JAVA COLLECTIONS

Pietro Di Lena

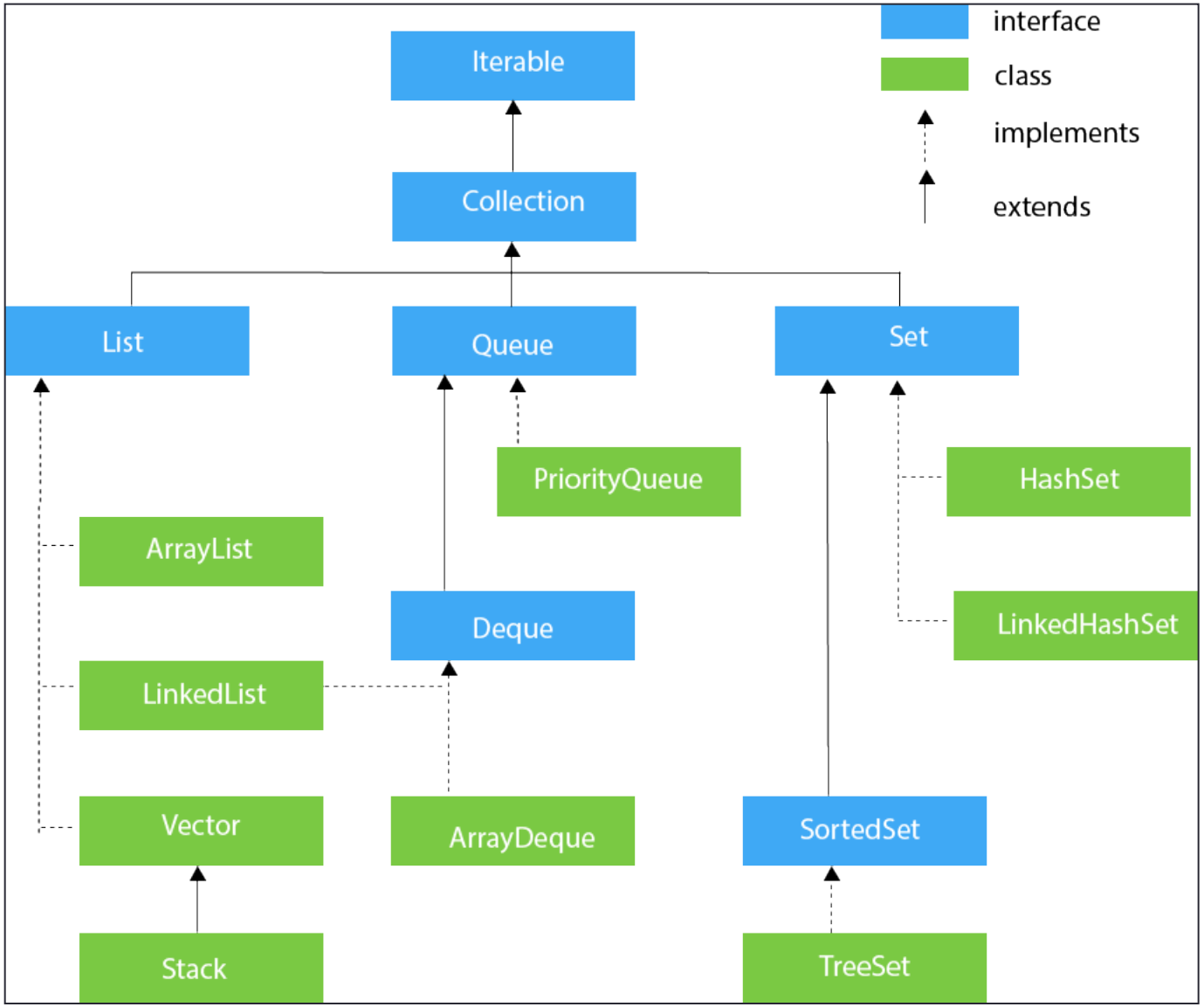
Università di Bologna

Credits: basate su slide create dal Prof. Angelo Di Iorio

Collezioni

- Una collezione Java è una classe che contiene un gruppo di oggetti
- *"The Java collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details"*
- Java Collections Framework include:
  - **Interfacce**: definiscono operazioni su collezioni e strutture dati
  - **Implementazioni**: classi che implementano le interfacce e forniscono strutture dati utilizzabili direttamente (e/o ottimizzate per scopi specifici)
  - **Operazioni**: implementazione di algoritmi comuni sulle collezioni, ad esempio ricerca o ordinamento

