



Collections



Collections in Java

- Una **collection** è un oggetto che raggruppa elementi multipli in una singola entità
 - A differenza degli array, la dimensione massima della collection non è prefissata
- Le collection sono usate per immagazzinare, recuperare e trattare dati, e per trasferire gruppi di dati da un metodo ad un altro
- Utili per rappresentare gruppi di dati, es.
 - una mano di poker (collection di carte)
 - un mail folder (collection di e-mail)
 - un elenco telefonico (collection di associazioni nome-numero)

Java Collection Framework

- Contiene tre elementi fondamentali:
 - **interfacce**: specificano insiemi di servizi associati a diversi tipi di collection, potenzialmente associate a diverse strutture dati
 - **implementazioni** di specifiche strutture dati di uso comune, che implementano le interfacce di cui sopra
 - **algoritmi**, codificati in metodi, implementano operazioni comuni a più strutture dati
 - ♦ Es., ricerca, ordinamento, mescolamento (*shuffling*), composizione
 - ♦ lo stesso metodo può essere usato in diverse implementazioni

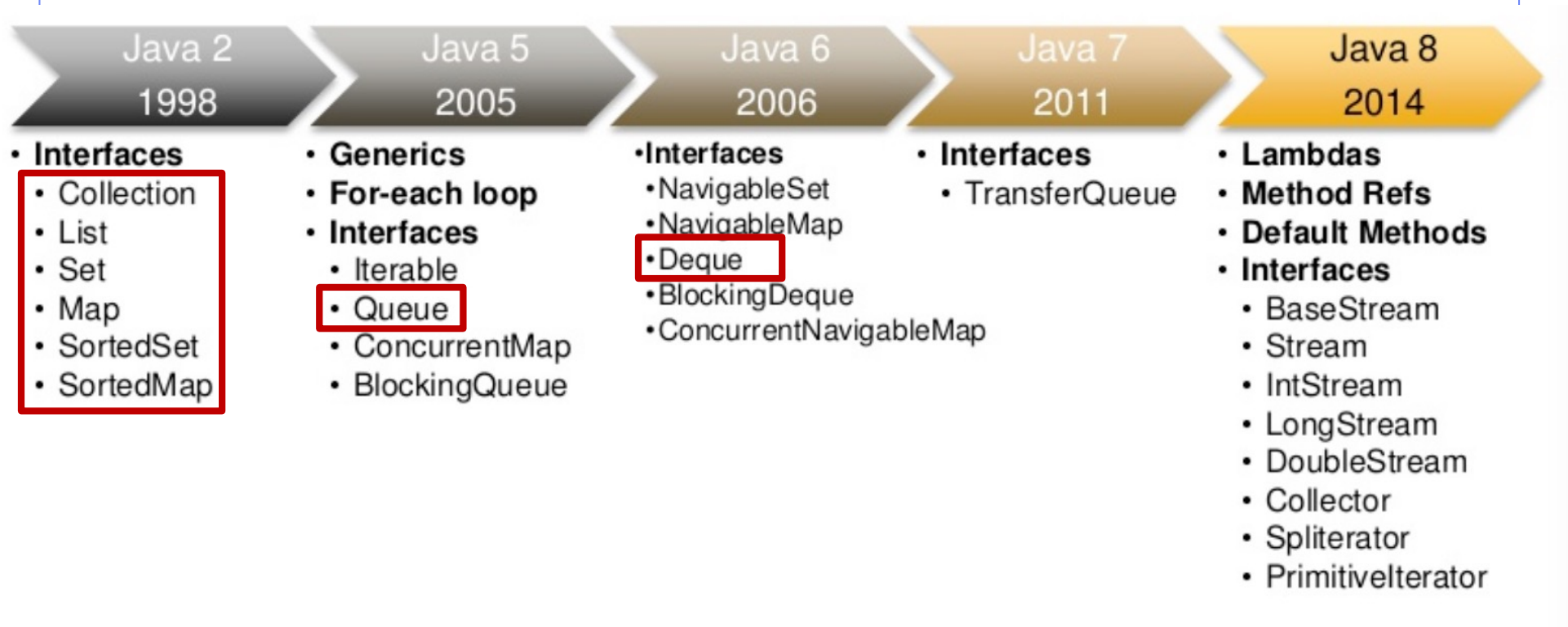
Collections e linguaggio

- Il Java Collections Framework è un punto di forza di Java ...
- ... perchè, a differenza di altri linguaggi, è parte integrante del linguaggio, invece di essere fornito a parte
 - Nota: il *framework* esiste dalla versione 1.2 del linguaggio; tuttavia, nelle versioni precedenti erano già presenti strutture dati di uso comune
- Ciò porta una serie di vantaggi, che si sommano a quelli generici derivanti dall'uso di una libreria

