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 Collection e List e relative implementazioni

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM

Richiami: Java Collections Framework

- L'infrastruttura Java Collections Framework (**JCF**) è una libreria formata da un insieme di interfacce e di classi che le implementano per lavorare con **gruppi di oggetti** (**collezioni**)
- Un esemplare di una classe del JCF rappresenta generalmente una collezione
- Il JCF offre **strutture dati efficienti** di supporto molto utili alla programmazione, come array di dimensione dinamica, liste, insiemi, mappe associative (anche chiamate dizionari) e code
- La raccolta di interfacce e classi, tra loro correlate, appartengono al package **java.util**

Java Collections Framework (2 di 4)

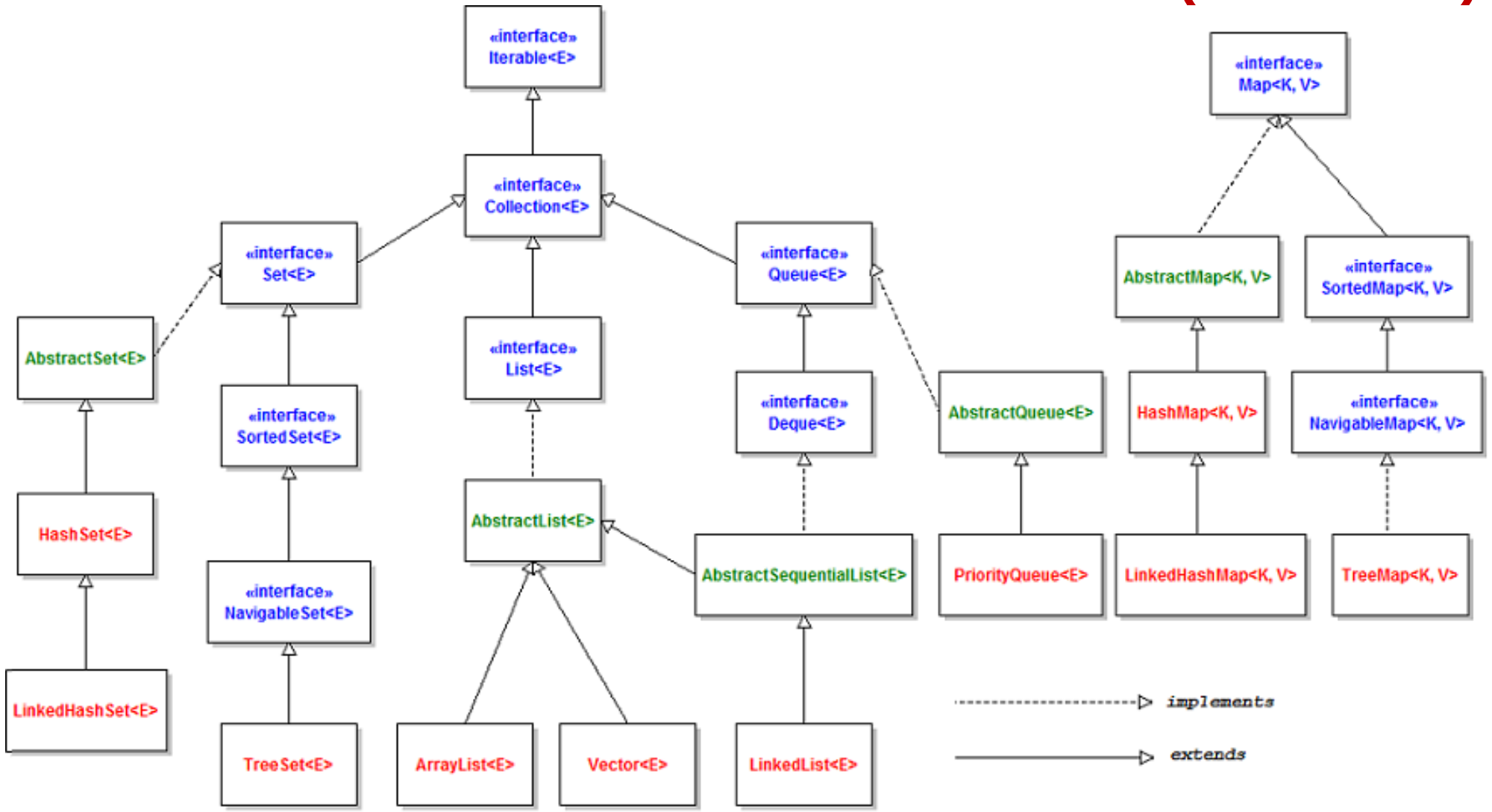
- In pratica, il JCF è costituito da una **gerarchia** che contiene classi astratte e interfacce ad ogni livello tranne l'ultimo, dove sono presenti soltanto classi che implementano interfacce e/o estendono classi astratte:
  - Le **interfacce** rappresentano vari tipi di collezioni di uso comune
  - Le implementazioni sono **classi concrete** che implementano le interfacce di cui sopra, utilizzando strutture dati efficienti
  - I **metodi** realizzano algoritmi di uso comune, quali algoritmi di ricerca e di ordinamento su oggetti che implementano le interfacce

Java Collections Framework (3 di 4)

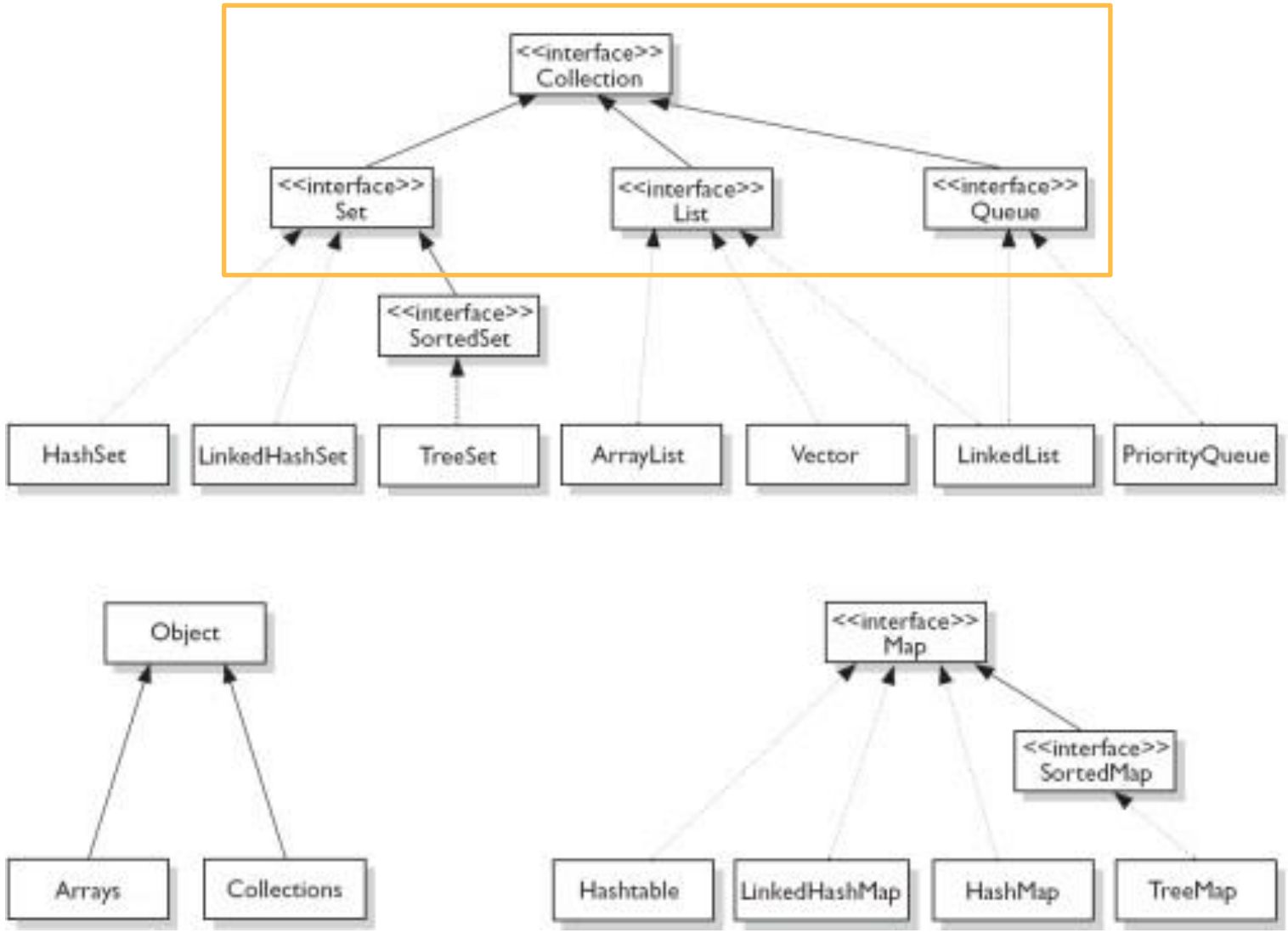
Perché usare il JCF?

- Generalità: permette di modificare l'implementazione di una collezione senza modificare i client
- Interoperabilità: permette di utilizzare (e farsi utilizzare da) codice realizzato indipendentemente dal nostro
- Efficienza: **le classi che realizzano le collezioni sono ottimizzate** per avere prestazioni particolarmente buone.

Java Collections Framework (4 di 4)



L'interfaccia `Collection`



```
public interface Collection<E> extends Iterable<E> {
  // Basic Operations
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(E element);          // Optional
  boolean remove(Object element); // Optional
  Iterator<E> iterator();

  // Bulk Operations
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c);//Optional
  boolean removeAll(Collection<?> c);       //Optional
  boolean retainAll(Collection<?> c);       //Optional
  void clear();                             //Optional

  // Array Operations
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```

L'interfaccia specifica

- Operazioni di base quali inserimento, cancellazione, ricerca di un elemento nella collezione
- Operazioni che lavorano su intere collezioni quali l'inserimento, la cancellazione la ricerca di collezioni di elementi
- Operazioni per trasformare il contenuto della collezione in un array.
- Operazioni **"opzionali"** che lanciano `UnsupportedOperationException` se non supportati da una data implementazione dell'interfaccia.

```
public interface Collection<E> extends Iterable<E> {
  // Basic Operations
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(E element);          // Optional
  boolean remove(Object element); // Optional
  Iterator<E> iterator();

  // Bulk Operations
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c);//Optional
  boolean removeAll(Collection<?> c);       //Optional
  boolean retainAll(Collection<?> c);       //Optional
  void clear();                             //Optional

  // Array Operations
```

L'interfaccia specifica

- Operazioni di base quali inserimento, cancellazione, ricerca di un elemento nella collezione
- Operazioni che lavorano su intere collezioni quali l'inserimento, la cancellazione la ricerca di collezioni di elementi
- Operazioni per trasformare il contenuto della collezione in un array.
- Operazioni **"opzionali"** che lanciano `rtedOperationException` supportati da una implementazione dell'interfaccia.

Ricorda: wildcard `?` indica un tipo non specificato
`Collection<?>` è supertipo di qualunque **`Collection<T>`**

```
public interface Collection<E> extends Iterable<E> {
  // Basic Operations
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(E element);          // Optional
  boolean remove(Object element); // Optional
  Iterator<E> iterator();

  // Bulk Operations
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c);//Optional
  boolean removeAll(Collection<?> c);       //Optional
  boolean retainAll(Collection<?> c);       //Optional
  void clear();                             //Optional

  // Array Operations
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```

Ricorda: Bounded Wildcard `<? extends E>`
indica un tipo non specificato sottotipo di **`E`**

L'interfaccia specifica

- Operazioni di base quali inserimento, cancellazione, ricerca di un elemento nella collezione
- Operazioni che lavorano su intere collezioni quali l'inserimento, la cancellazione la ricerca di collezioni di elementi
- Operazioni per trasformare il contenuto della collezione in un array.
- Operazioni **"opzionali"** che lanciano `UnsupportedOperationException` se non supportati da una data implementazione dell'interfaccia.

L'interfaccia `Collection`: esempio d'uso

- *Se* qualcuno ci fornisse un oggetto di classe Collection sapremmo già come usarlo!!!

```
Collection <String> miaColl = ...
/* NOTA: <String> definisce il tipo degli elementi in miaColl.
miaColl e' una "collezione di String" */

miaColl.add("Ciao"); // Aggiunge un oggetto di tipo String a miaColl
miaColl.clear(); // Svuota miaColl
String[] mioArray; // Crea un riferimento ad array di String;
miaColl.toArray(mioArray); // Popola mioArray con gli elementi di miaColl
...
```

L'interfaccia `Collection`: main methods

- **`int size()`**
  - restituisce il numero di elementi presenti nella collection
- **`boolean isEmpty()`**
  - verifica se la collection oggetto di invocazione (corrente) è vuota
- **`boolean add(E e)`**
  - aggiunge un oggetto alla collection corrente
- **`boolean remove(Object o)`**
  - rimuove un oggetto dalla collection corrente
- **`boolean contains(Object o)`**
  - verifica l'esistenza di un oggetto all'interno della collection corrente

L'interfaccia `Collection`: `contains` e `remove`

- 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

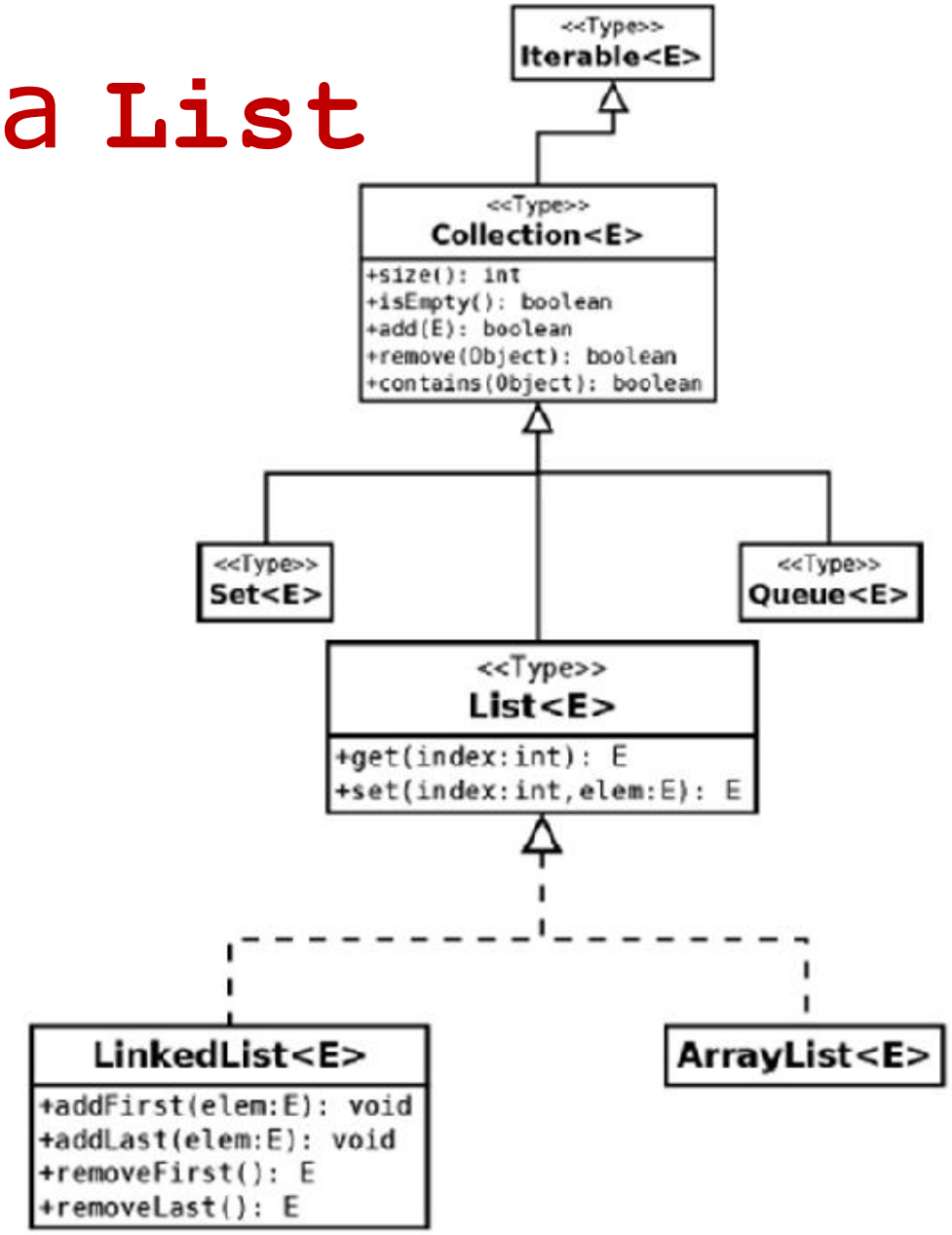
L'interfaccia `Collection`: other methods

- **`addAll(Collection<? extends E> c)`**
  - aggiunge una collection di oggetti alla collection corrente
- **`void clear()`**
  - svuota la collection corrente
- **`boolean containsAll(Collection<?> c)`**
  - verifica l'esistenza di tutti gli elementi della collection specificata all'interno della collection corrente
- **`Iterator<E> iterator()`**
  - restituisce un'istanza di una classe che implementa l'interfaccia **`Iterator`** che permette di scorrere la collezione oggetto di invocazione
  - Nota che si tratta del metodo «ereditato» dall'interfaccia **`Iterable`**

- **`boolean removeAll(Collection<?> c)`**
  - rimuove gli elementi della collection specificata dalla collection corrente
- **`boolean retainAll(Collection<?> c)`**
  - conserva solo gli elementi della collection che sono contenuti nella collection specificata
- **`Object[] toArray()`**
  - restituisce la collection corrente sottoforma di array;
- **`<T> T[] toArray(T[] a)`**
  - restituisce la collection corrente sottoforma di array; il tipo runtime dell'array restituito è quello dell'array specificato

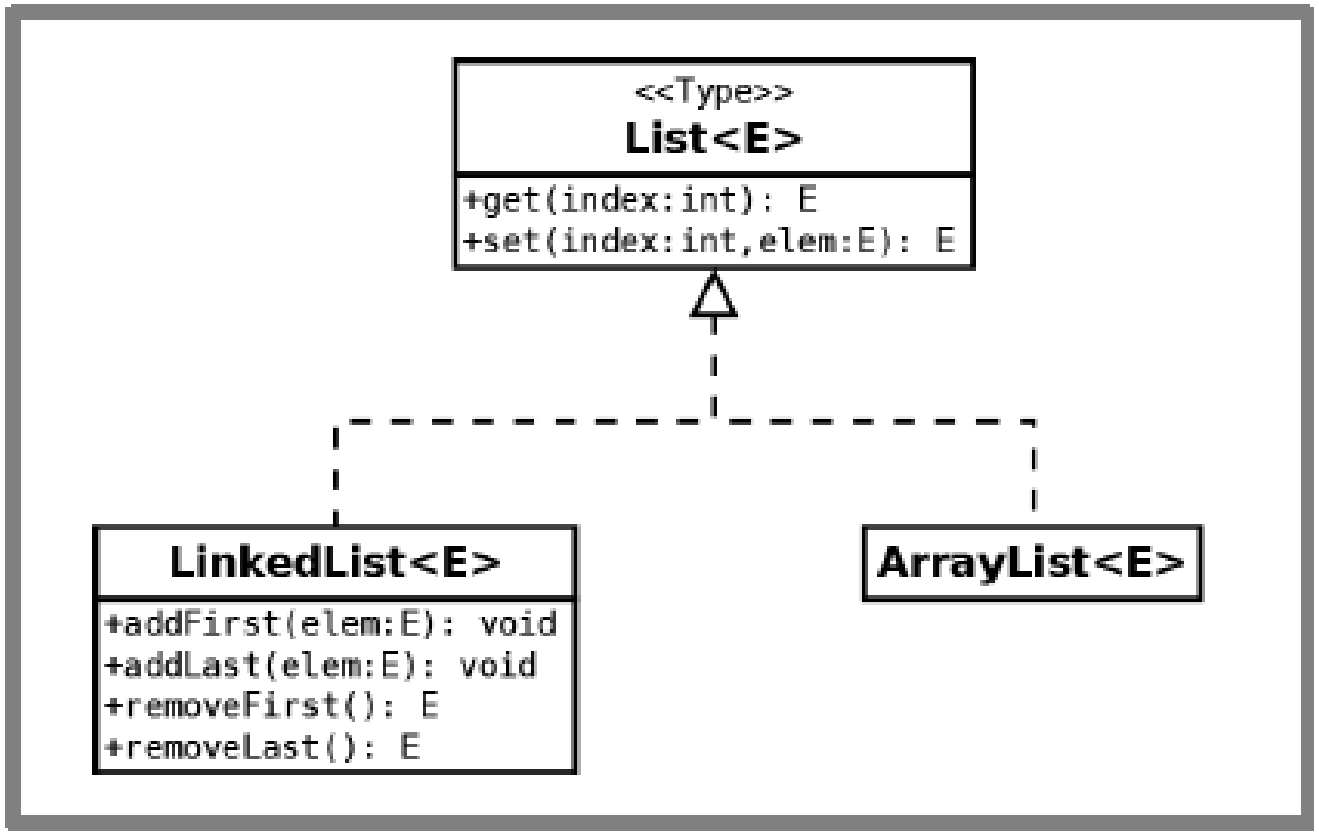
L'interfaccia `List`

- 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: main methods

- L’interfaccia List e le classi che la implementano:



L'interfaccia `List`: realizzazioni

- Come sappiamo per uno stesso tipo di dato sono possibili diverse realizzazioni alternative basate su strutture dati diverse
- In generale, la scelta di una particolare struttura dati consente un'implementazione delle operazioni richieste più o meno efficiente
- L'efficienza dipende anche dal modo in cui i dati sono organizzati all'interno della struttura.
- Un modo naturale per implementare una struttura dati che realizza un certo tipo di dato è scrivere una classe che ne implementa la corrispondente interfaccia

Tecniche per rappresentare collezioni di oggetti

Tecniche fondamentali usate per rappresentare collezioni di elementi:

1. Tecnica basata su **strutture indicizzate** (array)
2. Tecnica basata su **strutture collegate** (record e puntatori)

- La scelta di una tecnica piuttosto che di un'altra può avere un impatto cruciale sulle operazioni fondamentali (ricerca, inserimento, cancellazione, ...)

1. Strutture indicizzate: proprietà

- (**Forte**) Gli indici delle celle di un array sono numeri interi consecutivi
  - Il tempo di accesso ad una qualsiasi cella è costante ed indipendente dalla dimensione dell'array
- (**Debole**) Non è possibile aggiungere nuove celle ad un array
  - Il ridimensionamento è possibile solo mediante la riallocazione dell'array, ossia la creazione di un nuovo array e la copia del contenuto dal vecchio al nuovo array

Ridimensionamento di array

- L'idea è quella di non effettuare riallocazioni ad ogni inserimento/cancellazione, ma solo ogni Ω(n) operazioni
- Se h è la dimensione dell'array e le prime n>0 celle dell'array contengono gli elementi della collezione, la tecnica consiste nel mantenere una **dimensione h** che soddisfa, per ogni n>0, la seguente **invariante**:

$$\mathbf{n \leq h < 4n}$$

Analisi ammortizzata – cenno (1 di 2)

- È una tecnica di analisi di complessità che considera il tempo richiesto per eseguire, nel caso pessimo, un'**intera sequenza di operazioni** su una struttura dati.
- Esistono operazioni più o meno costose.
- Se le operazioni più costose sono poco frequenti (come il ridimensionamento di array), allora il loro costo può essere ammortizzato con l'esecuzione dalle operazioni meno costose.

Analisi ammortizzata – cenno (2 di 2)

- Si calcola la complessità O(f(n)) dell'esecuzione di una sequenza di n operazioni nel caso pessimo.
- Il costo ammortizzato della singola operazione si ottiene quindi dividendo per n tale complessità ottenendo O(f(n)/n).
- In questo modo viene attribuito lo stesso costo ammortizzato a tutte le operazioni.

Tecnica del raddoppiamento-dimezzamento (1 di 2)

L'invariante $n \leq h < 4n$ sulla dimensione dell'array viene mantenuta mediante riallocazioni così effettuate:

- Inizialmente, per n=0, si pone h=1
- Quando n>h, l'array viene riallocato raddoppiandone la dimensione (**h ← 2h**)
- Quando n scende a h/4 l'array viene riallocato dimezzandone la dimensione (**h ← h/2**)

Tecnica del raddoppiamento-dimezzamento (2 di 2)

Nota teorica: Se v è un array di dimensione h≥n contenente una collezione non ordinata di n elementi, usando la tecnica del raddoppiamento-dimezzamento ogni operazione di inserimento o cancellazione di un elemento richiede "tempo ammortizzato" costante

- Previo eventuale raddoppiamento dell'array, l'inserimento si effettua in posizione n, e poi si incrementa n di 1
- Per la cancellazione dell'elemento in posizione i, lo si sovrascrive con l'elemento in posizione n-1, decrementando n di 1 ed eventualmente dimezzando l'array

2. Strutture dati collegate: record e puntatori

- In Java un record può essere rappresentato in modo naturale mediante un oggetto
- I numeri associati ai record sono i loro indirizzi in memoria
- I record sono creati e distrutti individualmente ed in maniera dinamica, per cui gli indirizzi non sono necessariamente consecutivi
- Un record viene creato esplicitamente dal programma tramite l'istruzione **`new`**, mentre la sua distruzione avviene in modo automatico quando non è più in uso (**garbage collection**)
- Per mantenere i record di una collezione in relazione tra loro ognuno di essi deve contenere almeno un indirizzo di un altro record della collezione

Strutture dati collegate: proprietà

- (**Forte**) è possibile aggiungere o eliminare record ad una struttura collegata
- (**Debole**) Gli indirizzi dei record di una struttura collegata non sono necessariamente consecutivi

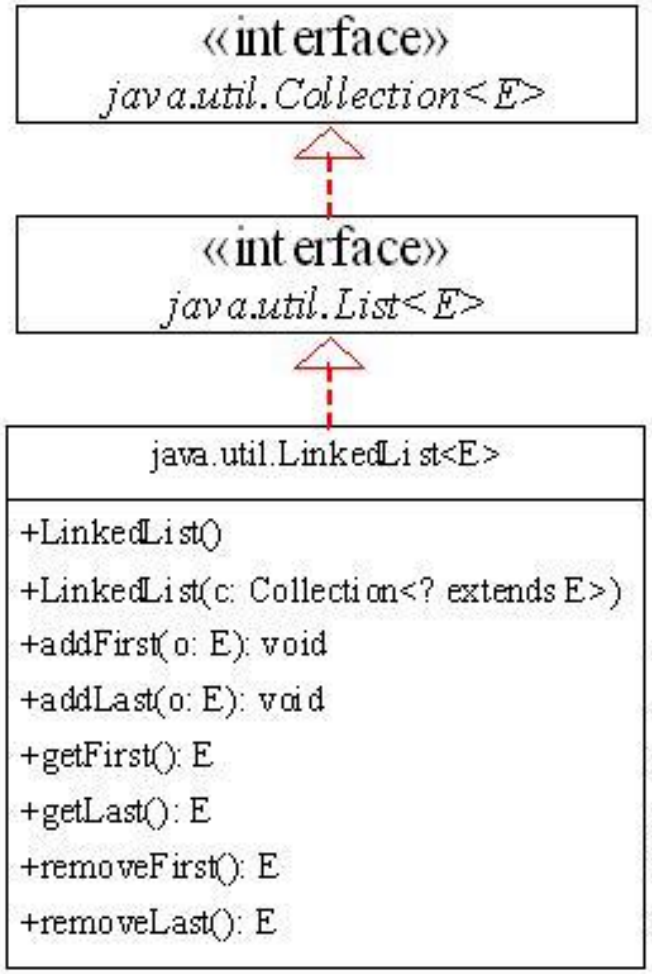
Le classi `ArrayList<E>` e `LinkedList<E>`

- La classe **`ArrayList<E>`** realizza l'interfaccia **`List<E>`** mediante un array
- La classe **`LinkedList<E>`** realizza l'interfaccia **`List<E>`** mediante liste (doppiamente) collegate

La classe `ArrayList`

- **`ArrayList`** è un'implementazione di **`List`**, realizzata internamente con un array dinamico
- La riallocazione dell'array 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

La classe `LinkedList`



Creates a default empty linked list.
Creates a linked list from an existing collection.
Adds the object to the head of this list.
Adds the object to the tail of this list.
Returns the first element from this list.
Returns the last element from this list.
Returns and removes the first element from this list.
Returns and removes the last element from this list.

La classe `LinkedList`: metodi `*First`,`*Last`

- I metodi permettono di utilizzare un oggetto **`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`**

Esercitazione: la classe `RandomList`

Esercizio (PARTE 1): la classe `RandomList` crea e manipola un "oggetto List" contenente numeri interi casuali (rif. `RandomList.java`)

- La variabile `randList<E>` è stata dichiarata come riferimento polimorfico e inizializzata con un riferimento ad un oggetto di tipo `ArrayList<E>`.
- Per eseguire nuovamente il programma usando un oggetto di tipo `LinkedList<E>` l'unica modifica necessaria è l'invocazione del costruttore:

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

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