Java Collections Core



Java Collections

- Java Collections Framework include molte altre interfacce e classi (astratte e concrete) che espongono/forniscono comportamenti specifici, ad esempio:
  - `NavigableSet, NavigableMap`: per permettere navigazione/attraversamento
  - `BlockingQueue, TransferQueue, ConcurrentMap, DelayQueue`, etc.: per gestire accesso concorrente
- l'interfaccia per i dizionari (`Map<K,V>`) non è derivata da `Collection` ma ci sono diversi punti di contatto con tra dizionari e collezioni
- Documentazione completa: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/

Iterable e Iterator

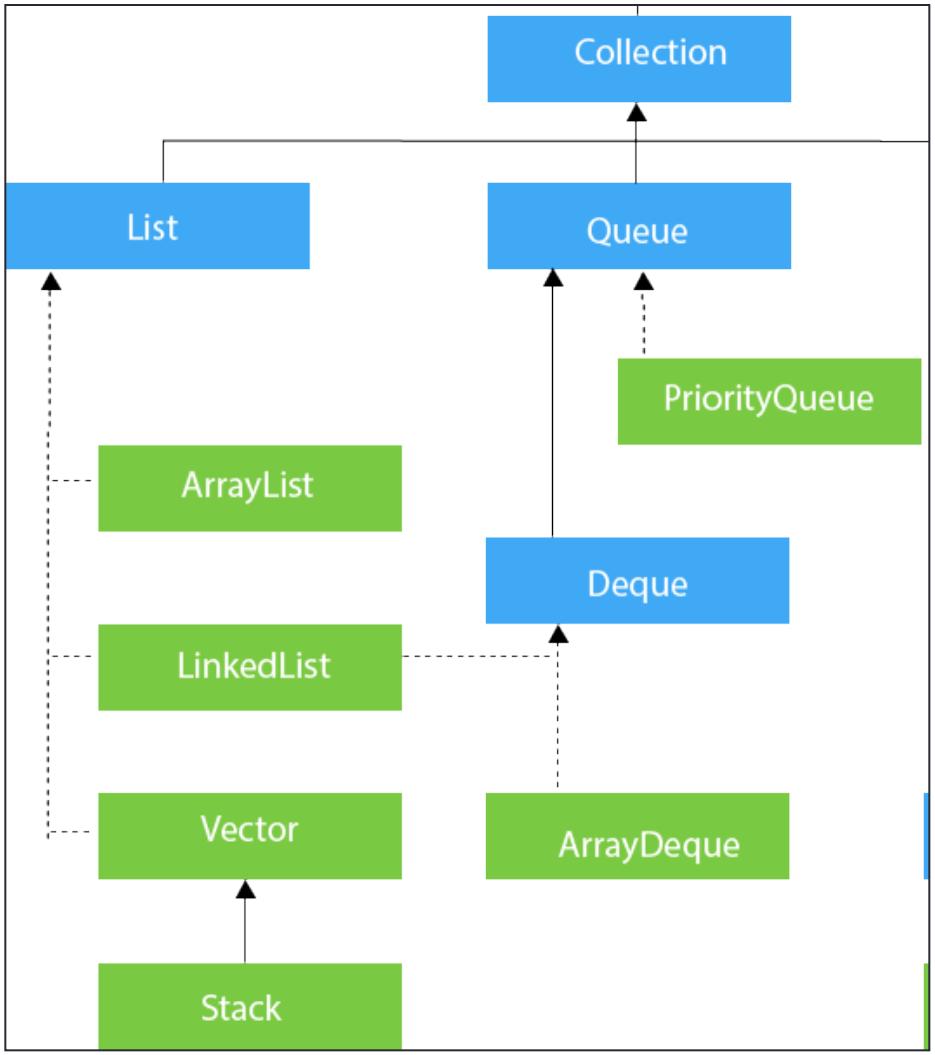
- Un **iteratore** è un oggetto che rappresenta un cursore con cui scandire una collezione di oggetti
- In Java gli iteratori implementano l'interfaccia `Iterator<E>` i cui metodi principali sono:
  - `boolean hasNext()`: verifica se c'è un altro elemento su cui iterare
  - e `next()`: ritorna l'elemento successivo
- Una classe che implementa l'interfaccia `Iterable` ha un metodo `iterator()` che restituisce un iteratore sugli elementi interni alla classe stessa

Iterator e for-each

```
ArrayList<Integer> integers = new ArrayList<>();
integers.add(5);
integers.add(10);

int somma = 0;

Iterator<Integer> integersIterator = integers.iterator();

while (integersIterator.hasNext()) {
        somma = somma + integersIterator.next();
}
System.out.println("Somma: " + somma);

somma = 0;
for (Integer i : integers) {
        somma = somma + i;
}
```

classe `DemoIterators`

Collections: Liste e Code (e Pile)



Implementazioni general-purpose

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Interfaccia List

- L'interfaccia `List<E>` definisce una **lista ordinata di oggetti (eventualmente duplicati)**
- Estende `Collection<E>` ed espone i metodi per aggiungere, cancellare, accedere agli elementi della lista
  - `boolean containsAll(Collection<?> c);`
  - `boolean add(E e);`
  - `void add(int index, E element); //posizione`
  - …
- Implementata da:
  - `ArrayList`
  - `LinkedList`
  - `Vector` (retrocompatibilità)
  - `Stack` (retrocompatibilità)

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

public class DemoLists {
  public static void main(String[] args) {

    List<Integer> li1 = new ArrayList<Integer>();

    li1.add(1); li1.add(4); li1.add(6); li1.add(3);

    System.out.println(li1.get(2));
    System.out.println(li1);

    List<String> ls2 = new LinkedList<String>();

    ls2.addAll(Arrays.asList("ciao","hello","hallo","hola"));

    System.out.println(ls2);
  }
}
```

classe DemoLists

ArrayList e LinkedList

- **ArrayList**: dati memorizzati in un vettore dinamico, di cui si può settare la capacità iniziale e che viene ri-dimensionato a run-time
  - Veloce accesso all'elemento i-esimo (indice vettore)
  - Dispendiosa aggiunta e rimozione di elementi (shift)

- **LinkedList**: dati memorizzati in una lista concatenata bidirezionale
  - Dispendioso accedere all'elemento i-esimo (scan della lista)
  - Veloce aggiungere/rimuovere (aggiornamento puntatori)

ArrayList vs LinkedList

- Scriviamo un metodo per calcolare la somma di tutti gli interi in una lista
- Costo computazionale?

```
private static int getSum(LinkedList<Integer> l) {

	int sum = 0;

	for (int i = 0; i < l.size(); i++) {
		sum += l.get(i);
	}

	return sum;
}
```

ArrayList e vettori

- La classe `ArrayList` è implementata tramite un vettore che è ri-dimensionato durante l'esecuzione del programma
- Due punti da tenere a mente
  - `ArrayList` è meno efficiente di un array
  - `ArrayList` memorizza oggetti e non tipi primitivi
    - Boxing/unboxing
- Java include ancora la classe `Vector` che è molto simile ad `ArrayList` ma usata in versioni precedenti del linguaggio
  - L'implementazione di `ArrayList` è più efficiente
    - `Vector` è sincronizzato mentre `ArrayList` no
    - `Vector` alloca più spazio quando aumenta le dimensioni del vettore interno

Interfaccia Deque (Double-Ended Queue)

- L'interfaccia `Deque<E>` definisce una **coda ordinata di oggetti (eventualmente duplicati) su cui è possibile fare operazioni sia in testa che in coda.**
- Estende `Queue<E>` ed espone i metodi per aggiungere, cancellare, accedere agli elementi della lista
  - `boolean containsAll(Collection<?> c);`
  - `boolean add(E e);`
  - `boolean addLast(E element); //coda`
  - `E removeLast(); //coda`
  - …
- Implementata da:
  - `LinkedList`
  - `ArrayDeque`

```
public class DemoQueue {
  public static void main(String[] args) {
    ArrayDeque<Integer> queue = new ArrayDeque<Integer>();

    // come addLast() - FIFO
    queue.add(1); queue.add(4); queue.add(6); queue.add(3);
    // come removeFirst() - FIFO
    queue.remove();
    System.out.println(queue);

    LinkedList<Integer> listqueue = new LinkedList<Integer>();

    listqueue.addFirst(1); listqueue.addFirst(4);
    listqueue.addLast(6);  listqueue.addFirst(3);
    listqueue.removeLast();
    System.out.println(listqueue);
  }
}
```

classe DemoQueue

Esercizio

- Usare le classi Java `LinkedList` e `ArrayDeque` per eseguire in ordine le seguenti operazioni su una sequenza (lista/coda) di interi inizialmente vuota:
- Caso 1: *`add(5), add(3), remove(), add(2), add(8), remove(), remove(), add(4)`*
- Caso 2: *`addFirst(3), addLast(8), addFirst(2), removeLast( ), addLast(7) addLast(4), removeFirst( ), removeFirst( )`*

- Qual è il risultato finale?

classe `DemoListQueue`