UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Informatica
@UnivAQ

Laboratorio di Algoritmi e Strutture Dati
a.a. 2023/2024

Java Collections Framework:
Interfacce Iterable, Iterator, ListIterator

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM

Gli iteratori

- Un iteratore è un oggetto che rappresenta il «cursore» con cui esplorare sequenzialmente la collezione alla quale è associato
- Un iteratore è sempre associato ad un oggetto collezione
- Per funzionare, un oggetto iteratore deve essere a conoscenza degli aspetti più nascosti di una classe, quindi la sua realizzazione dipende interamente dalla collection class concreta che implementa la collezione
- **`Iterator` è un'interfaccia** (non una classe). Questa è sufficiente per utilizzare tutte le funzionalità dell'iteratore senza doverne conoscere alcun dettaglio implementativo.

Interfaccia Iterable<E>

```
public interface Iterable<E> {
    public Iterator<E> iterator();
    default void forEach(Consumer<? super T> action)
}
```

- Ogni classe che implementa **`Iterable<E>`** deve avere un metodo **`iterator()`** che restituisce un iteratore sugli elementi interni alla classe stessa
- Il metodo **`forEach`** esegue la data azione per ogni elemento della classe Iterabile. L'implementazione di default si comporta come:

```
for (T t : this)
    action.accept(t);
```

Interfaccia `Iterator<E>` (1 di 2)

```
public interface Iterator <E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
}
```

- `next()` che restituisce l'elemento corrente della collezione, e contemporaneamente sposta il cursore all'elemento successivo;
- `hasNext()` che verifica se il cursore ha ancora un successore o se si è raggiunto la fine della collezione;
- `remove()` che elimina l'elemento restituito dall'ultima invocazione di `next()`;
- `remove()` è opzionale perché in certi casi non si vogliono mettere a disposizione del cliente metodi che permettano modifiche arbitrarie alla collezione.

L'interfaccia `Iterator<E>` (2 di 2)

- Si noti che l'iteratore non ha metodi che lo reinizializzino
  - una volta iniziata la scansione, non si può fare tornare indietro l'iteratore
  - una volta finita la scansione, l'iteratore non è più utilizzabile (ne serve uno nuovo)
- È possibile usare più iteratori contemporaneamente

L'interfaccia `Iterator<E>`: esempio d'uso

```
Collection <String> c = ... //collezione in cui sono memorizzati oggetti di classe String
...
Iterator it = c.iterator();  // restituisce l'iteratore associato a c
while (it.hasNext()) {       // finche' il cursore non e' all'ultimo elemento
   String s = it.next();     // poni l'elemento corrente in s ed avanza
   System.out.println(s);    // stampa l'elemento corrente (denotato da s)
   ...
}
```

L'interfaccia `Iterator`: schema tipico

```
Iterator<T> it = «ottieni un iteratore per la collezione»
while (it.hasNext()){
   T elem = it.next();
   «elabora l'elemento»
}
```

Iteratori: il problema dei duplicati

```
public static boolean verificaDupOrdIterator(List<String> S){
      Collections.sort(S);  //ordina la lista di stringhe
      Iterator<String> it = S.iterator();
      if (!it.hasNext()) return false;
      String pred = it.next();
      while (it.hasNext()){
            String succ=it.next();
            if (pred.equals(succ)) return true;
            pred=succ;
      }
      return false; }
```

Il ciclo for-each

Se un oggetto **`myColl`** appartiene ad una collection class che implementa **`Iterable<A>`**, per una data classe A, è possibile scrivere il seguente ciclo for-each:

```
for (A a: myColl) {
// corpo del ciclo
...
}
```

- `for (A a: <exp>){…}` è corretto a queste condizioni:
  - <exp> è una espressione di tipo “array di T” oppure di un sottotipo di “Iterable<T>”
  - T è assegnabile ad A

Il ciclo for-each vs iterator

- Il ciclo precedente è equivalente al blocco seguente:

```
Iterator<A> it = myColl.iterator();
while (it.hasNext()) {
A a = it.next();
// corpo del ciclo
...
}
```

- Come si vede, il ciclo for-each è più sintetico e riduce drasticamente il rischio di scrivere codice errato

Il ciclo for-each: esempio

- Esempio: sia **`myColl`** un riferimento ad un esemplare di una classe `Collection<String>`. Si vogliono visualizzare tutti i suoi elementi che iniziano con la lettera 'a'
- "for each word di tipo `String` in `myColl` … "

```
for (String word: myColl)

   if (word.charAt(0)=='a') System.out.println(word);
```

Iteratori: il metodo `remove()`

- Durante l'iterazione di una collezione, il modo più sicuro per eliminare un elemento della collezione è eseguire il metodo **`remove()`** dell'iteratore (da non confondere con il metodo **`remove()`** della collezione)
- **`remove()`** elimina l'elemento restituito dall'ultima invocazione di **`next()`** (optional operation) e può essere invocato solo una volta
- il comportamento dell'iteratore non è specificato se la collezione è modificata in modo diverso dalla chiamata di **`remove()`**, mentre l'iteratore è in esecuzione
- **Nota**: l'iterazione su una collezione usando i costrutti for/forEach crea implicitamente un iteratore che è necessariamente inaccessibile. Pertanto in questo caso la collezione può essere solo ispezionata, non è possibile effettuare nessuna operazione di cancellazione.
- Esempio: rif. **`RandomList`** (Parte 2)

Interfaccia ListIterator<E>

```
public interface ListIterator<E>  extends Iterator<E>
```

Methods

Modifier and Type	Method and Description
void	**add**(E e) Inserts the specified element into the list (optional operation).
boolean	**hasNext**() Returns `true` if this list iterator has more elements when traversing the list in the forward direction.
boolean	**hasPrevious**() Returns `true` if this list iterator has more elements when traversing the list in the reverse direction.
E	**next**() Returns the next element in the list and advances the cursor position.
int	**nextIndex**() Returns the index of the element that would be returned by a subsequent call to **next**().
E	**previous**() Returns the previous element in the list and moves the cursor position backwards.
int	**previousIndex**() Returns the index of the element that would be returned by a subsequent call to **previous**().
void	**remove**() Removes from the list the last element that was returned by **next**() or **previous**() (optional operation).
void	**set**(E e) Replaces the last element returned by **next**() or **previous**() with the specified element (optional operation).

`void add(E e)`

L'elemento **`e`** è inserito:

- immediatamente prima dell'elemento che sarebbe restituito invocando **`next`**(), se esiste, e
- dopo l'elemento che sarebbe restituito invocando **`previous`**(), se esiste
- se la lista è vuota, il nuovo elemento diventa l'unico della lista

void set(E e)

- Rimpiazza l'ultimo elemento restituito da **`next`**() o **`previous`**() con l'elemento specificato
- La chiamata può essere effettuata solo se non sono stati invocati remove() o add(E) dopo l'ultima chiamata a **`next`**() o **`previous`**()

- Esempio: rif. **`TestListIterator`**

UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Informatica
@UnivAQ

Domande?

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM