



Università degli Studi dell'Aquila



Dipartimento di Ingegneria e Scienze
dell'Informazione e Matematica

Università degli Studi dell'Aquila

Corso di Algoritmi e Strutture Dati con Laboratorio
A.A. 2014/15

Java Collections Framework

Java Collections Framework

- ▶ L'infrastruttura JCF è una raccolta di interfacce e classi, tra loro correlate, appartenenti al pacchetto `java.util`.
- ▶ Un esemplare di una classe di tale infrastruttura rappresenta generalmente una raccolta (**collection**) composta da elementi.
- ▶ Queste classi possono usare **tipi parametrici**, in modo che un utente possa specificare il tipo a cui appartengono gli elementi della raccolta nel momento in cui dichiara un esemplare di tale raccolta

Raccolte

- ▶ Una raccolta (collection) è un oggetto composto da elementi.
- ▶ Esempio: un array è una raccolta di elementi dello stesso tipo che vengono memorizzati in aree contigue della memoria

```
String[] names = new String[5];
```

Array

- ▶ **Vantaggio:**

- Si può accedere ad un singolo elemento dell'array in modo diretto (proprietà di accesso casuale)

- ▶ **Svantaggi:**

- La dimensione è fissa. Se la dimensione si rivela insufficiente occorre creare un array più grande e copiarvi il contenuto di quello più piccolo.
- L'inserimento o la rimozione di un elemento può richiedere lo spostamento di molti elementi.
- Queste operazioni di mantenimento devono essere gestite dal programmatore

“Collection class”

- ▶ Un’alternativa migliore all’uso degli array?
Usare esemplari di classi che rappresentano raccolte (“*collection class*”)
- ▶ Una collection class è una classe i cui singoli esemplari sono raccolte di elementi
- ▶ Gli elementi in un’istanza di una collezione devono essere riferimenti ad un oggetto.
 - Non possiamo creare un esemplare di una raccolta i cui singoli elementi siano di tipo primitivo
 - Possiamo usare le classi wrapper

Strutture di memorizzazione

- ▶ Strutture di memorizzazione per classi che rappresentano raccolte:
 1. **Contiguous collection** (Raccolta contigua): Il modo più semplice per archiviare in memoria una raccolta prevede di memorizzare in un array i riferimenti ai singoli elementi: in pratica la classe ha un array come campo

Strutture di memorizzazione

2. **Linked collection:** invece di usare la contiguità, gli elementi possono essere correlati tra loro mediante collegamenti (link), ovvero riferimenti.
 - In una classe che realizza una raccolta mediante collegamenti, ciascun elemento presente in un suo esemplare è memorizzato in una entry o nodo, che contiene almeno un collegamento ad un altro nodo.

Linked collection

- ▶ **Singly-linked list** (lista semplicemente concatenata): raccolta realizzata mediante collegamenti, dove ciascun nodo contiene un elemento ed un riferimento al nodo successivo presente nella raccolta
- ▶ **Doubly-linked list** (lista doppiamente collegata): raccolta realizzata mediante collegamenti, dove ciascun nodo contiene un elemento, un riferimento al nodo precedente ed un riferimento al nodo successivo presente nella raccolta
- ▶ **Binary search tree...**

Tipi parametrici

- ▶ Remark: gli elementi in un'istanza di una collection class devono essere riferimenti ad un oggetto.
- ▶ La versione 1.5 ha introdotto una nuova, importante funzionalità nel linguaggio: la programmazione parametrica, in Inglese anche detta *generics*.
- ▶ Si tratta della possibilità di specificare il tipo di un elemento dotando classi, interfacce e metodi di parametri di tipo. Questi parametri hanno come possibili valori i tipi del linguaggio. In particolare, possono assumere come valore qualsiasi tipo, esclusi i tipi primitivi (tipi base).

L'interfaccia Collection

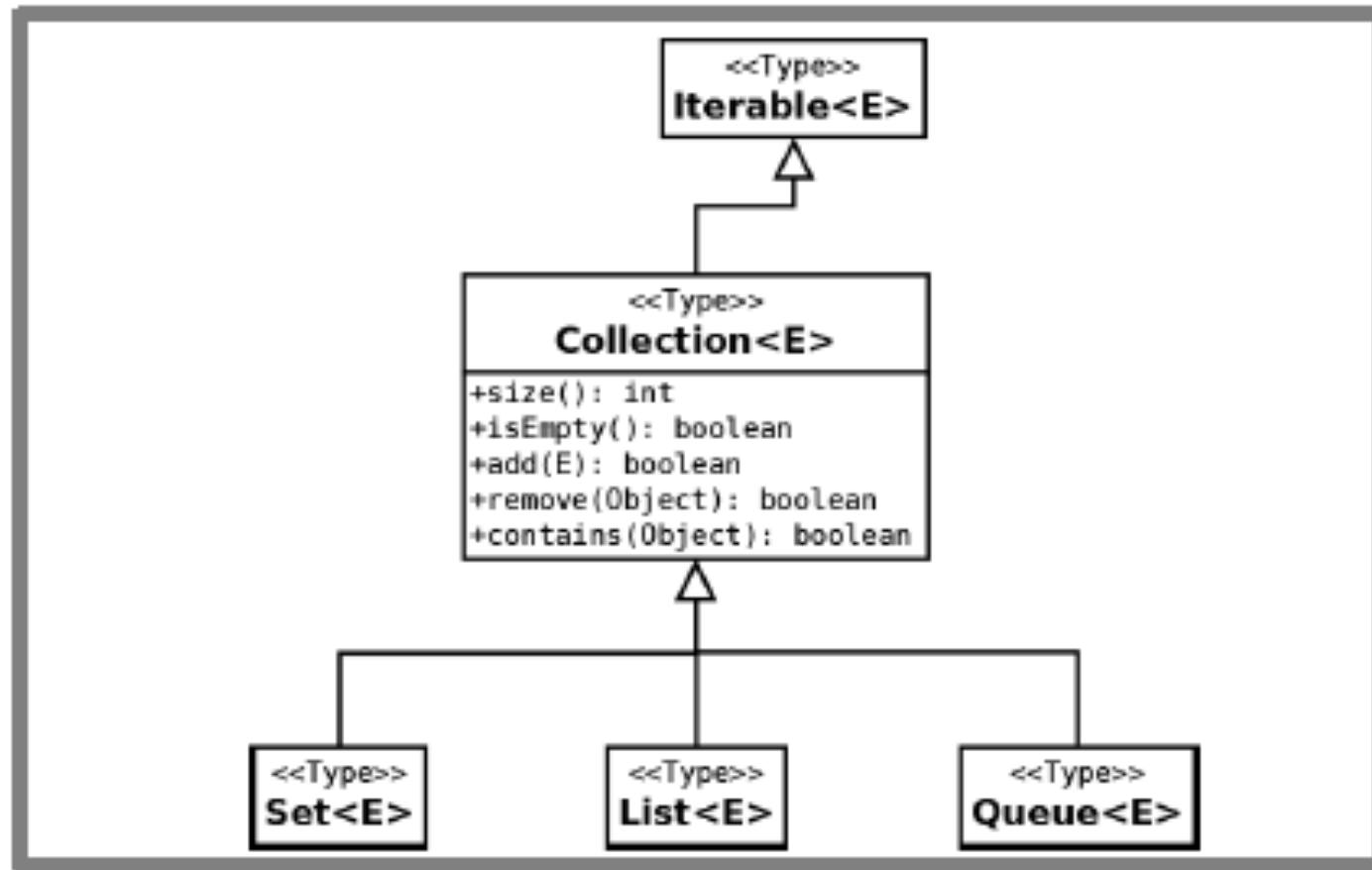
- ▶ Il Java Collection Framework (JCF) è una parte della libreria standard dedicata alle collezioni
- ▶ Offre strutture dati di supporto molto utili alla programmazione, come array di dimensione dinamica, liste, insiemi, mappe associative (anche chiamate dizionari) e code
- ▶ Il JCF è costituito in pratica da una gerarchia che contiene classi astratte e interfacce ad ogni livello tranne l'ultimo, dove sono presenti soltanto classi che implementano interfacce e/o estendono classi astratte

L'interfaccia Collection

- ▶ In cima alla gerarchia troviamo le interfacce **Collection** e **Map**.
- ▶ L'interfaccia Collection estende la versione parametrica di **Iterable**
- ▶ Inoltre, la classe Collections (si noti la “s” finale) contiene numerosi algoritmi di supporto
 - ad esempio, metodi che effettuano l'ordinamento

