14.
Java Collection Framework
Le collezioni: insiemi e liste

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli “Federico II”

Corso di Linguaggi di Programmazione II

Il Java Collection Framework

- Il Java Collection Framework (JCF) è una parte della libreria standard dedicata alle *collezioni*, intese come classi deputate a contenere altri oggetti
- Questa libreria offre strutture dati di supporto, molto utili alla programmazione, come liste, array di dimensione dinamica, insiemi, mappe associative (anche chiamate *dizionari*) e code

- Le classi e interfacce del JCF si dividono in due gerarchie:
  - quella che si diparte dall'interfaccia **Collection**
  - e quella che si diparte da **Map**

- Inoltre, la classe **Collections** (si noti la "s" finale) contiene numerosi algoritmi di supporto
  - ad esempio, metodi che effettuano l'ordinamento

- L'interfaccia Collection estende la versione parametrica di Iterable, che andiamo ad introdurre

La versione parametrica di Iterator e Iterable

- Ora che abbiamo introdotto la programmazione parametrica, possiamo svelare che le interfacce Iterator e Iterable sono, in realtà, parametriche
- La loro definizione è la seguente:

```
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 che il chiamante debba ricorrere ad un cast

Il ciclo enhanced-for

- Java 1.5 ha introdotto anche un nuovo tipo di ciclo for, chiamato "*enhanced for*" o "*for each*"

- Cominciamo con un esempio:

```
String[] array = {“uno”, “due”, “tre”};
for (String s: array)
   System.out.println(s);
```

- Il ciclo di sopra stampa tutte le stringhe contenute nell'array, una per rigo

- Questa nuova forma di ciclo permette di iterare su un array senza dover esplicitamente utilizzare un indice
  - quindi *senza il rischio di sbagliare gli estremi dell'iterazione*

- Oltre che per gli array, il ciclo for-each funziona anche su **tutti gli oggetti che implementano Iterable<E>**, come illustrato nella prossima slide

Il ciclo enhanced-for

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 di sopra 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 enhanced-for è più sintetico e riduce drasticamente il rischio di scrivere codice errato

Type checking del for-each

Il seguente ciclo:

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

è corretto a queste condizioni:

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

Esercizio (esame 23/2/2007)

Definire una classe **Primes** che rappresenta l'insieme dei **numeri primi**.

Il campo statico “all” fornisce un oggetto su cui si può iterare, ottenendo l'elenco di tutti i numeri primi.

Non deve essere possibile creare oggetti di tipo Primes.

L'implementazione deve rispettare il seguente esempio d'uso:

```
for (Integer i: Primes.all) {
   if (i > 20) break;
   System.out.println(i);
}
```

Output dell'esempio d'uso:

```
1
3
5
7
11
13
17
19
```

Esercizio (esame 23/2/2007)

Prima di presentare una soluzione, discutiamo brevemente della sua progettazione.

La traccia richiede che “all” sia un campo statico.

E' naturale che sia anche “public”, per essere visto dall'esterno, e “final”, in modo che non possa essere modificato.

Inoltre, in base al caso d'uso, deve puntare ad un oggetto che implementi Iterable<Integer>.

Per quanto riguarda la richiesta che la classe Primes non sia instanziabile, si può ottenere questo risultato dichiarandola astratta, oppure dotandola di un solo costruttore, privato.

La prossima slide presenta una possibile implementazione.

Si noti l'utilizzo di due classi anonime annidate, una per implementare Iterable e l'altra per implementare Iterator.

Esercizio (esame 23/2/2007)

