INTRODUZIONE A JAVA COLLECTIONS

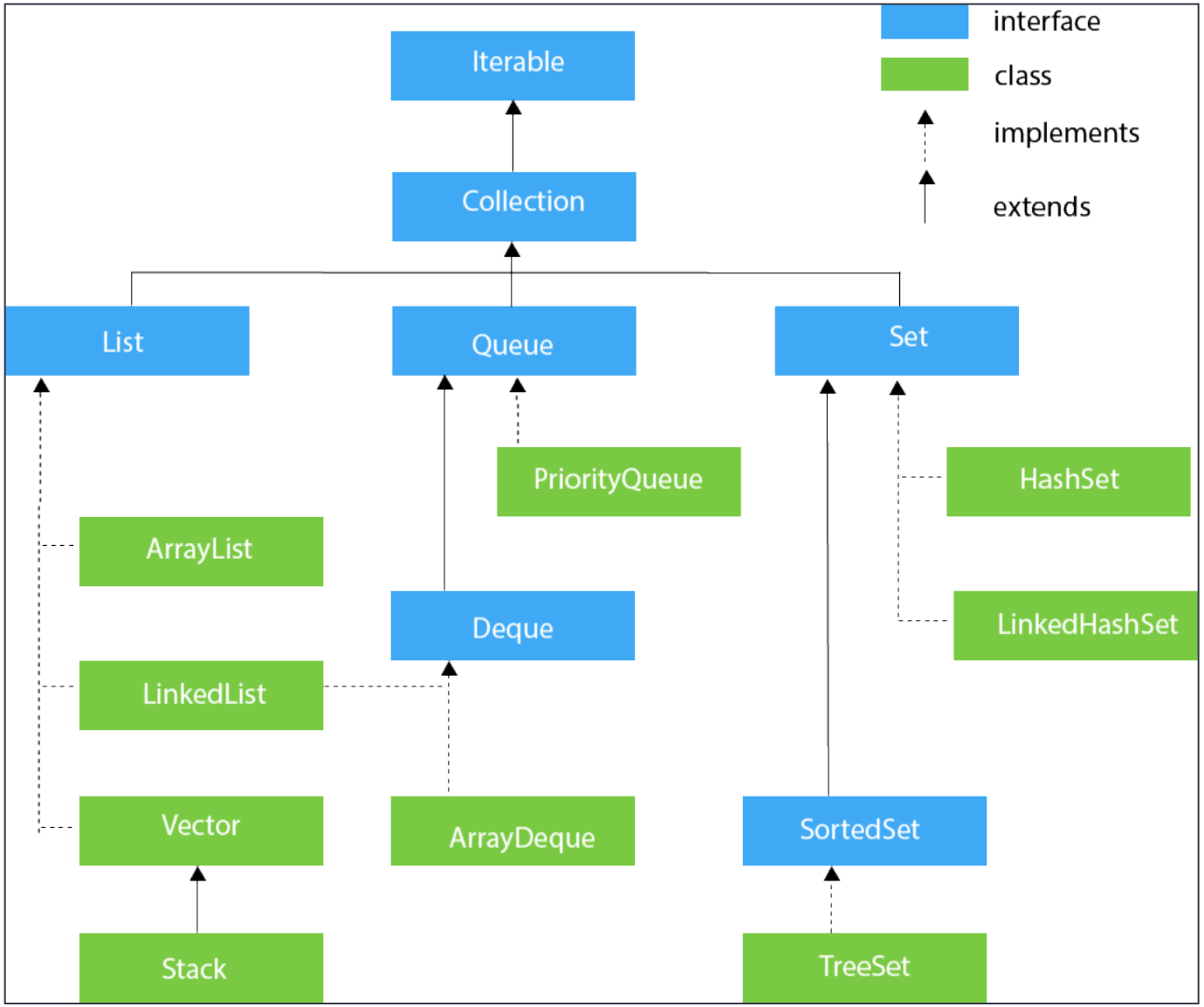
Angelo Di Iorio

Università di Bologna

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
- Inoltre 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
- API: https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Collection.html

Liste

- Una lista è una struttura dati che permette di memorizzare dati in maniera sequenziale, e permette di accedere e cancellare dati in posizioni arbitrarie
- Ci sono varie operazioni che si possono fare su una lista:
  - verifica se è vuota
  - trovare l'elemento in una certa posizione
  - inserire un elemento in una certa posizione
  - eliminare l'elemento in una certa posizione
  - …
- Come visto nelle precedenti lezioni, una lista può essere realizzata usando diverse strutture dati
  - Vettori
  - Puntatori (monodirezionali, bidirezionali, circolari, etc.)

Interfaccia List in Java Collections

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

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

public class ListsDemo {

	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);
```

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 bidirezionale
  - Dispendioso accedere all'elemento i-esimo (scan della lista)
  - Veloce aggiungere/rimuovere (aggiornamento puntatori)

ArrayList e LinkedList

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

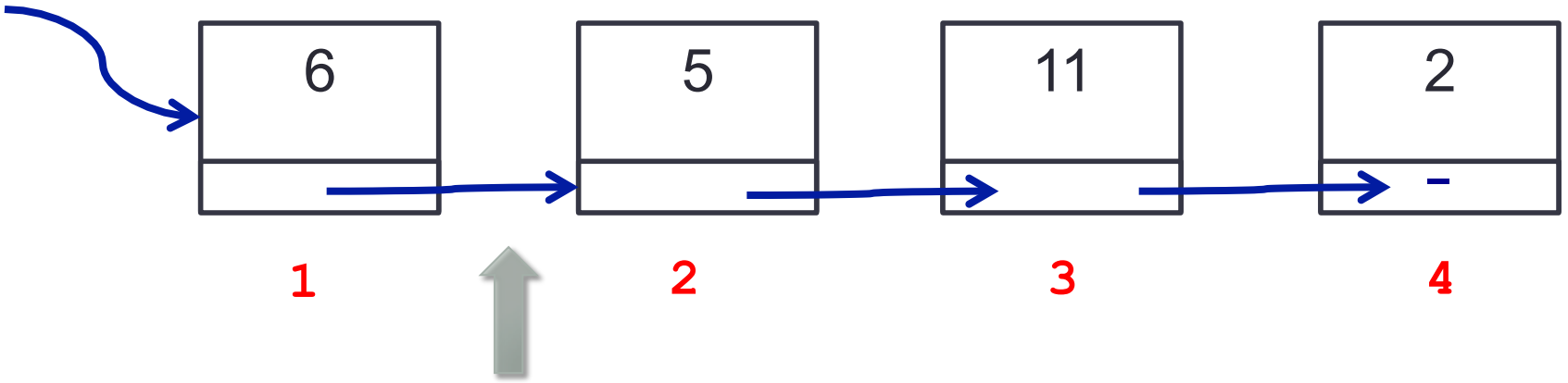
```
private static int getSum(LinkedList<Integer> l){

        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
- Alcuni punti da tenere a mente
  - `ArrayList` non ha dimensioni fisse a differenza di un array
  - `ArrayList` è meno efficiente di un array
  - `ArrayList` può memorizzare solo oggetti e non tipi primitivi
    - Boxing/unboxing
- Java include ancora la classe `Vector` che è molto simile ad `ArrayList`
- `Vector` è sincronizzato (thread-safe) mentre `ArrayList` no

Scandire gli elementi di una lista

- Esistono vari modi per scandire una lista, più o meno appropriati in base al problema da risolvere:
  - ciclo **for** con un contatore in base alla dimensione della lista
  - ciclo **for-each** su tutti gli elementi della lista (in generale su tutti gli elementi di una collezione)
  - **Iteratore.** *For-each* usa implicitamente un Iteratore anche se non esposto direttamente all'utente



for e for-each

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

int somma = 0;


for (int i = 0; i < integers.size(); i++) {
        somma = somma + integers.get(i);
}


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

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
- L'interfaccia `Iterable` (da cui deriva `Collection`) espone un metodo `iterator()` che restituisce un iteratore
- Su un'istanza di una classe che implementa `Iterable` (come `ArrayList` o `LinkedList`) è possibile quindi ottenere un iteratore

Iteratore

```
int somma = 0;

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

while (integersIterator.hasNext()) {
       somma = somma + integersIterator.next();
}

System.out.println("Somma: " + somma );
```

Esercizio

- Scrivere i metodi per calcolare, data una lista di interi (di tipo *List<Integer>*):
  A. Somma degli interi pari
  B. Somma degli interi in posizione pari (secondo, quarto, sesto, etc.)

- Provare con *for*, *for-each* e *Iterator*

Esercizio

- Scrivere un metodo che prende in input una lista di interi (di tipo *List<Integer>*) e **rimuove gli interi pari**
  - modifica direttamente la lista e ritorna `void`

- Il metodo `remove()` dell'interfaccia `Iterator<E>` permette di rimuovere l'elemento corrente

- Provare con *for*, *for-each* e *Iterator*
- Qualcosa non va?

Iterator<E> e ListIterator<E>

- Ad ogni passo un iteratore è posizionato tra due elementi della lista, in modo simile al cursore tra due caratteri di un testo
- L'interfaccia `Iterator<E>` non espone un metodo per aggiungere elementi
- L'interfaccia `ListIterator<E>` estende `Iterator<E>` e permette di:
  - attraversare una lista in entrambe le direzioni
    - metodi `hasPrevious()` e `previous()`
  - aggiungere e sostituire l'elemento "corrente"
    - metodo `add()`
- Il metodo `List.listIterator()` permette di istanzare un `ListIterator<E>` da una lista, come il corrispondente `.iterator()`

ListIterator<E>

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).

Esercizio

- Scrivere un metodo che prende in input una lista di interi (di tipo *List<Integer>*) e **duplica gli interi dispari**
  - modifica direttamente la lista e ritorna `void`

- Esempio
  - L = 4, 7, 6, 3, 2
  - L = 4, 7, 7, 6, 3, 3, 2

- Provare con i diversi cicli e iteratori

Esercizio

- Scrivere un metodo che prende in input una lista di interi (di tipo *List<Integer>*) e **rimuove gli interi pari e replica ogni intero dispari tante volte quanti sono i pari che lo precedono**
  - modifica direttamente la lista e ritorna `void`

- Esempio
  - L = 4, 6, 7, 3, 2, 5
  - L = 7, 7, 7, 3, 3, 3, 5, 5, 5, 5

Esercizio

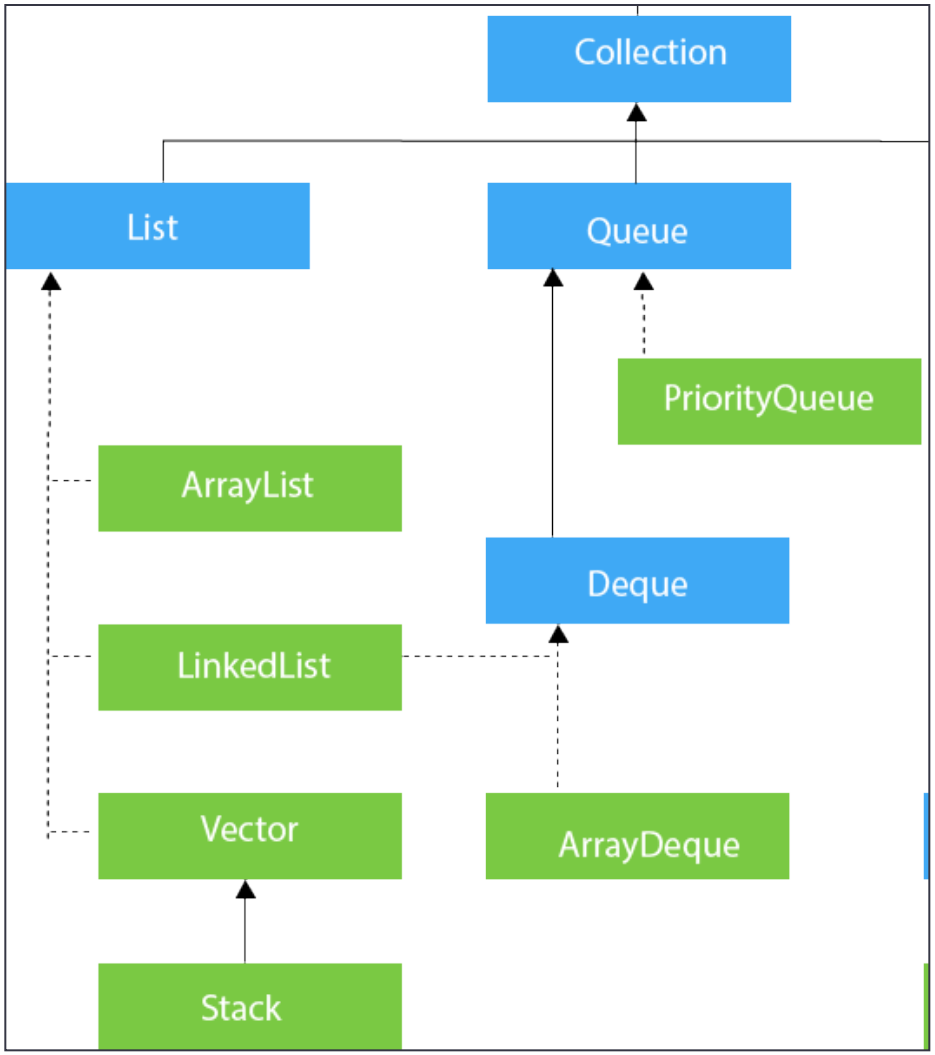
Scivere inoltre i metodi per:

A. Stampare i valori in una lista dall'ultimo al primo
B. Invertire ordine degli elementi (il primo diventa l'ultimo, il secondo il penultimo, etc.)

- Come negli esercizi precedenti i metodi prendono in input una lista di interi (tipo `List<Integer>`) e ritornano `void`

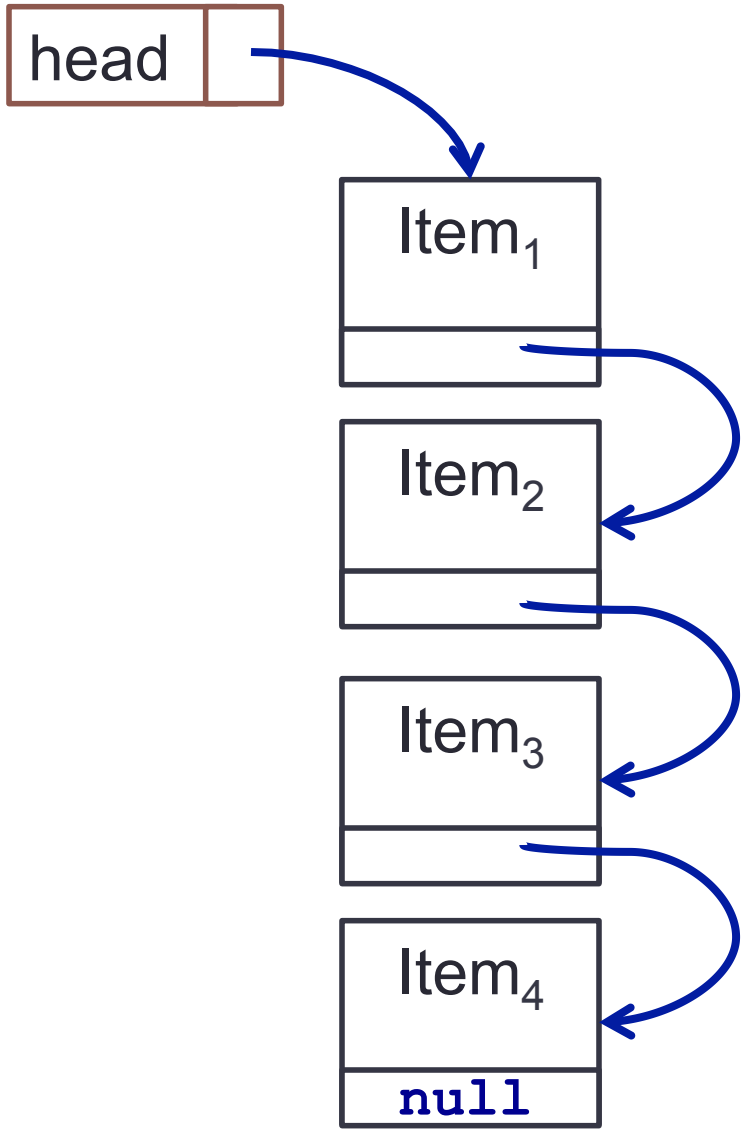
- Provare con i diversi cicli e iteratori

Collections: Liste e Code (e Pile)



Pile – LIFO – con puntatori

```
public interface IStack<T> {

   public boolean isEmpty();

   public void push(T item);

   public T pop();

   public T top();
}
```



Pile - LIFO

```
public class MyStack<T> implements IStack<T> {
	//implementazione in MyStack omessa
```

```
public class Book {

	private String title;

	public Book(String title) {
		this.title = title;
	}

	@Override
	public String toString() {
		return "Book [title=" + title + "]";
	}
}
```

```
public class BookDemo {

	public static void main(String[] args) {

		Book b1 = new Book("Il nome della Rosa");
		Book b2 = new Book("The Da Vinci Code");
		Book b3 = new Book("Outliers");
		Book b4 = new Book("The Client");

		MyStack<Book> pilaLibri = new MyStack<Book>();

		pilaLibri.push(b1);
		pilaLibri.push(b2);
		pilaLibri.push(b3);
		pilaLibri.pop();
		pilaLibri.push(b4);
		pilaLibri.pop();

		System.out.println(pilaLibri.top());
	}
}
```

Coda – Queue - FIFO

Double-Ended Queque

Interfaccia Deque (Double-Ended Queue)

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

```
public class DemoBookDeque {

	public static void main(String[] args) {

		Book b1 = new Book("Il nome della Rosa");
		Book b2 = new Book("The Da Vinci Code");
		Book b3 = new Book("Outliers");
		Book b4 = new Book("The Client");

		Deque<Book> pilaLibri = new ArrayDeque<Book>();

		pilaLibri.addFirst(b1);
		pilaLibri.addFirst(b2);
		pilaLibri.addFirst(b3);
		pilaLibri.pop();
		pilaLibri.addFirst(b4);
		pilaLibri.pop();

		System.out.println(pilaLibri.peek());
	}
}
```

```
public class BookDemo {

        public static void main(String[] args) {

                Book b1 = new Book("Il nome della Rosa");
                Book b2 = new Book("The Da Vinci Code");
                Book b3 = new Book("Outliers");
                Book b4 = new Book("The Client");

                Deque<Book> pilaLibri = new ArrayDeque<Book>();

                pilaLibri.add(b1);
                pilaLibri.add(b2);
                pilaLibri.addFirst(b3);
                pilaLibri.pop();
                pilaLibri.addFirst(b4);
                pilaLibri.pop();

                System.out.println(pilaLibri.top());
        }
}
```

```
ArrayDeque<Integer> queue = new ArrayDeque<Integer>();

queue.add(1); // come addLast() - FIFO
queue.add(4);
queue.add(6);
queue.add(3);

System.out.println(queue);

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

listqueue.addFirst(1); // LIFO
listqueue.addFirst(4);
listqueue.addLast(6);
listqueue.addFirst(3);

System.out.println(listqueue);

listqueue.removeLast();
```

Esercizio

- Usare le classi Java `LinkedList` e `ArrayDeque` per eseguire in ordine le seguenti operazioni su una collezione di interi inizialmente vuota:
- A: *`enqueue(5), enqueue(3), dequeue(), enqueue(2), enqueue(8), dequeue(), dequeue(), enqueue(4)`*
- B: *`addFirst(3), addLast(8), addFirst(2), removeLast( ), addLast(7) addLast(4), removeFirst( ), removeFirst( )`*
- Qual'è il risultato finale nei due casi precedenti?

Esercizio

- Scrivere il codice Java di un metodo che prende in input una pila di interi (`ArrayDeque<Integer>`) ed elimina dalla pila i valori pari
  - restituisce `void`.
- L'ordine degli elementi deve rimanere invariato ed è possibile eseguire operazioni solo in testa alla pila
- E' ammesso usare strutture dati ausiliarie