

Come si realizzano le liste concatenate in Java e più in generale nella programmazione a oggetti: problemi e soluzioni.

Soluzione adottata dalla Sun: la classe predefinita LinkedList.

#### NOTA BENE:

Sviluppiamo lo studio delle liste prendendo come esempio le liste di stringhe, ma le considerazioni svolte valgono ovviamente per liste di elementi di qualsiasi tipo.

### Lista concatenata (di stringhe) in Java: realizzazione semplice con due classi separate

- una classe che realizza il **nodo**, con due campi: il dato e il nodo successivo, senza metodi;
- una classe distinta per la **lista**, con un campo di tipo nodo che contiene il riferimento al primo nodo, e con metodi per inserimento e cancellazione in testa, ricerca, ecc.;
- affinché i metodi della **classe-lista** possano accedere e manipolare i nodi, i campi della **classe-nodo** devono essere non privati.

### Primo tentativo (errato): nodo con campi pubblici

```
class Node {  
    public String elem;  
    public Node next;  
  
    public Node(String elem, Node next) {  
        this.elem = elem;  
        this.next = next;  
    }  
  
    public Node(String elem) {  
        this(elem, null);  
    }  
}
```

### La classe che realizza la lista

```
public class StringList {  
    private Node first;  
  
    public StringList() {  
        first = null;  
    }  
  
    public void aggiungiInTesta(String elem) {  
        first = new Node(elem, first);  
    }  
  
    public String firstElem() {  
        if(first == null)  
            throw new NoSuchElementException();  
        return first.elem;  
    }  
}
```

```
public void togliInTesta() {  
    if(first == null)  
        throw new NoSuchElementException();  
    first = first.next;  
}  
  
public void write() {  
    Node node = first;  
    while(node != null) {  
        out.println(node.elem);  
        node = node.next;  
    }  
    out.println();  
}
```

## Input di lista di stringhe da file di testo

```
public static StringList read(String nomeFile)
throws IOException {
    StringList newList = new StringList();
    Scanner input = new Scanner(new File(nomeFile));
    // la new Scanner(...) può generare un'eccezione
    if(input.hasNextLine()) {
        newList.first = new Node(input.nextLine());
        Node last = newList.first;
        while(input.hasNextLine()) {
            last.next = new Node(input.nextLine());
            last = last.next;
        }
    }
    return newList;
}
```

AlgELab-05-06 - Lez.01

7

## import necessari

- `import java.util.*;`  
per usare Scanner
- `import java.io.File;`  
• `import java.io.IOException;`  
per i files e le eccezioni di IO
- `import static java.lang.System.*;`  
opzionale, per poter scrivere `out.println` invece di `System.out.println`

AlgELab-05-06 - Lez.01

8

## Invarianti di classe

Un **invariante di classe** è (l'enunciazione di) una proprietà riguardante **lo stato di un oggetto** (della classe), la quale proprietà vale:

- al termine dell'esecuzione del costruttore, quando l'oggetto viene creato;
- al termine di ogni invocazione di un metodo sull'oggetto (purché l'invocazione rispetti le precondizioni del metodo), e di ogni accesso agli eventuali campi pubblici dell'oggetto.

Un invariante di classe è quindi una proprietà che vale per ogni oggetto della classe, per tutta la sua vita.

Nel progettare una classe occorre definire bene quali sono gli invarianti significativi della classe.

AlgELab-05-06 - Lez.01

9

Problema n.1 : come mantenere l'invariante di classe in presenza di un metodo *ricerca* che restituisce un nodo.

- sugli array un metodo di *ricerca* realistico deve restituire non semplicemente un booleano, bensì l'indice dell'elemento, in modo da potervi poi accedere per modificarlo, cancellarlo, ecc. (se l'elemento non c'è, il metodo deve restituire un valore speciale, ad es. l'indice inesistente -1)
- analogamente sulle liste una ricerca realistica dovrebbe restituire non un booleano ma ...un puntatore all'elemento, cioè un (riferimento a) un nodo; se l'elemento non esiste, l'unico risultato ragionevole è `null`;
- in questo modo, però:
  - la classe `Node` deve essere `public`, perché chi usa la classe `List` può ricevere un nodo come risultato della ricerca;
  - ma allora chi usa `List` ha accesso ai campi pubblici del nodo, e può quindi modificare il campo `next` in modo arbitrario, accedere ai nodi successivi, creare liste circolari, ...
- se vogliamo che gli oggetti `List` rappresentino solo sequenze finite, per mezzo di strutture concatenate lineari senza cicli, tale proprietà invariante di classe non è garantita dalla nostra realizzazione.

AlgELab-05-06 - Lez.01

10

```
public Node ricerca(String elem) {
    Node node = first;
    while(node != null && !node.elem.equals(elem))
        node = node.next;
    return node;
}
```

Nota Bene: se l'elemento non compare nella lista (incluso il caso in cui la lista è vuota), il metodo restituisce `null`, come richiesto.

AlgELab-05-06 - Lez.01

11

## Prova della classe-lista

```
public class UsaStringList {
    public static void main(String[] a) throws Exception {
        StringList myList = StringList.read("nomi.txt");
        myList.write();
        Node node = myList.ricerca("Ida");
        node.elem = "Abelardo"; modifica l'elemento trovato: OK!

        node.next.next.elem = "Eloisa";
        modifica nodo diverso da quello trovato:
        forse non è molto elegante, ma non viola l'invariante della classe

        node.next.next = node;
        crea lista circolare: viola l'invariante della classe!

        myList.write();
    }
}
```

AlgELab-05-06 - Lez.01

12

## Un tentativo di soluzione: il package

- si mettono le classi in una cartella di nome **liste** o **lists**;
- si definiscono le classi lista e nodo come appartenenti a un package avente lo stesso nome della cartella;
- la classe-lista viene mantenuta invariata;
- nella classe-nodo, invece:
  - il campo **next** viene dichiarato senza qualificatori di visibilità: in tal modo esso risulta **visibile solo all'interno del package**;
- così l'utente esterno non può più modificare la struttura della lista, mentre i metodi della classe-lista, essendo nello stesso package, possono farlo.

## Dichiarazione di package

i files nella cartella *lists* devono avere come prima riga:

```
package lists;
...
```

al di fuori del package, una classe che usa le liste:

```
import lists.*;
public class UsaStringList {
    ...
}
```

## La classe Node nel package lists

```
package lists;

public class Node {
    public String elem;
    Node next;    campo next visibile solo nel package

    public Node(String elem, Node next) {
        this.elem = elem;
        this.next = next;
    }
    ...
}
```

## La classe che usa la lista

```
import lists.*;

public class UsaStringList {
    public static void main(String[] args)
        throws Exception {
        StringList myList = StringList.read("nomi.txt");
        myList.write();
        Node node = myList.ricerca("Ida");
        // modifica di nodo diverso da quello cercato:
        // non è possibile - ERRORE !!
        node.next.next.elem = "Eloisa";
        // modifica della struttura: non è possibile: ERRORE !!
        node.next.next = node;
        ...
    }
}
```

## Problema n.2: come percorrere la lista ?

- alcuni metodi di classi utilizzatrici potrebbero dover scorrere la lista, ad es. per modificarne tutti i dati;
  - a tal fine devono poter "vedere":
    - il contenuto del campo **first** degli oggetti **List**;
    - il contenuto del campo **next** degli oggetti **Node**;
- in modo da poter scrivere un ciclo "simile" al seguente:
- ```
Node node = miaLista.primo;
while (node != null) {
    modifica node.elem;
    node = node.next;
}
```
- bisogna allora definire dei **metodi di accesso**, cioè che permettono di accedere ai campi ma non di modificarli.

## Soluzione - Prima Parte

La classe Node con metodo pubblico di accesso a next

```
package lists;

public class Node {
    public String elem;
    Node next;    campo next visibile solo nel package

    public Node(String elem, Node next) {
        this.elem = elem;
        this.next = next;
    }
    ...
    public Node next() { metodo di accesso
        return next;
    }
}
```

## Soluzione - Seconda Parte

La classe `StringList` con metodo pubblico di accesso a `first`

```
package lists;
...
public class StringList {
    private Node first;

    public StringList() {
        first = null;
    }
    ...
    public Node firstNode() {
        return first;
    }
    ...
}
```

NOTA: L'invariante di classe non può venire violato.

## Classe utilizzatrice di `StringList`

```
import liste.*;

public class UsaStringList {

    public static void main(String[] args)
        throws Exception {
        StringList myList = StringList.read("nomi.txt");
        myList.write();

        Node node = myList.firstNode();
        while(node != null) {
            modifica node.elem;
            node = node.next();
        }

        myList.write();
    }
}
```

## Ciclo for equivalente

Invece del ciclo `while`:

```
Node node = myList.firstNode();
while(node != null) {
    modifica node.elem;
    node = node.next();
}
```

si può usare un ciclo `for`:

```
for(Node nd = myList.firstNode(); nd != null; nd = nd.next())
    modifica nd.elem;
```

## Soluzione con uso di classi annidate

La visibilità differenziata realizzata nella soluzione precedente può essere ottenuta in modo più elegante ricorrendo alle classi annidate. Per mezzo di esse è infatti possibile avere contemporaneamente:

- classe `Nodo` usabile dagli utilizzatori di `StringList` per creare un cursore con cui scorrere la lista;
- campo `next` di `Node`:
  - `privato`, quindi non modificabile dagli utilizzatori;
  - ma `accessibile e modificabile` dai metodi di `StringList`, perché la classe `Node` è `annidata` dentro `StringList`;
  - accessibile ma non modificabile dagli utilizzatori, attraverso il metodo di accesso `getNext()`.

Basta porre la dichiarazione della classe `Node` *all'interno* della dichiarazione della classe `StringList`.