```
public abstract class Primes {
   public static final Iterable<Integer> all = new Iterable<Integer>() {
      public Iterator<Integer> iterator() {
         return new Iterator<Integer>() {
               private int n = 1;
               public boolean hasNext() { return true; }
               public Integer next() {
             ...
               }
             public void remove() { throw new UnsupportedOperationException(); }
          }; // fine classe anonima derivata da Iterator
      } // fine metodo iterator()
   }; // fine classe anonima derivata da Iterable
} // fine classe Primes
```

Esercizio (esame 23/2/2007)

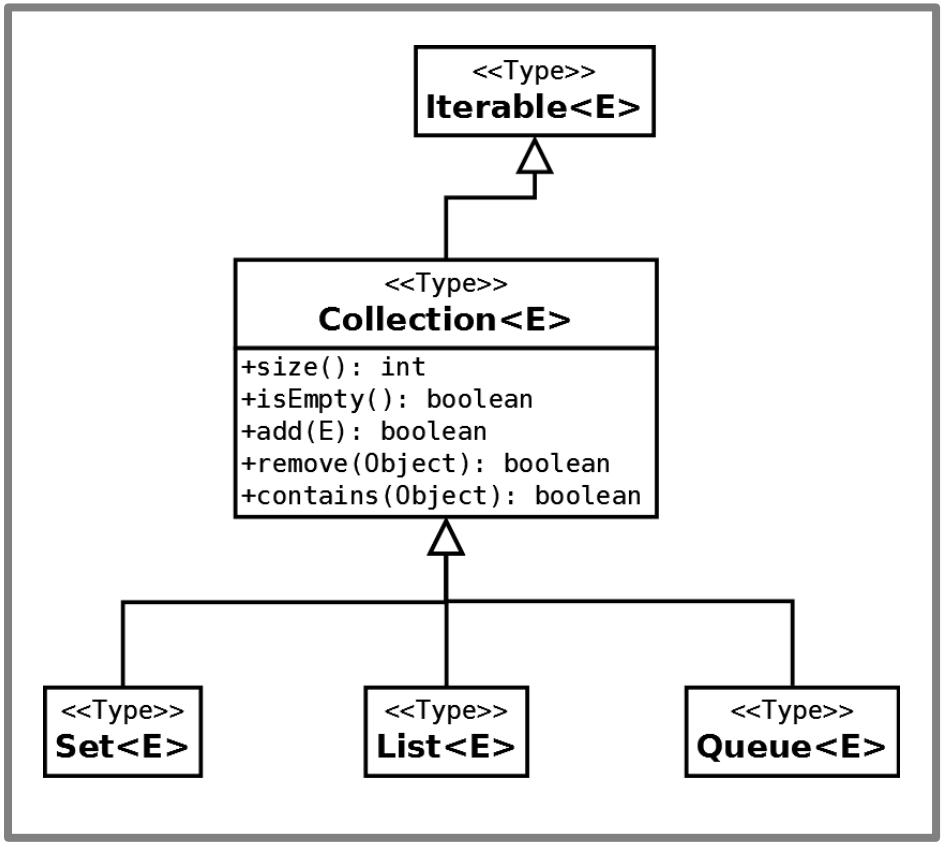
```
public abstract class Primes {
   public static final Iterable<Integer> all = new Iterable<Integer>() {
      public Iterator<Integer> iterator() {
         return new Iterator<Integer>() {
               private int n = 1;
               public boolean hasNext() { return true; }
               public Integer next() {
                int j, temp = n;

