Programmazione Orientata agli Oggetti in Linguaggio Java

Tecniche di Programmazione: Collezioni Parte a

versione 2.3

G. Mecca – Università della Basilicata – mecca@unibas.it

Sommario

- Liste
- Iteratori
  - ⇨Ciclo For Migliorato

Liste

- Nel package java.util
  - ⇨due tipi di liste
  - ⇨entrambe implementano l'interfaccia java.util.List
- ArrayList
  - ⇨basata su array e indicatore di riempimento
- LinkedList
  - ⇨basata su rappresentazione collegata

Liste

- Attenzione
  - ⇨i due tipi di liste hanno prestazioni diverse nelle operazioni fondamentali
- Infatti
  - ⇨la rappresentazione con array facilita l'accesso agli elementi data la posizione
  - ⇨ma penalizza gli inserimenti e le cancellazioni in mezzo alla lista
  - ⇨è necessario spostare gli elementi su o giù

Liste

- Viceversa
  - ⇨LinkedList è più lenta nell'accesso agli elementi data la posizione
  - ⇨accedere l'elemento in posizione i richiede la scansione di i riferimenti
  - ⇨ma è più veloce negli inserimenti e nelle cancellazioni (approssimativamente costano quanto la scansione)

Liste

- Un test di prestazioni
  - ⇨basato su circa 1000 operazioni
  - ⇨tempi misurati in millisecondi

Type	**Get**	**Insert**	**Remove**
array	172	na	na
ArrayList	281	328	30484
LinkedList	5828	109	16
Vector	422	360	30781

fonte: Thinking in Java

Liste

- Considerazioni sulle prestazioni
  - ⇨gli array sono i più veloci, ma non consentono inserimenti e cancellazioni
  - ⇨ArrayList è veloce nell'accesso agli elementi, lenta in inserimenti e cancellazioni in mezzo
  - ⇨LinkedList è più lenta nell'accesso, ma decisamente più veloce in inserimenti e canc.
  - ⇨Vector è più lenta di entrambe e non dovrebbe essere utilizzata

Liste

- Di conseguenza
  - ⇨nel caso di liste a bassa dinamica, per ridurre i tempi di scansione è opportuno usare ArrayList
  - ⇨viceversa, per liste ad alta dinamica, con frequenti inserimenti e cancellazioni conviene utilizzare LinkedList
- E che succede se devo cambiare tipo ?
  - ⇨es: passare da ArrayList a LinkedList

Liste

- Linea guida
  - ⇨è opportuno programmare con le interfacce invece che con le implementazioni
  - ⇨le interfacce riducono l'accoppiamento tra le classi e semplificano i cambiamenti
- Nel caso delle liste
  - ⇨è opportuno utilizzarle per quanto possibile attraverso riferimenti di tipo java.util.List

Liste

- Esempio: lista di impegni
  - ⇨dichiaro il riferimento alla lista di impegni del giorno di tipo java.util.List
  - ⇨creo l'oggetto utilizzando una delle implementazioni; es: ArrayList
  - ⇨nel resto del codice, utilizzo solo riferimenti di tipo java.util.List per manipolare l'oggetto

```
package it.unibas.appuntamenti.modello;

public class Giorno {

    private java.util.List listaImpegni = new java.util.ArrayList();

    public java.util.List getListaImpegni() {
        return this.listaImpegni;
    }
    ...
}

public class Principale {

    public void stampaImpegni(Giorno giorno) {
        java.util.List lista = giorno.getListaImpegni();
        // codice per la stampa degli elementi della lista
    }
}
```

la lista viene creata attraverso una delle implementazioni ma viene manipolata esclusiv. con riferimenti all'interfaccia

Liste

- In questo modo
  - ⇨le modifiche sono semplificate
  - ⇨basta cambiare le poche istruzioni in cui gli oggetti di tipo lista sono creati cambiando la classe usata per l'implementazione
  - ⇨il resto dell'applicazione resta intatta
  - ⇨i metodi si comportano polimorficamente e viene utilizzata la nuova implementazione