## Classi annidate in Java

- Campi e metodi di una classe annidata sono sempre visibili e accessibili dalla classe circostante.
- Le classi annidate possono essere soggette alle stesse restrizioni di visibilità che si usano per campi e metodi: `private`, `protected`, ecc. con significati analoghi, ad esempio:
  - classe annidata `private`: non è visibile al di fuori della classe circostante;
  - classe annidata `protected`: visibile nelle classi derivate e nel package;
  - ... ecc.
- Le classi annidate possono essere dichiarate `static` oppure no:
  - **classe annidata statica**:
    - non ha accesso a metodi e campi non statici della classe circostante;
    - può essere usata anche dai metodi statici della classe circostante;
  - **classe annidata NON statica (inner class)**:
    - ogni oggetto della classe annidata contiene un puntatore ad un oggetto della classe circostante;
    - i metodi della classe annidata hanno accesso ai campi e metodi (di istanza) di tale "oggetto circostante".

## Riassunto

Dichiarando la classe `Node` all'interno della classe `StringList`, è dunque possibile avere:

- una classe `Node` pubblica, visibile dagli utilizzatori di `StringList`;
- campo `next` (di `Node`):
  - `privato`, quindi non manipolabile dagli utilizzatori di `StringList`;
  - visibile e manipolabile dai metodi di `StringList`;
  - accessibile (ma non modificabile) dalle classi utilizzatrici, tramite il metodo `getNext()`;
- ovviamente una classe `List` pubblica, come sempre.

### NOTA

Salvo casi specifici che vedremo, le classi annidate è bene che siano `statiche`, per ragioni sia di efficienza che di semplicità di comprensione.

```

public class StringList {

    public static class Node {
        public String elem;
        private Node next;

        public Node(String elem, Node next) {
            this.elem = elem;
            this.next = next;
        }
        ...
        public Node next() {
            return next;
        }
    }

    private Node first;
    ... tutti i costruttori e metodi della classe StringList
}

```

AlgELab-05-06 - Lez.01

25

## Utilizzazione

```

import liste.StringList;

public class UsaStringList {

    public static void main(String[] args) throws ... {
        StringList myList = StringList.read("nomi.txt");
        myList.write();

        // cerco il nodo contenente l'elemento "Ida" e lo cambio in "Zoroastro"
        StringList.Node node = myList.ricerca("Ida");
        node.elem = "Zoroastro";

        // cambio tutti gli elementi della lista
        StringList.Node i = myList.firstNode();
        while(i != null) {
            i.elem = "ciao, " + i.elem;
            i = i.next();
        }
        ...
    }
}

```

AlgELab-05-06 - Lez.01

26

## Sintassi semplificata usando l'import static

```

import liste.StringList;
import static liste.StringList.*;
public class UsaStringList {

    public static void main(String[] args) throws ... {
        StringList myList = read("nomi.txt");
        myList.write();

        // cerco il nodo contenente l'elemento "Ida" e lo cambio in "Zoroastro"
        Node node = myList.ricerca("Ida");
        node.elem = "Zoroastro";

        // cambio tutti gli elementi della lista
        Node i = myList.firstNode();
        while(i != null) {
            i.elem = "ciao, " + i.elem;
            i = i.next();
        }
        ...
    }
}

```

AlgELab-05-06 - Lez.01

27

## NOTA BENE

un'istruzione come una delle seguenti:

```

// node.next() = node;
// node.next().next() = node;
ecc.

```

crea una lista circolare ?

**NO ! ERRORE DI COMPILAZIONE !**  
(spiegare chiaramente perché)

AlgELab-05-06 - Lez.01

28

## Esercizio

Definire un metodo che realizzi l'algoritmo selection-sort su oggetti della classe StringList (come definita in precedenza):

```

static void selSort(StringList lis)
    per ogni nodo i di lis, a cominciare dal primo fino all'ultimo {
        trova la posizione del minimo nella parte di lista dal nodo i alla fine;
        scambia la stringa del nodo "minimo" con quella del nodo i
    }

```

AlgELab-05-06 - Lez.01

29

## Esercizio

Definire un metodo che realizzi l'algoritmo selection-sort su oggetti della classe StringList (come definita in precedenza):

```

...
public static void selSort(StringList lis) {
    for(Node i = lis.firstNode(); i != null; i = i.next()){
        trova la posizione del minimo nella parte di lista dal nodo i alla fine:
        Node iMin = i;
        for(Node j = ...; ...; ...) {
            if(j.elem.compareTo(iMin.elem) < 0) il nuovo min è ...;
        }
        scambia la stringa del nodo "minimo" con quella del nodo i
    }
}

```

AlgELab-05-06 - Lez.01

30

## Esercizio

Definire un metodo che realizzi l'algoritmo selection-sort su oggetti della classe `StringList` (come definita in precedenza):

```
...
public static void selSort(StringList lis) {
    for(Node i = lis.firstNode(); i != null; i = i.next()) {
        Node iMin = i;
        for(Node j = i.next(); j != null; j = j.next()) {
            if(j.elem.compareTo(iMin.elem) < 0) iMin = j;
        }
        String temp = i.elem;
        i.elem = iMin.elem;
        iMin.elem = temp;
    }
}
```

