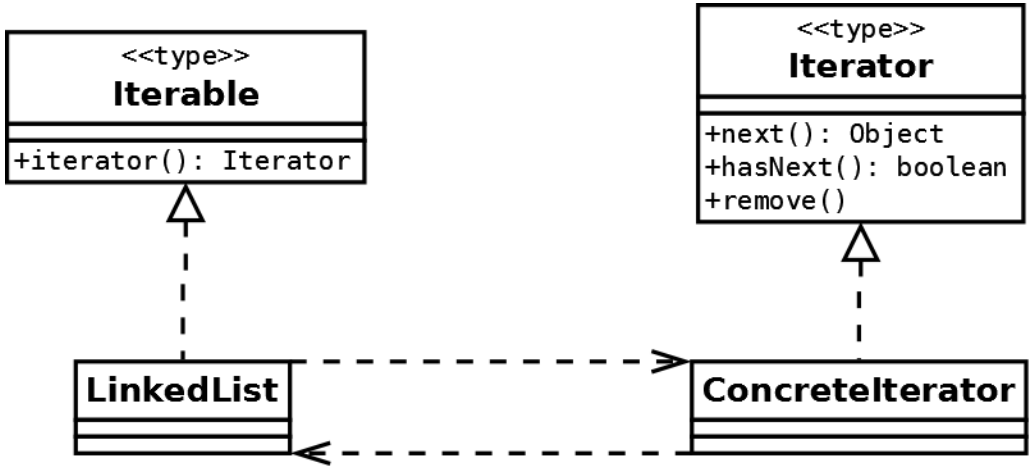
- Come nel pattern, l'interfaccia Iterator rappresenta l'iteratore
- Il suo metodo **next** si comporta come previsto dal pattern
  - se viene chiamato quando non ci sono più elementi da visitare, next deve lanciare l'eccezione *NoSuchElementException*
- Il metodo **hasNext** è l'opposto di isDone
  - restituisce vero se c'è un almeno un altro elemento del contenitore che deve ancora essere visitato



Iteratori in Java (2)

- L'interfaccia *Iterable* fa le veci di *Aggregate*
- Il metodo iterator si comporta come createIterator, restituendo un nuovo iteratore

- Un esempio di classe concreta che implementa Iterable è **LinkedList**, che rappresenta una lista concatenata
- La slide successiva introduce brevemente questa classe



- Questa descrizione fa riferimento alla versione *non parametrica* di queste interfacce, come si trovava in Java 1.4
- A partire da Java 1.5, queste interfacce, e le classi corrispondenti, sono state aggiornate per sfruttare la programmazione parametrica
- Lezioni successive tratteranno di queste modifiche

- Le interfacce Iterable e Iterator sono contenute nel package *java.util*

Liste concatenate (grezze)

- La classe java.util.LinkedList rappresenta una **lista doppiamente concatenata**
- Essa appartiene alla *Java Collection Framework*, una parte della API Java che sarà oggetto di lezioni successive
- Qui, presentiamo brevemente i suoi metodi principali
- Per cominciare, LinkedList implementa Iterable, e quindi dispone di un metodo "iterator" che restituisce un iteratore

public boolean **add**(Object x)	Aggiunge x in coda alla lista e restituisce true Il motivo per cui restituisce un valore verrà chiarito quando si introdurranno le interfacce List, Collection e Set
public boolean **contains**(Object x)	Restituisce true se la lista contiene un oggetto y tale che x.equals(y) è vero Ha complessità di tempo proporzionale alla lunghezza della lista
public boolean **remove**(Object x)	Rimuove il primo oggetto della lista uguale a x Restituisce vero se ha trovato e rimosso l'oggetto
public int **size**()	Restituisce la dimensione della lista
public void **addFirst**(Object x)	Aggiunge x in testa alla lista
public void **addLast**(Object x)	Equivalente ad add(x), ma senza valore restituito
public Object **removeFirst**()	Rimuove e restituisce la testa della lista Solleva *NoSuchElementException* se la lista è vuota
public Object **removeLast**()	Rimuove e restituisce la coda della lista Solleva *NoSuchElementException* se la lista è vuota