                while (true) {
                   n++;
                   // cerca un divisore j di n
                   for (j=2; j<=n/2 ;j++) if (n % j == 0) break;
                   // esce dal while se n e' primo
                   if (j > n/2) break;
                }
                return temp;
               }
            public void remove() { throw new UnsupportedOperationException(); }
          }; // fine classe anonima derivata da Iterator
      } // fine metodo iterator()
   }; // fine classe anonima derivata da Iterable
} // fine classe Primes
```

Interfacce legate a Collection

- Il diagramma a destra illustra le interfacce direttamente collegate con **Collection**
- Abbiamo già detto che Collection estende Iterable, e quindi fornisce un metodo che restituisce un iteratore
- La prossima slide illustra gli altri metodi elencati nella figura

- Collection viene estesa dalle tre interfacce parametriche Set, List e Queue

- Set rappresenta un *insieme* in senso matematico
  - non sono ammessi duplicati
  - l'ordine in cui gli elementi vengono inseriti non è rilevante
- List rappresenta un *vettore*
  - sono ammessi duplicati
  - gli elementi vengono mantenuti nello stesso ordine in cui sono stati inseriti
- Queue rappresenta una *coda*



L'interfaccia Collection<E>

- Come dice il nome, una collection rappresenta un insieme di oggetti
- L'interfaccia ha un parametro di tipo che indica il tipo degli oggetti contenuti
- I metodi principali dell'interfaccia sono i seguenti:

`int size()`	restituisce il numero di oggetti contenuti
`boolean isEmpty()`	restituisce vero se e solo se la collezione è vuota
`boolean add(E x)`	aggiunge x alla collezione, se possibile; restituisce vero se e solo se x è stato aggiunto con successo
`boolean contains(Object x)`	restituisce vero se e solo se la collezione contiene un oggetto uguale (nel senso di equals) ad x
`boolean remove(Object x)`	rimuove l'oggetto x (o un oggetto uguale ad x secondo equals) dalla collezione; restituisce vero se e solo se un tale elemento era presente nella collezione

Osservazioni

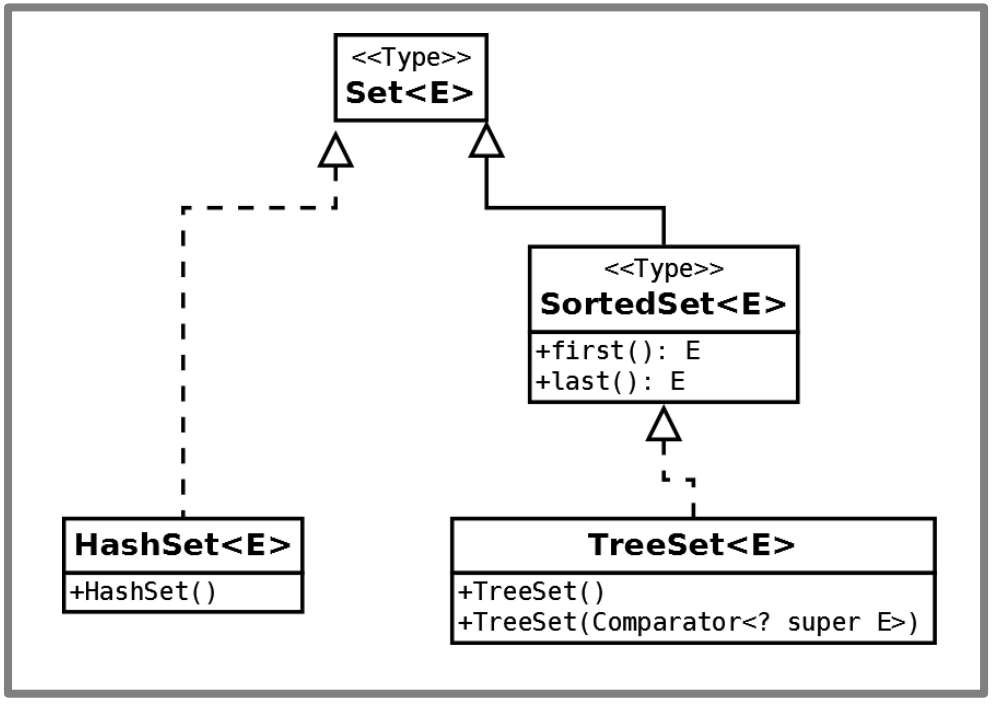
- Come si è visto, le collezioni fanno affidamento al metodo **equals** per identificare gli elementi
  - le implementazioni concrete di Collection usano anche altri metodi, come vedremo tra breve
- Per quanto riguarda il metodo add, il motivo per cui restituisce un valore booleano diventerà chiaro quando esamineremo alcune implementazioni concrete di 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
  - Semplicemente, entrambi i metodi restituiranno false, senza nessun effetto sulla collezione stessa
  - Inoltre, in certi casi la possibilità di passare oggetti qualsiasi potrebbe tornare utile
  - Ad esempio, potremmo trovarci in un metodo che ha a disposizione una Collection<String> e che riceve dall'esterno un Object, che *potrebbe* essere una stringa
  - La firma di contains consente di passargli quest'oggetto, senza doverci preoccupare preventivamente di controllare se si tratta di una stringa o meno