## Liste di oggetti e liste di valori di tipo primitivo

- **lista di elementi di un tipo primitivo (int, double, ecc.):**
  - il nodo contiene direttamente l'elemento (nel campo-dati, ad es. il campo `elem`);
  - allora tale dato, per poter essere modificato, deve essere un campo **pubblico**;
- **lista di oggetti non modificabili (come `String`):**
  - il nodo contiene un puntatore all'oggetto, ma poiché l'unico modo per "modificare" l'oggetto è quello di sostituirlo con un oggetto diverso, il campo-dati deve essere **pubblico**;
- **lista di oggetti modificabili:**
  - il campo-dati del nodo contiene un puntatore all'oggetto;
  - per modificare l'oggetto, basta avere il valore di tale puntatore;
  - quindi il campo-dati **può non essere pubblico**, purché vi sia un metodo pubblico (ad es. `getElem()`) che lo restituisce;
  - se invece si vuole poter sostituire l'elemento-oggetto con un altro oggetto, il campo deve ancora essere pubblico.

## Esempio

```
public class ListaDipendenti {

    static public class Nodo {
        private Dipendente elemento;
        private Nodo successivo;
        ...
        public Dipendente elemento() {
            return elemento;
        }
        ...
    }
}

una classe esterna che usa ListaDipendenti():

... Nodo nodo = ricerca("Luigi Rossi");
nodo.elemento().aumentaStipendio(7.5);
```

## Problema n.3: come percorrere la listaaggiungendo e togliendo elementi

La semplice soluzione delineata nelle slides precedenti, se dotata degli opportuni metodi di **inserimento** e di **cancellazione** dopo un dato nodo, permette di percorrere la lista e, durante tale scansione, di:

- **modificare** i dati contenuti in alcuni dei nodi;
  - **inserire** dei nodi (ad es. dopo gli elementi o i nodi che soddisfano una data proprietà);
  - **togliere** dei nodi (ad es. tutti quelli che soddisfano una data proprietà)
- Nota Bene:

- i metodi di **inserimento in testa** e **cancellazione in testa**, che modificano il campo `first` di `StringList`, saranno metodi di `StringList`;
- i metodi per **inserire** un nuovo elemento **dopo un nodo dato** e per **eliminare** il nodo successivo ad un dato nodo devono invece essere metodi della classe `Node`, poiché non modificano i campi di `StringList` né dipendono in alcun modo da essi.

```
...
public class StringList {

    public static class Node {
        public String elem;
        private Node next;

        // COSTRUTTORI di Node
        public Node(String elem, Node next) {
            this.elem = elem;
            this.next = next;
        }
        ...
        // METODI di Node
        ...
    }

    // CAMPI E METODI di StringList
    ...
    ...
}
```

## Alcuni metodi di Node

```
...
public void addAfter(String elem)
aggiunge dopo se stesso (this) un nuovo nodo contenente elem;

public void removeNext()
elimina il nodo successivo a this;
```

## Alcuni metodi di StringList

```
StringList.Node firstNode()
restituisce il riferimento al primo nodo;

public void aggiungiInTesta(String elem)
aggiunge in testa un nuovo nodo contenente elem;

public void togliaInTesta()
elimina il primo nodo;
...
```

### Altri metodi di StringList

```
public Node ricerca(String elem)
restituisce il riferimento al primo nodo contenente elem,
oppure, se un tale nodo non esiste, il riferimento nullo;

public StringList copy()
costruisce e restituisce una nuova lista identica a sé (this),
con nodi tutti nuovi e quindi distinti dai nodi originali;

public void write() {
scrive sullo schermo la sequenza dei propri elementi

public static StringList read(String fileName)
legge dal file di testo di nome fileName una sequenza di stringhe
date dalle righe del testi, e costruisce e restituisce il corrispondente
oggetto della classe StringList.

...

```

### Esercizio: copia iterativa

```
public StringList copy() {
creo un nuovo oggetto StringList vuota:
StringList copia = new StringList();
se la lista this è vuota non si deve fare nulla, altrimenti è
necessario un ciclo.

```

#### INVARIANTE

**nextToCopy** è il prossimo nodo originale da copiare;  
**lastOfCopy** è l'ultimo nodo della lista copia

#### INVARIANTE

**nextToCopy** è il prossimo nodo originale da copiare;  
**lastOfCopy** è l'ultimo nodo della lista copia



#### Corpo del ciclo

```
lastOfCopy.next = new Node(nextToCopy.elem);
nextToCopy = nextToCopy.next;
lastOfCopy = lastOfCopy.next;

```

### Esercizio: copia iterativa (continua)

#### Test del ciclo

Naturalmente si continua se c'è ancora qualche nodo da copiare:

```
while (nextToCopy != null)
```

#### Inizializzazione del ciclo

Il primo nodo va creato separatamente, come al solito;

```
copia.first = new Node(first.elem);
Node nextToCopy = first.next;
Node lastOfCopy = copia.first;

```

### Esercizio: copia iterativa (fine)

```
public StringList copy() {
StringList copia = new StringList();
if(first != null) {
copia.first = new Node(first.elem);
Node nextToCopy = first.next;
Node lastOfCopy = copia.first;
while(nextToCopy != null) {
lastOfCopy.next = new Node(nextToCopy.elem);
nextToCopy = nextToCopy.next;
lastOfCopy = lastOfCopy.next;
}
}
return copia;
}

```

### Uso della classe StringList: aggiungere elementi

Voglio inserire *dopo* ogni elemento iniziante per 'E' un nuovo nodo contenente la stringa "Dopo un nome in E" :

```
proviamo:
for(Node nodo = myList.firstNode(); nodo != null; nodo = nodo.next())
if(nodo.elem.startsWith("E"))
nodo.addAfter("Dopo un nome in E");
myList.write();

```

apparentemente funziona, ma ...  
...se la stringa da aggiungere inizia anch'essa per E ?

```
for(Node nodo = myList.firstNode(); nodo != null; nodo = nodo.next())
if(nodo.elem.startsWith("E"))
nodo.addAfter("Ecco un nome in E");

```

il programma non termina (o meglio: termina per esaurimento dello heap)!

## Uso della classe StringList: aggiungere elementi

Voglio inserire *dopo* ogni elemento iniziante per 'E' un nuovo nodo contenente la stringa "Dopo un nome in E" ;  
proviamo:  

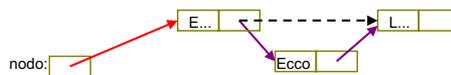
```
for(Node nodo = myList.firstNode(); nodo != null; nodo = nodo.next())
  if(nodo.elem.startsWith("E"))
    nodo.addAfter("Dopo un nome in E");
myList.write();
```

apparentemente funziona, ma ...  
...se la stringa da aggiungere inizia anch'essa per E ?

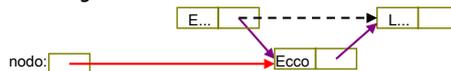
```
for(Node nodo = myList.firstNode(); nodo != null; nodo = nodo.next())
  if(nodo.elem.startsWith("E")) {
    nodo.addAfter("Ecco un nome in E");
    nodo = nodo.next(); // nodo.next è sicuramente non nullo
  }
}
il programma non termina (o meglio: termina per esaurimento dello heap)
```

## Osserva:

- nel primo caso il **nodo corrente**, dopo l'inserimento del nuovo nodo, rimane lo stesso di prima; quindi al passo successivo del ciclo il corrente è il nodo appena inserito



- nel secondo caso il **nodo corrente**, dopo l'inserimento del nuovo nodo, viene aggiornato ad essere il nuovo nodo; quindi al passo seguente è il nodo che era successivo nella lista originale



## Uso della classe StringList: togliere elementi

Problema: eliminare tutti gli elementi iniziati per "E".

## Uso della classe StringList: aggiungere e togliere elementi

- Vogliamo eliminare tutti gli elementi iniziati per "E".**  
Provate: nessuno dei due schemi precedenti funziona !
- Occorre usare uno schema ancora leggermente diverso: **realizzatelo per esercizio** (ricordando che per poter eliminare un nodo occorre avere un riferimento al precedente).
- I metodi della classe StringList permettono all'utilizzatore di creare un cursore con cui scorrere la lista e modificarla, anche aggiungendo e togliendo nodi, e garantiscono il mantenimento dell'invariante. Tuttavia non permettono di adottare uno schema uniforme di ciclo per inserimenti, cancellazioni, ecc.
- Per permettere uno schema uniforme è conveniente definire un oggetto cursore più complesso, contenente due riferimenti a due nodi consecutivi, in modo da poter trattare più semplicemente il caso della cancellazione;
- nella terminologia Java: un oggetto **iteratore**.

## Senza definire una classe Iterator: soluzione insoddisfacente

- Si definiscono nella classe **List** due altri campi interni di tipo **Node**, *corrente* e *precedente*, con metodi per:
  - inizializzarli (impostando *corrente* al primo elemento);
  - avanzare di un passo;
  - sapere se la lista ha ancora degli elementi oppure è finita
  - modificare o cancellare l'elemento corrente;
  - ecc.
- La classe annidata **Node** può essere resa privata, quindi invisibile all'utilizzatore, poiché non serve più.
- Svantaggi: così si ha un solo cursore per la lista, e non è possibile percorrere la lista contemporaneamente con due cursori, come ad esempio nel selection-sort o nell'insertion sort.

## Nota terminologica

Nello scrivere e parlare di programmazione a oggetti si usa spesso, per brevità, la locuzione:

un oggetto *NomeDiClasse*  
invece di:  
un oggetto (o istanza) della classe *NomeDiClasse*.