Iteratori in Java (3)

- L'esempio seguente mostra l'uso tipico di un iteratore su una lista

```
LinkedList l = new LinkedList();
l.add(“Hello ”);
l.add(“world!”);

Iterator i = l.iterator();
while (i.hasNext()) {
   String s = (String) i.next();
   System.out.print(s);
}
```

- L'output dell'esempio è la stringa “Hello world!”
- Si noti il cast a String, necessario perché il tipo restituito da next è Object
  - questo cast viene reso superfluo dalla versione parametrica di Iterator, che verrà presentata più avanti nel corso

Esercizio (esame 27/4/2006, #3)

- Il seguente frammento di classe definisce un nodo in un albero binario

```
public class BinaryTreeNode {
      private Object value;
      private BinaryTreeNode left, right;
      public BinaryTreeNode getLeft()   { return left; }
      public BinaryTreeNode getRight()  { return right; }
      // aggiungere costruttore, metodo toString, etc.
}
```

- Si implementi una classe iteratore BinaryTreePreIterator che visiti i nodi dell'albero in **preorder** (ciascun nodo prima dei suoi figli). Tale classe deve poter essere usata nel seguente modo:

```
public static void main(String[] args) {
  BinaryTreeNode root = ...;
  Iterator i = new BinaryTreePreIterator(root);
  while (i.hasNext()) {
    BinaryTreeNode node = (BinaryTreeNode) i.next();
    ...
  }
}
```

Versioni parametriche

Liste concatenate (con parametro di tipo)

- La classe java.util.LinkedList rappresenta una **lista doppiamente concatenata**
- Essa appartiene alla *Java Collection Framework*, una parte della API Java che sarà oggetto di lezioni successive
- Qui, presentiamo brevemente i suoi metodi principali
- Per cominciare, LinkedList implementa Iterable, e quindi dispone di un metodo "iterator" che restituisce un iteratore

public boolean **add**(T x)	Aggiunge x in coda alla lista e restituisce true Il motivo per cui restituisce un valore verrà chiarito quando si introdurranno le interfacce List, Collection e Set
public boolean **contains**(Object x)	Restituisce true se la lista contiene un oggetto y tale che x.equals(y) è vero Ha complessità di tempo proporzionale alla lunghezza della lista
public boolean **remove**(Object x)	Rimuove il primo oggetto della lista uguale a x Restituisce vero se ha trovato e rimosso l'oggetto
public int **size**()	Restituisce la dimensione della lista
public void **addFirst**(T x)	Aggiunge x in testa alla lista
public void **addLast**(T x)	Equivalente ad add(x), ma senza valore restituito
public T **removeFirst**()	Rimuove e restituisce la testa della lista Solleva *NoSuchElementException* se la lista è vuota
public T **removeLast**()	Rimuove e restituisce la coda della lista Solleva *NoSuchElementException* se la lista è vuota

Esercizio (esame 27/4/2006, #3)

- Il seguente frammento di classe definisce un nodo in un albero binario

```
public class BinaryTreeNode {
      private Object value;
      private BinaryTreeNode left, right;
      public BinaryTreeNode getLeft()   { return left; }
      public BinaryTreeNode getRight()  { return right; }
      // aggiungere costruttore, metodo toString, etc.
}
```

- Si implementi una classe iteratore BinaryTreePreIterator che visiti i nodi dell'albero in **preorder** (ciascun nodo prima dei suoi figli). Tale classe deve poter essere usata nel seguente modo:

```
public static void main(String[] args) {
  BinaryTreeNode root = new BinaryTreeNode(“ciao”,
                                           new BinaryTreeNode(“pippo”,NULL,NULL),
                                           new BinaryTreeNode(“pluto”,NULL,NULL));

  Iterator<BinaryTreeNode> i = new BinaryTreePreIterator(root);
  while (i.hasNext()) {
    BinaryTreeNode node = i.next(); // senza cast
    ...
  }
}
```