Collections in Java: vantaggi

- Riduce il lavoro del programmatore, ne aumenta la produttività e migliora la qualità del codice prodotto
 - Il programmatore si può concentrare sull'applicazione invece che sullo sviluppo di strutture dati efficienti
 - Le strutture dati sono affidabili e ottimizzate direttamente dagli autori del linguaggio
- Permette interoperabilità tra API (*Application Programming Interface*) diverse e ne semplifica l'apprendimento e comprensione
 - Fornendo una definizione comune delle strutture dati
- Facilita il riuso di codice
 - Evitando la duplicazione di funzionalità, consentendone al contempo l'adattamento ove necessario

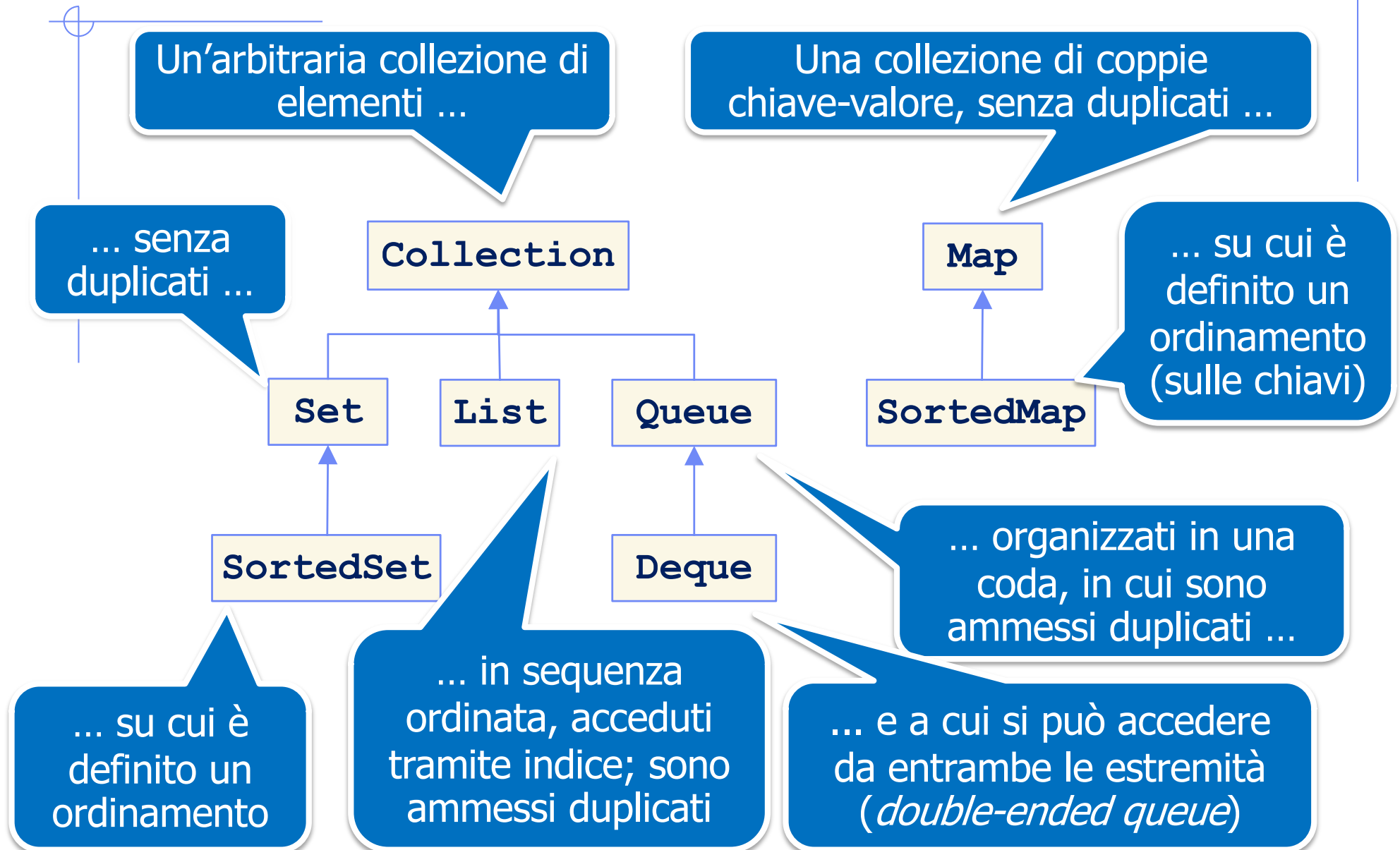
Un po' di storia...



Per usare il Java Collections Framework:

```
import java.util.*;
```

Interfacce (*core*)



Implementazioni (alcune)

Implementazioni
«*general purpose*»
più comuni. Ve ne
sono molte altre...

		classi (implementazioni)			
		hash table	resizable array	balanced tree	linked list
interfacce	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Collection: operazioni base

`int size();`

`boolean isEmpty();`

`boolean contains(Object element);`

`boolean add(Object element);`

`boolean remove(Object element);`

`Iterator iterator();`

ritornano **true** se la collection è cambiata a seguito dell'operazione

senza generics
(< Java 5)

Collection<E>: operazioni base

`int size();`

`boolean isEmpty();`

`boolean contains(Object element);`

`boolean add(E element);`

`boolean remove(Object element);`

`Iterator<E> iterator();`

ritornano **true** se la collection è
cambiata a seguito dell'operazione

con generics
(Java 5+)

Collection<E>: *bulk operations*

- Consentono di effettuare operazioni su più elementi contemporaneamente

```
boolean containsAll(Collection<?> c);  
boolean addAll(Collection<? extends E> c);  
boolean removeAll(Collection<?> c);  
boolean retainAll(Collection<?> c);  
void clear();
```

con generics
(Java 5+)

List<E>: operazioni base (aggiunte)

rimpiaccia l'elemento
in posizione `index`

E `get(int index);`

E `set(int index, E element);`

`void add(int index, E element);`

E `remove(int index);`

`int indexOf(Object o);`

`int lastIndexOf(Object o);`

inserisce un elemento in posizione `index`, spostando in
avanti quello attuale e successivi;
la `boolean add(E element)` ereditata da `Collection`
invece appende l'elemento in fondo alla lista

con generics
(Java 5+)

Ancora pile e code...

```
package strutture;
import java.util.*;
public abstract class VettoreDati<T> {
    List<T> contenuto;
    int marker;
    public VettoreDati() {
        marker = 0;
        contenuto = new LinkedList<T>();
    }
}
```

potrei usare `ArrayList<T>`
senza modifiche
al resto del codice

Ancora pile e code...

```
final public void inserisci(T k) {  
    contenuto.add(marker,k);  
    marker++;  
}  
abstract public T estrai();
```

```
public T estrai() {  
    assert(marker>0) : "Estrazione da Pila vuota";  
    return contenuto.get(--marker);  
}
```

In **Pila**

Map: operazioni base

```
Object put(Object key, Object value);  
Object get(Object key);  
Object remove(Object key);  
boolean containsKey(Object key);  
boolean containsValue(Object value);  
int size();  
boolean isEmpty();  
Set keySet();  
Collection values();
```

senza generics
(< Java 5)

Map: un esempio

```
import java.util.*;

public class TestMap {
    public static void main(String[] args) {
        Map moto = new HashMap();
        moto.put("Ducati", 2); moto.put("BMW", 7);
        moto.put("Honda", 3); moto.put("Yamaha", 5);
        System.out.println("Numero marche in magazzino: " + moto.size());
        System.out.println("Elenco marche in magazzino: " + moto.keySet());
        System.out.print("Yamaha in magazzino: ");
        if(moto.containsKey("Yamaha")) System.out.println(moto.get("Yamaha"));
        else System.out.println("nessuna");
        System.out.print("Totale moto in magazzino: ");
        int tot = 0;
        for(Iterator i = moto.values().iterator(); i.hasNext(); )
            tot += (Integer) i.next();
        System.out.println(tot);
    }
}
```

oppure
for(Object i: moto.values())
tot += (Integer) i;

```
Numero marche in magazzino: 4
Elenco marche in magazzino: [Ducati, Yamaha, BMW, Honda]
Yamaha in magazzino: 5
Totale moto in magazzino: 17
```


Map<K, V>: operazioni base

V put(**K** key, **V** value) ;

V get(Object key) ;

V remove(Object key) ;

boolean containsKey(Object key) ;

boolean containsValue(Object value) ;

int size() ;

boolean isEmpty() ;

Set<**K**> keySet() ;

Collection<**V**> values() ;

con generics
(Java 5+)

Map<K, V>: un esempio

```
import java.util.*;

public class TestMap {
    public static void main(String[] args) {
        Map<String,Integer> moto = new HashMap<>();
        moto.put("Ducati", 2); moto.put("BMW", 7);
        moto.put("Honda", 3);  moto.put("Yamaha", 5);
        System.out.println("Numero marche in magazzino: " + moto.size());
        System.out.println("Elenco marche in magazzino: " + moto.keySet());
        System.out.print("Yamaha in magazzino: ");
        if(moto.containsKey("Yamaha")) System.out.println(moto.get("Yamaha"));
        else System.out.println("nessuna");
        System.out.print("Totale moto in magazzino: ");
        int tot = 0;
        for(Iterator i = moto.values().iterator(); i.hasNext(); )
            tot += (Integer) i.next();
        System.out.println(tot);
    }
}
```

oppure
for(Integer i: moto.values())
 tot += ~~(Integer)~~ i;

```
Numero marche in magazzino: 4
Elenco marche in magazzino: [Ducati, Yamaha, BMW, Honda]
Yamaha in magazzino: 5
Totale moto in magazzino: 17
```

Collection e array

- `Collection<E>` offre i metodi
`Object[] toArray()`
`<T> T[] toArray(T[] a)`
che consentono di ottenere il contenuto della collection in un array
 - Utili per interagire con API basate su array anziché collection
- Se `c` è di tipo `Collection`
 - `Object[] a = c.toArray();`
ritorna il contenuto di `c` in un array di `Object`
 - `String[] a = c.toArray(new String[0]);`
ritorna il contenuto di `c` in un array di `String`

Metodo generico!

Avvisi in compilazione

- È possibile compilare codice scritto per il Java Collection Framework senza generics con un compilatore Java 5+
 - Es., l'IDE usato nelle esercitazioni
- Tuttavia, in questo caso il compilatore emette dei *warning* ...
- ... ricordando al programmatore che sta aggirando i controlli di tipo previsti dalla API

```
warning: [unchecked] unchecked call to put(K,V) as a member of  
the raw type Map
```

ArrayList o LinkedList?

- Per una struttura relativamente stabile, (elementi contenuti che cambiano poco e occorre spesso accedervi) => ArrayList (supporta l'accesso casuale a tempo costante $O(1)$).
- Struttura in cui spesso si inseriscono e si cancellano elementi e il costo di query non è importante => LinkedList.
- In ogni caso, usare interfacce!

HashSet o TreeSet?

- Nello **HashSet** gli elementi non sono ordinati. `add`, `remove`, and `contains` methods hanno complessità costante **$O(1)$**
- Nel **TreeSet** gli elementi sono ordinati (come?). `add`, `remove`, and `contains` methods hanno complessità **$O(\log(n))$** .
- In ogni caso, usare interfacce!


In ogni caso, un Set non ammette elementi duplicati!

Un oggetto fittizio: Number

```
package structures;  
class Number {  
    private int n;  
  
    Number(int n) {  
        this.n = n;  
    }  
    int getInt() {  
        return n;  
    }  
    void setInt(int n) {  
        this.n = n;  
    }  
}
```

Vogliamo usarlo con una
Coda e una Pila,
riscritte con il Java
Collections Framework...

La classe base: ArrayDati



```
package structures;  
import java.util.*;  
public abstract class ArrayDati  
    extends LinkedList {  
    public void inserisci(int x) {  
        Number n = new Number(x);  
        this.add(n);  
    }  
    abstract public int estrai();  
}
```

Potrei
estendere
da
ArrayList
senza
modifiche al
resto del
codice

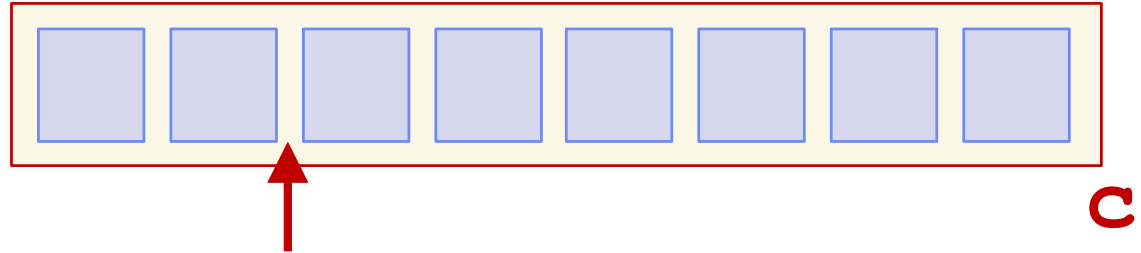
... e se avessi esteso da
HashSet?