Collezioni senza duplicati: insiemi

L'interfaccia Set e le sue implementazioni

- L'interfaccia Set non aggiunge metodi a Collection
- Tuttavia, restringe, ovvero rende più specifici, i contratti di alcuni metodi, come **add**
- La specificità di un Set, rispetto ad una Collection generica, è che un Set **non può contenere elementi duplicati**
- Quindi, se si tenta di aggiungere con add un elemento che è già presente (ovvero, un oggetto che risulta uguale, secondo *equals*, ad uno già presente), la collezione non viene modificata e add restituisce *false*

- L'interfaccia SortedSet, che estende Set, rappresenta un insieme sui cui elementi è definita una relazione d'ordine (totale)
- L'iteratore di un SortedSet garantisce che gli elementi saranno visitati in ordine, dal più piccolo al più grande
- Inoltre, un tale insieme dispone di due metodi extra:
  - **first** restituisce l'elemento minimo tra quelli presenti nella collezione
  - **last** restituisce l'elemento massimo
  - questi due metodi non modificano la collezione



La classe TreeSet

- TreeSet è un insieme, implementato internamente come **albero di ricerca bilanciato**

- Gli elementi devono essere dotati di una *relazione d'ordine*, in uno dei seguenti modi:

  1) Gli elementi sono dotati di *ordinamento naturale*; in questo caso, si può utilizzare il costruttore di TreeSet senza argomenti.

  Oppure

  2) Bisogna passare al costruttore di TreeSet un opportuno oggetto Comparator; il parametro di tipo "? super E" che compare nella firma del costruttore che accetta Comparator sarà spiegato nelle lezioni successive

Complessità di TreeSet

Le operazioni principali di TreeSet hanno la seguente complessità di tempo, tipica degli alberi di ricerca bilanciati:

size	O(1)
isEmpty	O(1)
add	O(log n)
contains	O(log n)
remove	O(log n)
first	O(1)
last	O(1)

Coerenza tra ordinamento ed uguaglianza

- TreeSet utilizza un ordinamento, fornito tramite Comparable o da Comparator, per **smistare** e poi **ritrovare** gli elementi all'interno dell'albero
- In particolare:

se due oggetti sono equivalenti per l'ordinamento,
saranno considerati uguali dal TreeSet

- Allo stesso tempo, l'interfaccia Set prevede che si usi *equals* per identificare gli elementi

- Quindi, se l'ordinamento non è coerente con l'uguaglianza definita da equals, TreeSet può **violare il contratto di Set**

Coerenza tra ordinamento ed uguaglianza

- Se vogliamo un TreeSet che rispetti il contratto di Set, dobbiamo fornire un ordinamento **coerente con equals**
- Consideriamo il caso di Comparable
- Per ogni coppia di elementi x e y, in aggiunta alle normali proprietà di equals e compareTo, deve valere la seguente condizione:

  x.equals(y) è vero **se e solo se** x.compareTo(y) == 0

- Una condizione analoga deve valere nel caso venga fornito un oggetto di tipo Comparator

La classe HashSet

La classe HashSet

- HashSet è un Set realizzato internamente come **tabella hash**
- Utilizza il metodo hashCode della classe Object per selezionare il bucket in cui posizionare un elemento:

```
public int hashCode()
```

- Per ulteriori dettagli sulle tabelle hash, si consulti un testo di algoritmi e strutture dati

- I principali metodi di HashSet hanno la seguente complessità media:

size	O(1)
isEmpty	O(1)
add	O(1)
contains	O(1)
remove	O(1)

- Tuttavia, le prestazioni reali dipendono dalla “bontà” della funzione hash utilizzata

Coerenza tra equals ed hashCode

- Come previsto dall'interfaccia Set, HashSet utilizza il metodo equals per identificare gli elementi
- Quindi, equals ed hashCode devo rispettare la seguente **regola di coerenza**
- Per ogni coppia di elementi x ed y

se x.equals(y) è vero **allora** x.hashCode() == y.hashCode()

- Si noti che le versioni di equals ed hashCode presenti in Object rispettano tale regola
- Infatti, se x.equals(y) è vero, allora x ed y puntano al medesimo oggetto, e quindi x ed y hanno lo stesso hashCode

Coerenza tra equals ed hashCode

- Esaminiamo il caso di una classe che *non rispetta* la regola di coerenza tra equals ed hashCode
- Supponiamo che una classe Employee ridefinisca equals in modo che confronti il nome degli impiegati, ma non ridefinisca hashCode di conseguenza
- Due oggetti Employee x ed y con lo stesso nome risulteranno uguali secondo equals, ma avranno codici hash diversi

Se inseriamo l'oggetto x in un HashSet:

```
Set<Employee> s = new HashSet<Employee>();
s.add(x);
```

una successiva chiamata a s.contains(y) *potrebbe* restituire “false”!

Infatti, il codice hash di y potrebbe indurre la struttura dati a cercare l'oggetto in questione in un bucket diverso da quello in cui è stato inserito x.

Ridefinizione di hashCode

Riassumendo, una buona ridefinizione di hashCode deve rispettare le seguenti proprietà:

1) Coerenza con equals (necessario)

2) Coerenza temporale, cioè il valore dipende solo dallo stato dell'oggetto (necessario)

3) Uniformità, cioè il valore di ritorno è uniformemente distribuito sugli interi (desiderabile)

Ridefinizione di hashCode

- Perché HashSet funzioni in maniera efficiente, ed in particolare perché le operazioni principali abbiano complessità costante, è necessario che la classe componente (cioè, la classe degli elementi contenuti) disponga di un opportuno metodo hashCode
- Senza entrare nel dettaglio della teoria delle funzioni di hash, è importante che il metodo hashCode assegni ad ogni oggetto un numero intero il più possibile *uniformemente distribuito*

- Ad esempio, supponiamo che la classe Employee disponga dei campi name (String) e salary (int), e che siano considerati uguali gli impiegati che hanno questi due campi uguali
- Il metodo hashCode dovrebbe restituire un intero derivato dai valori dei due campi
- Per i tipi non numerici, come le stringhe, conviene partire dal loro codice hash
- Poi, si devono combinare gli interi ottenuti dai vari campi in un unico numero, che verrà restituito
- Un modo comune di combinare due interi, senza correre il rischio di andare in overflow, è rappresentato dall'or esclusivo (XOR) bit a bit, eseguito in Java dall'operatore "^"
- Quindi, per l'esempio in questione, si potrebbe procedere come segue:

```
public int hashCode() {
     return name.hashCode() ^ salary;
}
```

Mutabilità degli elementi

- Un altro problema che affligge sia TreeSet che HashSet riguarda la mutabilità degli elementi
- Gli elementi vengono inseriti in queste strutture dati in base al valore dei loro campi
  - Un TreeSet posiziona gli elementi sulla base di confronti con altri oggetti
  - Un HashSet li posiziona sulla base del loro codice hash

- Se un elemento viene modificato dopo essere stato inserito in una di queste strutture dati, il posizionamento di quell'elemento non corrisponderà più al valore dei suoi campi
- In questo caso, la struttura dati si comporterà in modo inatteso

- Ad esempio, supponiamo che la classe Employee ridefinisca sia equals sia hashCode, in modo che operino sulla stringa che rappresenta il nome dell'impiegato

Inseriamo l'impiegato x in un HashSet:

```
Set<Employee> s = new HashSet<Employee>();
Employee x = new Employee(“Mario”);
s.add(x);
```

se il nome di x viene modificato:

```
x.setName(“Luigi”);
```

una successiva chiamata a s.contains(x) potrebbe restituire “false”!

Esercizio (esame 19/2/2009)

Si implementi la classe `Container`, che rappresenta un contenitore per liquidi di dimensione fissata.

Ad un contenitore, inizialmente vuoto, si può aggiungere acqua con il metodo `addWater`, che prende come argomento il numero di litri.

Il metodo `getAmount` restituisce la quantità d'acqua presente nel contenitore.

Il metodo `connect` prende come argomento un altro contenitore, e lo collega a questo con un tubo. Dopo il collegamento, la quantità d'acqua nei due contenitori (e in tutti quelli ad essi collegati) sarà la stessa.

Esempio d'uso (l'output è sulla slide successiva):

```
Container a=new Container(), b=new Container(), c=new Container(), d=new Container();

a.addWater(12);
d.addWater(8);
a.connect(b);
System.out.println(a.getAmount()+" "+b.getAmount()+" "+c.getAmount()+" "+d.getAmount());
b.connect(c);
System.out.println(a.getAmount()+" "+b.getAmount()+" "+c.getAmount()+" "+d.getAmount());
c.connect(d);
System.out.println(a.getAmount()+" "+b.getAmount()+" "+c.getAmount()+" "+d.getAmount());
```

Esercizio (esame 19/2/2009)

Output dell'esempio d'uso:

```
6.0  6.0  0.0  8.0
4.0  4.0  4.0  8.0
5.0  5.0  5.0  5.0
```

L'interfaccia List e le sue implementazioni

- List rappresenta una sequenza di elementi, ovvero un vettore
- La figura a destra rappresenta l'interfaccia List e le sue implementazioni
- A differenza di Set, l'interfaccia List aggiunge alcuni metodi a Collection
- Qui, presentiamo solo i due metodi responsabili per l'**accesso posizionale**
  - get(int i) restituisce l'elemento al posto i-esimo della sequenza; solleva un'eccezione se l'indice è minore di zero o maggiore o uguale di size()
  - set(int i, E elem) sostituisce l'elemento al posto i-esimo della sequenza con elem; restituisce l'elemento sostituito; come get, solleva un'eccezione se l'indice è scorretto

Nota bene: non si può usare set per allungare una lista

- Analizzeremo le due principali classi che implementano List: **LinkedList** ed **ArrayList**

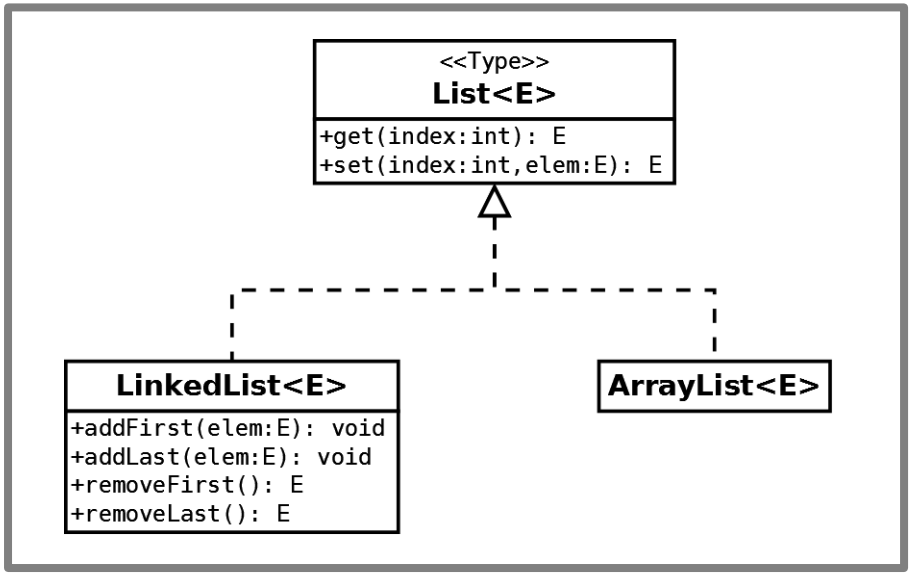


Figura 1: L'interfaccia List e le classi che la implementano.

La classe LinkedList

- La versione grezza di LinkedList è stata già presentata nella lezione dedicata agli iteratori
- Ricordiamo solamente che essa offre i seguenti metodi, in aggiunta a quelli di List, qui presentati nella loro versione parametrica

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

- Questi 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 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**

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

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*)
  - difatti, per accedere all'elemento di posto n è necessario scorrere la lista, a partire dalla testa, o dalla coda, fino a raggiungere la posizione desiderata
- 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 ArrayList

- Il fatto che l'accesso posizionale sia indicato per ArrayList e sconsigliato per LinkedList è anche segnalato dal fatto che, delle due, solo ArrayList implementa l'interfaccia **RandomAccess**
- RandomAccess è una interfaccia “di tag”, come Cloneable
  - ovvero, è una interfaccia vuota
  - serve a segnalare che la classe che la implementa offre l'accesso posizionale in maniera efficiente

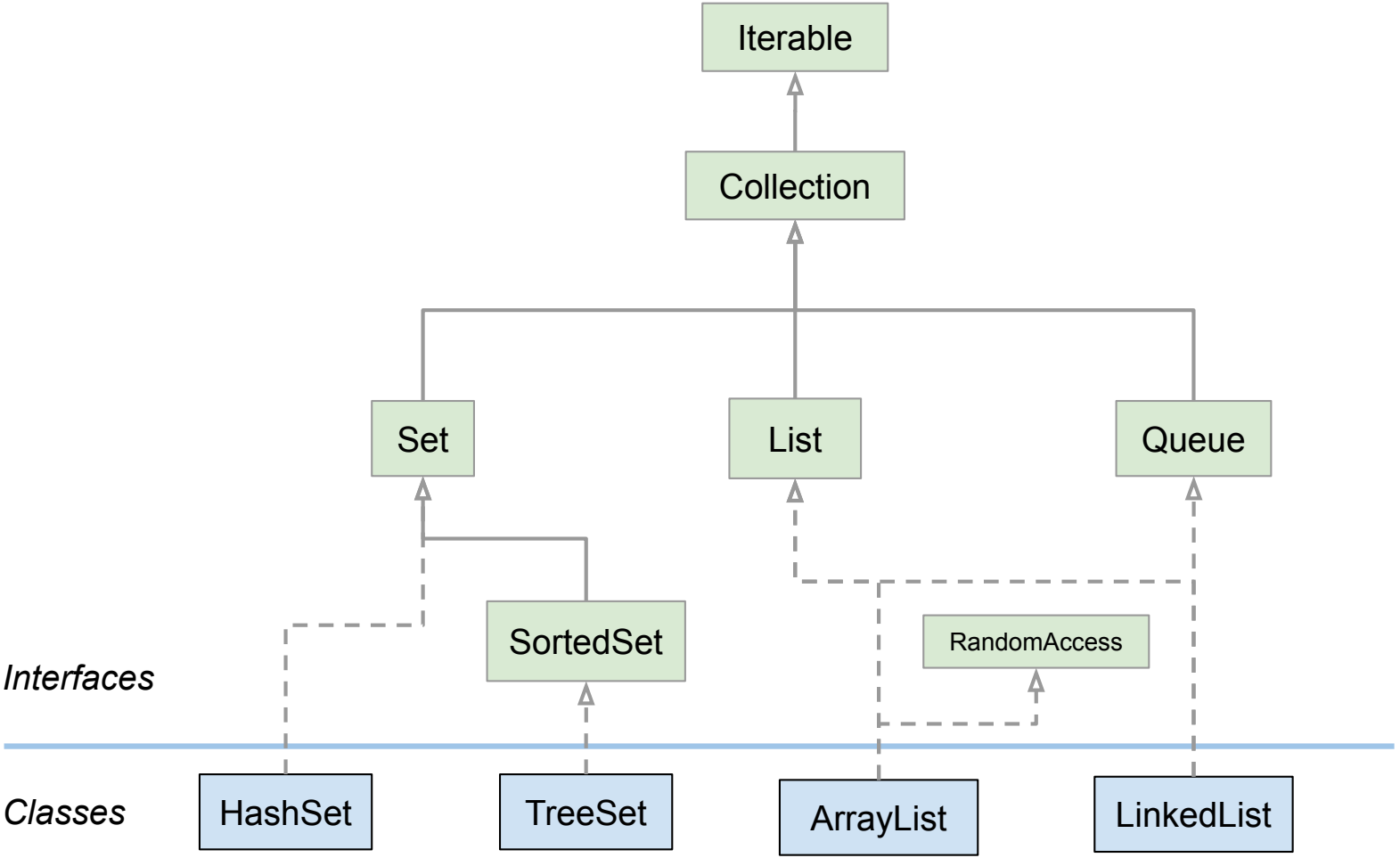
La seguente tabella riassume la complessità computazionale dei principali metodi delle liste

Metodo	Complessità in LL	Complessità in AL
add	O(1)	O(1)*
remove	O(n)	O(n)
contains	O(n)	O(n)
get, set	**O(n)**	**O(1)**
addFirst, addLast, removeFirst, removeLast	O(1)	-

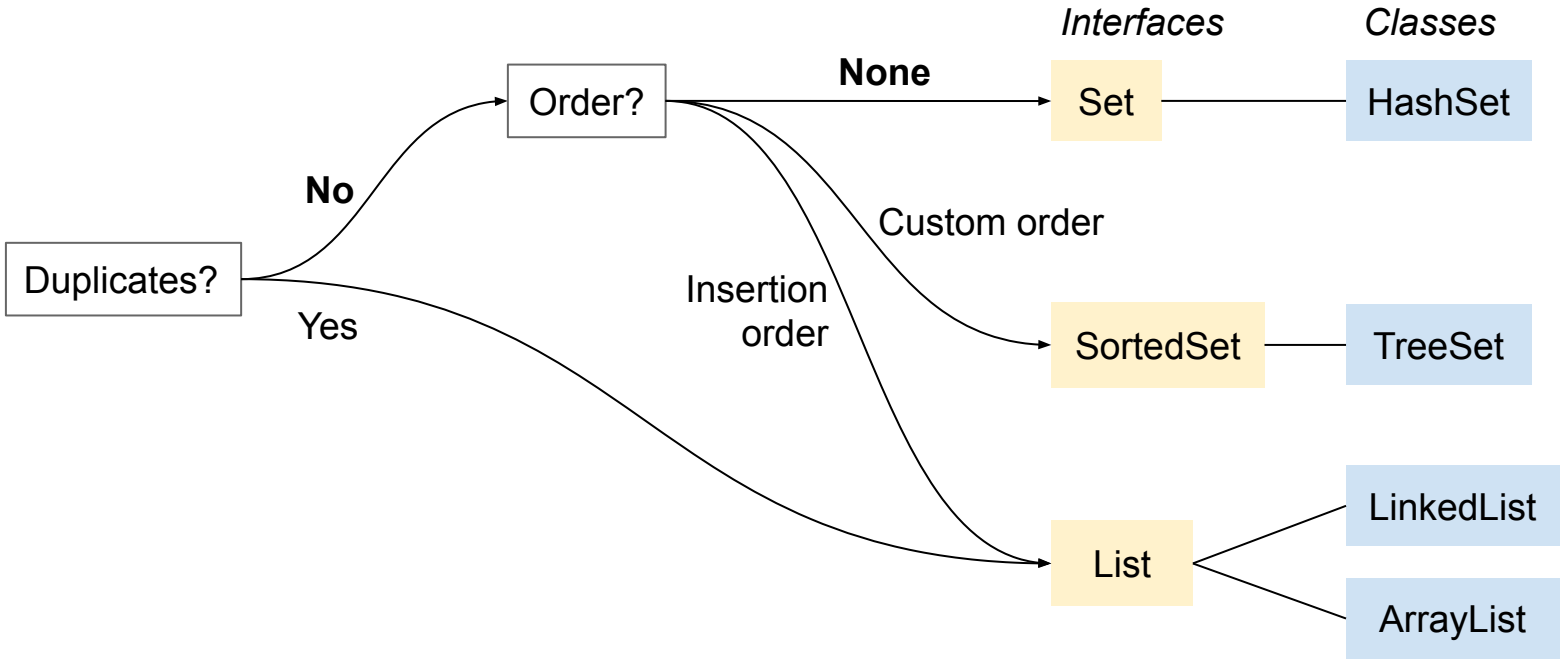
Note:

- (*) complessità ammortizzata
- add aggiunge in coda
- remove deve trovare l'elemento prima di rimuoverlo

Collezioni: diagramma riassuntivo



Collezioni: scegliere una collezione



Tratto da: Seriously Good Software, M. Faella, Manning 2020