L'interfaccia Collection

- ▶ Interfacce collegate con Collection:



L'interfaccia Collection

- ▶ Può sorprendere che i metodi `contains` e `remove` accettino `Object` invece del tipo parametrico `E`
- ▶ Lo fanno perché non si corre alcun rischio a passare a questi due metodi un oggetto di tipo sbagliato
- ▶ Entrambi i metodi restituiranno `false`, senza nessun effetto sulla collezione stessa

Iteratori

- ▶ Consideriamo i seguenti esempi:
 - Dato un oggetto di una classe che implementa l'interfaccia Collection ("oggetto Collection" in breve) di studenti, visualizzare gli studenti migliori
 - Dato un oggetto Collection di membri di un club, aggiornare le quote dovute da ciascuno di essi
 - Dato un oggetto Collection di dipendenti a tempo pieno, calcolare il loro salario medio
- ▶ Notiamo che in ciascun esempio il compito da svolgere richiede l'accesso a tutti gli elementi di un oggetto Collection, uno dopo l'altro.

Iteratori

- ▶ Come si può fare in modo che qualsiasi implementazione dell'interfaccia Collection consenta ai suoi utilizzatori di eseguire un'iterazione che coinvolga, uno dopo l'altro, tutti gli elementi presenti in un suo esemplare, senza violare il principio di astrazione per i dati?
- ▶ La soluzione risiede nell'uso degli **iteratori**, oggetti che consentono di accedere agli elementi di oggetti Collection .

Iteratori

- ▶ In Java un iteratore ha due primitive fondamentali specificate dall'interfaccia `java.util.Iterator`:
 1. `hasNext()` : verifica se c'è ancora un elemento nella collezione
 2. `next()` : restituisce il prossimo elemento della collezione

Iteratori

- ▶ Le interfacce Iterator e Iterable sono parametriche:

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

- ▶ Lo scopo ultimo del parametro di tipo consiste nel permettere al metodo next di restituire un oggetto del tipo appropriato, evitando il ricorso ad un cast

Iteratori: un esempio

- ▶ Esempio: sia `myColl` un riferimento ad un esemplare di una classe `Collection` contenente elementi di tipo `String`. Si vogliono visualizzare tutti i suoi elementi che iniziano con la lettera ‘a’.
- ▶ Creiamo un oggetto iteratore. Un iteratore si ottiene invocando il metodo `iterator()` sull’oggetto che rappresenta la collezione stessa:

```
Iterator<String> itr = myColl.iterator();
```

Iteratori: un esempio

► Eseguiamo la scansione:

```
String word;  
while (itr.hasNext()) {  
    word=itr.next();  
    if (word.charAt(0)=='a')  
        System.out.println(word);  
}
```

► Schema tipico:

Iterator<T> it= ottieni un iteratore per la collezione

```
while (it.hasNext()) {  
    T elem=it.next();  
    elabora l'elemento  
}
```

Iteratori

- ▶ Ogni classe per rappresentare collezioni di elementi dovrebbe implementare l'interfaccia Iterable.
- ▶ Ad esempio l'interfaccia `java.util.List` estende l'interfaccia Iterable, pertanto ogni oggetto di tipo List è “iterabile”.

Iteratori: il problema dei duplicati

```
public static boolean verificaDupOrdIterator(List S) {  
    Collections.sort(S);  
    Iterator it=S.iterator();  
    if (!it.hasNext()) return false;  
    Object pred = it.next();  
    while (it.hasNext()) {  
        Object succ=it.next();  
        if (pred.equals(succ)) return true;  
        pred=succ;  
    }  
    return false;  
}
```

Il ciclo for-each

- ▶ Se un oggetto x appartiene ad una classe che implementa `Iterable<A>`, per una data classe A , è possibile scrivere il seguente ciclo:

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

Il ciclo for-each

- ▶ Il ciclo precedente è equivalente al blocco seguente:

```
Iterator<A> it = x.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

- ▶ Il seguente ciclo:

```
for (A a: <exp>) {  
    // corpo del ciclo  
    ...  
}
```

- ▶ è corretto a queste condizioni:

1. <exp> è una espressione di tipo “array di T” oppure di un sottotipo di “Iterable<T>”
2. T è assegnabile ad A

Il ciclo for-each

- ▶ Esempio: sia myColl un riferimento ad un esemplare di una classe Collection contenente elementi di tipo String. Si vogliono visualizzare tutti i suoi elementi che iniziano con la lettera ‘a’.
- ▶ “*for each word in myColl...*”

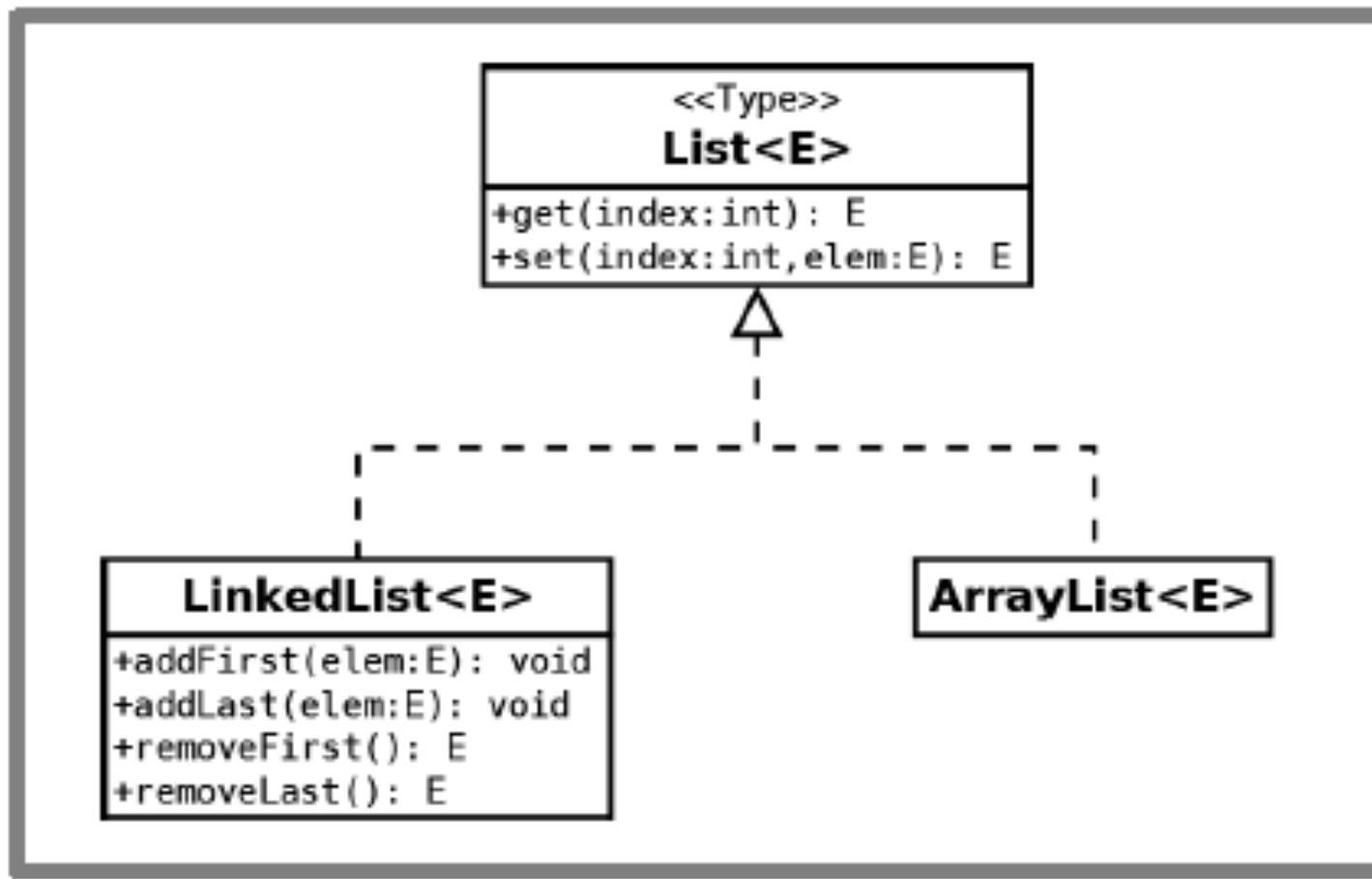
```
for (String word: myColl)
    if (word.charAt(0) == 'a')
        System.out.println(word);
```

L'interfaccia List

- ▶ L'interfaccia List estende l'interfaccia Collection aggiungendo alcuni metodi relativi all'uso di indici
- ▶ In ogni esemplare di una classe che implementa l'interfaccia List gli elementi sono memorizzati in sequenza, in base ad un indice
- ▶ Vista come entità indipendente dal linguaggio di programmazione, una lista è un tipo di dato astratto

L'interfaccia List

- ▶ L'interfaccia List e le classi che la implementano:



L'interfaccia List

- ▶ La classe `ArrayList` realizza l'interfaccia `List` mediante un array
- ▶ La classe `LinkedList` realizza l'interfaccia `List` mediante liste doppiamente concatenate
- ▶ Esempio: la classe `RandomList` crea e manipola un oggetto `List` contenente numeri interi casuali (`RandomList.java`)

L'interfaccia List

RandomList.java:

- ▶ Avremmo potuto usare un enunciato for-each avanzato per scandire `randList`? No, perché oltre a ispezionare la lista, eliminiamo alcuni elementi contenuti in essa.
- ▶ La variabile `randList` è stata dichiarata come riferimento polimorfico e inizializzata con un riferimento ad un oggetto di tipo `ArrayList`.
- ▶ Per eseguire nuovamente il programma usando un oggetto di tipo **LinkedList** l'unica modifica necessaria è l'invocazione del costruttore:

```
List<Integer> randList=new LinkedList<Integer>();
```

La classe ArrayList

- ▶ ArrayList è un'implementazione di List, realizzata internamente con un **array di dimensione dinamica**
- ▶ Ovvero, quando l'array sottostante è pieno, esso viene riallocato con una dimensione maggiore, e i vecchi dati vengono copiati nel nuovo array
- ▶ Questa operazione avviene in modo trasparente per l'utente
- ▶ Il metodo `size` restituisce il numero di elementi effettivamente presenti nella lista, non la dimensione dell'array sottostante
- ▶ Il ridimensionamento avviene in modo che l'operazione di inserimento (`add`) abbia *complessità ammortizzata costante* (per ulteriori informazioni sulla complessità ammortizzata, si consulti un testo di algoritmi e strutture dati)

La classe LinkedList

- ▶ La classe `LinkedList` offre i seguenti metodi, in aggiunta a quelli di `List`, presentati nella loro versione parametrica

<code>public void addFirst(E elem)</code>	aggiunge <code>x</code> in testa alla lista
<code>public void addLast(E elem)</code>	equivalente ad <code>add(x)</code> , ma senza valore restituito
<code>public E removeFirst()</code>	rimuove e restituisce la testa della lista
<code>public E removeLast()</code>	rimuove e restituisce la coda della lista

La classe LinkedList

- ▶ I metodi permettono di utilizzare una `LinkedList` sia come **stack** sia come **coda**
- ▶ Per ottenere il comportamento di uno **stack** (detto **LIFO**: *last in first out*), inseriremo ed estrarremo gli elementi dalla **stessa estremità della lista**
 - ad esempio, inserendo con `addLast` (o con `add`) ed estraendo con `removeLast`
- ▶ Per ottenere, invece, il comportamento di una **coda** (**FIFO**: *first in first out*), inseriremo ed estrarremo gli elementi da **due estremità opposte**

Le liste e l'accesso posizionale

- ▶ L'accesso posizionale (metodi `get` e `set`) si comporta in maniera molto diversa in `LinkedList` rispetto ad `ArrayList`
- ▶ In `LinkedList`, ciascuna operazione di accesso posizionale può richiedere un tempo proporzionale alla lunghezza della lista (complessità *lineare*)
- ▶ In `ArrayList`, ogni operazione di accesso posizionale richiede tempo *costante*
- ▶ Pertanto, è fortemente sconsigliato utilizzare l'accesso posizionale su `LinkedList`
- ▶ Se l'applicazione richiede l'accesso posizionale, è opportuno utilizzare un semplice array, oppure la classe `ArrayList`