Esempio:  
un oggetto StringList = un oggetto della classe StringList

Non si confondano però gli oggetti con le loro classi !  
Nella programmazione Java avanzata (che non faremo), si vede che anche ogni classe è rappresentata da un oggetto (della classe Class ...); naturalmente, non si deve confondere l'oggetto-classe con gli oggetti della classe !

## Gli iteratori: introduzione.

- ogni oggetto della classe `StringListIterator` rappresenta uno stato astratto di "percorrenza di una lista", costituito da:
  - indicazione dell'**ultimo nodo visitato** (o nodo **corrente**);
  - indicazione del **penultimo visitato** (o nodo **precedente**);
- su tale stato astratto sono definiti i seguenti predicati:
  - **ci sono ancora elementi da visitare**;
  - **è permesso effettuare una cancellazione**;
- ogni invocazione del metodo `next()` fa due azioni:
  - avanza al nodo successivo;
  - restituisce il dato contenuto in tale nodo successivo.

## Gli iteratori: inizializzazione e avanzamento.

- stato iniziale dell'iteratore:
  - il nodo **ultimo visitato** (o **corrente**) è ovviamente nullo;
  - il **penultimo visitato** (o **precedente**) è pure nullo;
- dopo il primo passo:
  - l'**ultimo visitato** è uguale al primo della lista (`first`);
  - il **precedente** è ancora nullo;
- dopo il k-esimo passo (con  $k > 1$ ):
  - l'**ultimo** è uguale al successivo del vecchio ultimo;
  - il **penultimo** è uguale al vecchio ultimo;

## Possibile definizione alternativa

- Lo "stato di percorrenza" di una lista potrebbe essere invece definito come costituito da:
  - indicazione del **prossimo nodo da visitare**;
  - indicazione dell'**ultimo visitato**.
- Ma in genere si vuole operare su un nodo in base al dato in esso contenuto, cioè dopo averlo visitato; allora bisogna avere, in caso di cancellazione, accesso al nodo che precede l'ultimo visitato.
- Se lo stato è rappresentato dai nodi **prossimo** e **ultimo**, invece che **ultimo** e **penultimo**, bisogna che la visita del nodo non faccia automaticamente avanzare la percorrenza.
- Al posto del metodo `next` vanno allora definiti due metodi distinti:
  - `current()` che restituisce il dato nel nodo corrente
  - `goToNext()` che avanza di un passo.

## Iteratori in Java

- La libreria di Java ha scelto la prima alternativa, cioè lo stato costituito da **ultimo** e **penultimo**.
- Altri linguaggi fanno scelte un po' diverse, e inoltre usano una terminologia un po' diversa.
- Ad esempio in `C#` attualmente gli iteratori si chiamano enumeratori, e non permettono la cancellazione o inserimento di elementi.

## Gli iteratori: introduzione (continua).

- per percorrere le liste vogliamo usare oggetti (della classe) `StringListIterator` **invece di** oggetti della classe `Node`, la quale sarà pertanto dichiarata **solo package-visibile**.
- ma quando si crea un oggetto `StringListIterator`, bisogna passargli il nodo da cui inizia l'iterazione:  
**se la classe-nodo non è pubblica, come si fa ?**

### due possibili soluzioni:

1. si passa come argomento al costruttore dell'iteratore l'oggetto-lista su cui operare;
2. gli oggetti iteratori vengono creati e forniti all'utilizzatore dagli oggetti-lista stessi, e solo da essi, attraverso opportuni metodi della classe-lista (soluzione SUN).

## Gli iteratori: introduzione (continua).

- per percorrere e modificare una lista ho bisogno dei metodi `next()`, `addAfter(...)`, `removeNext()`; questi metodi erano nella classe `Node`,  
**ma ora Node non è pubblica: come si fa ?**
- se il dato contenuto nel nodo è di tipo primitivo, per poterlo modificare occorre che il campo-dati sia pubblico;
- se il dato nel nodo è un (riferimento a) oggetto, ma si vuole poter sostituire l'intero oggetto, il campo-dati deve essere pubblico;  
**ma ora Node non è pubblica: come si fa?**

### ovvia soluzione:

- `next()`, `addAfter(...)`, `removeNext()` e un nuovo metodo `set()` diventano metodi della classe-iteratore:
- o definiti direttamente (e solamente) nell'iteratore;
  - oppure richiamanti gli omonimi metodi di `Node`;

## Soluzione 1: iteratore creato all'esterno della lista, lista passata come argomento al costruttore dell'iteratore

```
package lists;
public class StringListIterator {
    private Node current;
    private Node previous;
    private StringList list;

    public StringListIterator(StringList lis) {
        current = null;
        previous = null;
        list = lis;
    }

    public String() next() {
        ...
        if(current == null)
            current = list.first();
        ...
    }
}
```

AlgELab-05-06 - Lez.01

55

## Soluzione 1: utilizzazione

```
...
class UseStringList {
    public static void main(String[] args) {
        StringList myList = StringList.read("nomi.txt");

        StrListIterator i = new StrListIterator(myList);
        while(i.hasNext()) {
            String elem = i.next();
            i.set("Ciao, " + elem);
        }
        // supponendo di aver definito in StringList il metodo toString
        System.out.println(myList);
    }
}
```

AlgELab-05-06 - Lez.01

56

## Soluzione 2: iteratore creato (solo) all'interno della lista

```
package lists;
public class StringListIterator {
    ...
    private Node list;
    costruttore non pubblico !
    StringListIterator(lis) {
        ...
        list = lis;
    }
} //
```

non approfondiamo  
questa soluzione  
"intermedia"

```
package lists;
public class StringList {
    ...
    public StringListIterator iterator() {
        return new StringListIterator(this);
    }
} //
```

AlgELab-05-06 - Lez.01

57

## Soluzione ottima: iteratore come classe interna

- Si definisce la classe-iteratore **dentro** la classe-lista, ma definendola come **non statica**, cioè come vera **inner class**.
- Così (vedi prossima slide) un oggetto della classe-iteratore ha un "**oggetto circondante**" della classe lista, e può accedere direttamente ai campi di tale oggetto.
- La **classe-iteratore** può essere addirittura resa **privata**, purché si definisca un' **interfaccia-iteratore** (pubblica) e si dichiarino la classe interna privata iteratore come implementante l'interfaccia.
- In questo modo l'**utilizzatore** non deve preoccuparsi del nome composto *ClasseLista.ClasseIteratore*: anzi, **non conosce il nome della classe degli iteratori** che riceve e usa! Conosce solo il nome dell'interfaccia.
- È la soluzione adottata dalla SUN nella libreria di Java.

AlgELab-05-06 - Lez.01

58

## Ricorda: classi annidate in Java (slide già vista)

- Campi e metodi di una classe annidata sono sempre visibili e accessibili dalla classe circondante.
- Le classi annidate possono essere soggette alle stesse restrizioni di visibilità che si usano per campi e metodi: **private**, **protected**, ecc. con significati analoghi, ad esempio:
  - classe annidata **private**: non è visibile al di fuori della classe circondante;
  - classe annidata **protected**: visibile nelle classi derivate e nel package;
  - ... ecc.
- Le classi annidate possono essere dichiarate **static** oppure no:
  - **classe annidata statica**:
    - non ha accesso a metodi e campi non statici della classe circondante;
    - può essere usata anche dai metodi statici della classe circondante;
  - **classe annidata NON statica (inner class)**:
    - ogni oggetto della classe annidata contiene un puntatore ad un oggetto della classe circondante;
    - i metodi della classe annidata hanno accesso ai campi e metodi (di istanza) di tale "**oggetto circondante**".

AlgELab-05-06 - Lez.01

59

## Un'interfaccia iteratore

```
package liste;
public interface StringListIterator {
    boolean hasNext();
    String next();
    void add(String element);
    void remove();
    void set(String element);
}
```

Ricorda: i metodi dichiarati in una interfaccia sono implicitamente pubblici.

AlgELab-05-06 - Lez.01

60

## La classe StringList con il metodo che fornisce iteratori

```
public class StringList2 {
    private static class Node {
        ...
    }

    private class Itr implements StringListIterator {
        private Node current;
        private Node previous;
        ...
    }

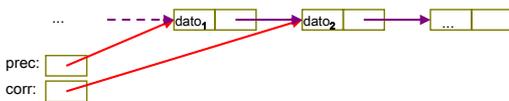
    private Node first;
    ...
    public StringListIterator iterator() {
        return new Itr();
    }
    ...
}
```

## Iteratore: eliminazione del nodo corrente.

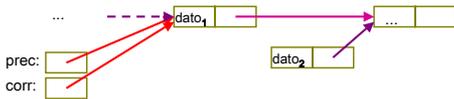
- La conoscenza del nodo precedente al corrente permette la cancellazione del nodo corrente;
- affinché in un ciclo l'avanzamento di un nodo faccia sempre passare al primo nodo ancora da visitare (senza saltarne nessuno), occorre che **dopo la cancellazione si consideri come nuovo nodo corrente il precedente**;
- ma allora non è possibile aggiornare correttamente il precedente, che dovrebbe diventare il precedente del precedente; lasciandolo invariato, esso risulta invece uguale al corrente;
- in questa situazione quindi non sarà quindi possibile cancellare il corrente, perché non abbiamo il suo vero precedente;
- la situazione di "non è permessa la cancellazione" è quindi segnalata dalla **coincidenza "anomala" fra corrente e precedente**; non c'è pertanto bisogno di una rappresentazione esplicita di tale stato per mezzo di un campo.
- in questo stato "anomalo" il primo avanzamento del nodo corrente riporta il precedente ad essere il vero precedente, e quindi rende di nuovo possibile una successiva cancellazione.