Iteratori

- Attenzione
  - ⇨in questo approccio, quando manipolo la lista devo tenere in considerazione che l'implementazione potrebbe cambiare
- In particolare
  - ⇨devo fare attenzione a non basare la scrittura del codice su una o l'altra delle implementaz.
- Un'operazione critica
  - ⇨la scansione

Iteratori

- Il modo tipico di scandire una lista
  - ⇨utilizzando indici interi
- Esempio

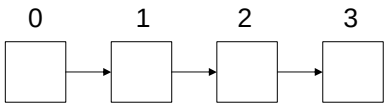
```
for (int i = 0; i < lista.size(); i++) {
    Object o = lista.get(i);
    // operazioni su o
}
```

-chiamo lista.get(0);
-poi lista.get(1);
-poi lista.get(2); ecc.

Iteratori

- Questo tipo di scansione
  - ⇨è particolarmente adatto ad ArrayList (il metodo get viene eseguito rapidamente)
  - ⇨ma disastrosamente lenta su LinkedList
- Perchè ?
  - ⇨perchè come detto l'accesso all'elemento in posizione i di una LinkedList richiede di scandire i elementi (i operazioni circa)

Iteratori



- Di conseguenza
  - ⇨lista.get(0) richiede 1 operazione
  - ⇨lista.get(1) ne richiede 2
  - ⇨lista.get(2) ne richiede 3
  - ⇨...
  - ⇨lista.get(i) ne richiede i+1
  - ⇨lista.get(lista.size() - 1) ne richiede lista.size()

Iteratori

- Detta n la dimensione della lista
  - ⇨1 + 2 + 3 + 4 + ... + n
  - ⇨pari circa a n(n +1)/2, ovvero dell'ordine di n^2
  - ⇨es: per una lista di 100 elementi: 5000
  - ⇨nel caso di ArrayList: circa 100 operazioni
- In casi normali
  - ⇨il problema non sorge (liste piccole)
  - ⇨ma in alcuni casi si tratta di un costo di calcolo che può diventare inaccettabile

Iteratori

- Il problema
  - ⇨la scansione attraverso indici interi NON è la scansione più naturale per LinkedList
- ArrayList
  - ⇨implementazione basata su indici >> scansione naturale basata su indici
- LinkedList
  - ⇨implementazione basata su riferimenti >> scansione naturale basata su riferimenti

Iteratori

- Idealmente
  - ⇨vorrei che per ciascuna tipologia di lista potesse essere utilizzata automaticamente la scansione più adatta
  - ⇨senza che il programmatore se ne debba preoccupare
- Attenzione
  - ⇨in questo caso il polimorfismo da solo non basta

Iteratori

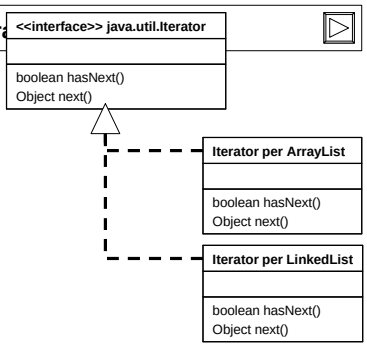
- Infatti
  - ⇨la scansione della lista è un'operazione che deve necessariamente essere effettuata da un oggetto diverso dalla lista
  - ⇨non posso quindi semplicemente sovrascrivere il metodo "scandisciti()" e utilizzarlo polimorficamente
  - ⇨devo necessariamente definire altri oggetti la cui responsabilità è quella di scandire la lista

Iteratori

- Soluzione
  - ⇨utilizzare un oggetto “iteratore”
- Iteratore
  - ⇨oggetto specializzato nella scansione di una lista
  - ⇨fornisce al programmare un’interfaccia per effettuare la scansione in modo indipendente dalla strategia di scansione concreta (indici, puntatori, ecc.)
  - ⇨implementa la scansione in modo ottimale per ciascun tipo di lista

Iteratori

- L’utilizzo in java.util
  - ⇨interfaccia java.util.Iterator, che prevede i seguenti metodi
  - ⇨Object next() per spostarsi in avanti
  - ⇨boolean hasNext() per fermarsi
  - ⇨esiste poi una implementazione per ArrayList
  - ⇨ed una implementazione per LinkedList



Iteratori

- Iteratore per ArrayList
  - ⇨utilizza indici interi
- Iteratore per LinkedList
  - ⇨scandisce la lista usando i riferimenti
- Come si ottiene l'iteratore ?
  - ⇨utilizzando il metodo Iterator iterator() di java.util.List


```
package it.unibas.appuntamenti.modello;

public class Giorno {

    private java.util.List listaImpegni = new java.util.LinkedList();

    public String stringaImpegni() {
        ordinaImpegni();
        StringBuffer stringaImpegni = new StringBuffer();
        java.util.Iterator iterator = this.listaImpegni.iterator();
        while (iterator.hasNext()) {
            Object impegnoIesimo = iterator.next();
            stringaImpegni.append(impegnoIesimo.toString());
            stringaImpegni.append("\n");
        }
        return stringaImpegni.toString();
    }
```

```
package it.unibas.volpieconigli.modello;
public class Gioco {
   private Scacchiera scacchiera;

    public void simula() {
      java.util.List listaAnimali = this.scacchiera.getListaAnimali();
      for (java.util.Iterator it = listaAnimali.iterator(); it.hasNext(); ) {
         Animale animale = (Animale)it.next();
         animale.agisci(this.scacchiera);
      }
   }
```

il for è equivalente a questo while:

```
java.util.Iterator it = listaAnimali.iterator();
while (it.hasNext()) {
    Animale animale = (Animale)it.next();
    if (animale != null) {
        animale.agisci(this.scacchiera);
    }
}
```


Iteratori

- Dettagli sugli iteratori di java.util
  - ⇨sostanzialmente si basano sui metodi next() e previous() forniti dalle due liste
  - ⇨sono però più complicati di quanto si pensa, dal momento che consentono anche di modificare la lista durante la scansione
  - ⇨attraverso il metodo remove()
  - ⇨senza doversi preoccupare della consistenza dei riferimenti

Iteratori

- Un ulteriore esempio di iteratore
  - ⇨uno StringTokenizer è in effetti un iteratore
  - ⇨ha un metodo boolean hasNext() che consente di scandire stringhe che contengono un numero di token variabile

```
while (tokenizer.hasNext()) {
    String token = tokenizer.next();
    ...
}
```


Ciclo For Migliorato

ATTENZIONE
ciclo for migliorato
J2SE 5.0

- Una novità di J2SE 1.5
  - ⇨il ciclo for migliorato (“enhanced for loop”)
  - ⇨un modo sintatticamente compatto per utilizzare un iteratore su una collezione
- Sintassi

```
for (<Tipo> <riferimento> : <Collezione>) {
    <operazioni su <riferimento>>
}
```

Ciclo For Migliorato

- Esempio

```
public String stringaImpegni() {
        ordinaImpegni();
        StringBuffer stringaImpegni = new StringBuffer();
        for (Object iesimo : this.listaImpegni) {
            stringaImpegni.append(iesimo.toString());
            stringaImpegni.append("\n");
        }
        return stringaImpegni.toString();
}
```


Ciclo For Migliorato

- Esempio
  - ⇨in questo caso nel corpo è necessario un cast ad Animale
  - ⇨nel for ottengo solo riferimenti a Object

```
public void simula() {
   java.util.List listaAnimali = this.scacchiera.getListaAnimali();
   for (Object o : listaAnimali) {
         Animale animale = (Animale)o;
         animale.agisci(this.scacchiera);
   }
}
```

Riassumendo

- Liste
- Iteratori
  - ⇨Ciclo For Migliorato

Termini della Licenza