## Eliminazione del nodo corrente

### PRIMA



### DOPO



## La classe StringListIterator: specifica

Definizione preliminare di due nozioni:

- **nodo corrente o ultimo-visitato**: è il nodo il cui dato è stato restituito dall'ultima invocazione del metodo `next()`; se `next()` non è mai stato invocato, è nullo;
- **nodo precedente o penultimo**: è il nodo che precede l'ultimo, eccetto i seguenti casi: se "non è permesso effettuare una cancellazione", coincide con l'ultimo; se il nodo corrente è nullo o è il primo, è nullo.

Metodi (pubblici):

- **boolean hasNext()**: è true se il successivo del corrente non è nullo; nel caso di corrente nullo, è true se il primo della lista non è nullo;
- **String next()**:  
 PRE : `hasNext()` è true, `corrente = nodo C`, `precedente = nodo P`  
 POST: `C` non è nullo  $\Rightarrow$  `corrente = successivo di C` e `precedente = C`;  
`C` è nullo  $\Rightarrow$  `corrente = primo` e `precedente = nullo`;  
 "è permesso effettuare una cancellazione" è vero;  
 il risultato restituito è il dato contenuto nel nuovo corrente;

## La classe StringListIterator: specifica (continua)

- **void add(String elem)**: aggiunge un elemento dopo il corrente o, se il corrente è nullo, lo aggiunge in testa; l'elemento aggiunto diventa il nuovo corrente, il vecchio corrente diventa il precedente; "è permesso effettuare una cancellazione" è vero; (per esercizio si scrivano più precisamente PRE e POST)
- **void remove()**:  
 PRE: "è permesso effettuare una cancellazione" è vero; elimina il nodo corrente; se il nodo corrente è il primo, aggiorna il riferimento al primo; il nuovo nodo corrente è il precedente; "è permesso effettuare una cancellazione" diventa falso (per esercizio si scrivano più precisamente PRE e POST)
- **public void set(String element)**: cambia l'ultimo elemento visitato.

## Utilizzazione

```
StringList2 myList =
    StringList2.read("nomi.txt");

StringListIterator i = myList.iterator();
while(i.hasNext()) {
    if(i.next().startsWith("E")) i.remove();
}

i = myList.iterator();
while(i.hasNext()) {
    String elem = i.next();
    i.set("Ciao, " + elem);
}
```

## Ulteriori raffinamenti

- Nelle API Java esiste l'interfaccia **Iterator** che dichiara i metodi **hasNext()**, **next()** e **remove()**.
- Nelle API Java esiste l'interfaccia **Iterable**, che dichiara il metodo **iterator()**; cioè una classe implementa **Iterable** se ha un metodo che restituisce un iteratore.
- Con gli oggetti **Iterable** si può usare il nuovo costrutto **for-each**, nel modo che vedremo.
- Allora la nostra interfaccia **StringListIterator** può essere dichiarata estensione di **Iterator**.
- La nostra classe **StringList** può essere dichiarata come implementante **Iterable**.
- **Iterator** e **Iterable** sono interfacce "generiche", cioè aventi un *parametro di tipo* (argomento che vedremo meglio in seguito).

## Interfaccia estensione di Iterator

```
public interface StringListIterator
extends java.util.Iterator<String> {
    boolean hasNext();
    String next();
    void add(String element);
    void remove();
    void set(String element);
}
```

## Classe implementante Iterable

```
public class StringList2 implements Iterable<String> {
    private static class Node {
        ...
    }

    private class Itr implements StringListIterator {
        private Node current;
        private Node previous;
        ...
    }

    private Node first;
    ...
    public StringListIterator iterator() {
        return new Itr();
    }
    ...
}
```

## Utilizzazione con ciclo for-each

```
StringList2 myList = ...;
...
for(String s: myList) {
    System.out.println(s.toUpperCase());
}
```

è equivalente a:

```
Iterator i = myList.iterator();
while(i.hasNext()) {
    s = i.next();
    System.out.println(s.toUpperCase());
}
```

cioè: per ogni elemento **s** della lista **myList** fai ...

Nota: il costrutto non può essere usato per un ciclo in cui si modificano, aggiungono o tolgono nodi, poiché in tal caso occorre l'esplicito riferimento all'iteratore **i**.

## Compito 0.1

- Realizzare, nel package */iste*, la classe **StringList2** come descritto nei lucidi precedenti, cioè:
  - con classe-iteratore interna privata non statica la quale implementi un'interfaccia-iteratore;
  - un metodo **iterator()** che restituisce un oggetto iteratore;
  - ecc.
- Realizzare, fuori del package */iste*, una classe **UsaStringList2** contenente un **main** che provi il funzionamento della classe **StringList2**.
- Si evitino, per ora, raffinamenti sostanziali rispetto a quanto descritto (come nodo header iniziale, liste doppiamente concatenate, ecc.); chi lo desidera potrà realizzarli e consegnarli in un secondo tempo.