Visita di una Collection



Iteratori



- A ogni oggetto di tipo `Collection<E>` è associato un oggetto di tipo `Iterator<E>`
`Iterator<Integer> i = c.iterator();`
- Consente di scandire gli elementi della collezione uno a uno

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

... true se ci sono ancora elementi da scandire

ritorna l'elemento successivo

elimina l'ultimo elemento ritornato da `next()`; invocabile una sola volta

Iteratori: esempio d'uso

```
void filter(Collection<E> x) {  
    Iterator<E> i = x.iterator();  
    while (i.hasNext()) {  
        if (!cond(i.next()))  
            i.remove();  
    }  
}
```

- Rimuove dalla collezione gli oggetti che non soddisfano una condizione data **cond** (definita altrove)
- Il codice è «polimorfico» nel senso che funziona per ogni collezione che supporta la rimozione di elementi, indipendentemente dalla specifica implementazione
 - Conseguenza (benefica) dell'ampio uso di interfacce e del polimorfismo legato ai tipi

Ciclo `for` ... oppure `for-each`?

- Usare gli iteratori tuttavia è macchinoso

```
for(Iterator<Element> i = c.iterator();  
    i.hasNext(); ) {  
    Element e = i.next();  
    ... // usa e  
}
```

- Da Java 5, è possibile usare il ciclo `for-each`
(«*enhanced for statement*»)



```
for(Element e : c) {  
    ... // usa e  
}
```

«for each `Element` `e` in `c`»

«Nasconde» gli
iteratori, chiamandoli
automaticamente ove
necessario:
più conciso e intuitivo

Un esempio

- Supponiamo di voler stampare tutte le combinazioni possibili nel lancio di due dadi

```
List<String> faces = new ArrayList<>();  
faces.add("ONE"); ... ; faces.add("SIX");
```

- Con gli iteratori:

```
for(Iterator<String> i = faces.iterator(); i.hasNext();)  
    for(Iterator<String> j = faces.iterator(); j.hasNext();)  
        System.out.println(i.next() + " " + j.next());
```

Problema: ambedue gli iteratori
vengono avanzati simultaneamente

ONE ONE
TWO TWO
THREE THREE
FOUR FOUR
FIVE FIVE
SIX SIX

Un esempio

- Una possibile soluzione:

```
for(Iterator<String> i = faces.iterator(); i.hasNext();) {  
    String s = i.next();  
    for(Iterator<String> j = faces.iterator(); j.hasNext();) {  
        System.out.println(s + " " + j.next());  
    }  
}
```

- Oppure, con "for-each":

```
for(String i : faces)  
    for(String j : faces)  
        System.out.println(i + " " + j);
```

Ambedue le soluzioni sono corrette

La seconda è più leggibile e intuitiva

```
ONE ONE  
ONE TWO  
ONE THREE  
ONE FOUR  
ONE FIVE  
ONE SIX  
  
... ..  
SIX FOUR  
SIX FIVE  
SIX SIX
```

Quando è possibile usare for-each?

```
int[] f = {1,2,3,4,5,6};
```

```
for(int i : f)
```

```
    for(int j : f)
```

```
        System.out.println(i + " " + j);
```

Funziona anche con gli array!

1	1
1	2
1	3
...	...
6	4
6	5
6	6

- Si può usare for-each con gli array e con ogni oggetto che implementa l'interfaccia **Iterable**
- Tuttavia, vi sono situazioni in cui è necessario accedere all'iteratore (o indice), es., quando serve:
 - filtrare (rimuovere) l'elemento corrente di una collection (necessario chiamare **remove** sull'iteratore)
 - trasformare l'elemento corrente (necessario l'iteratore della collection o indice dell'array)
 - avanzare due o più iteratori/indici insieme

remove: Collection 0 Iterator?

```
import java.util.*;
public class TestRemove {
    public static void main(String[] args) {
        Collection<Integer> c = new ArrayList<>();
        for(int i=0; i<10; i++) c.add(i);
        System.out.println(c);

        for(int i: c) if(i%2 == 0) c.remove(i);
        System.out.println(c);

        for(Iterator<Integer> i=c.iterator(); i.hasNext(); )
            if(i.next() %2 ==0) i.remove();
        System.out.println(c);
    }
}
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[1, 3, 5, 7, 9]

Non si può **modificare** una collection attraverso i suoi metodi mentre si sta usando un iteratore su di essa ...

... ma si può delegare all'iteratore

Exception in thread "main"
java.util.ConcurrentModificationException



Esercitazione



Parte 1

Creare una mazzo di carte da ramino $\{A, 2...10, J, Q, K\}_i, i \in \{C, Q, F, P\}$, due carte per tipo.

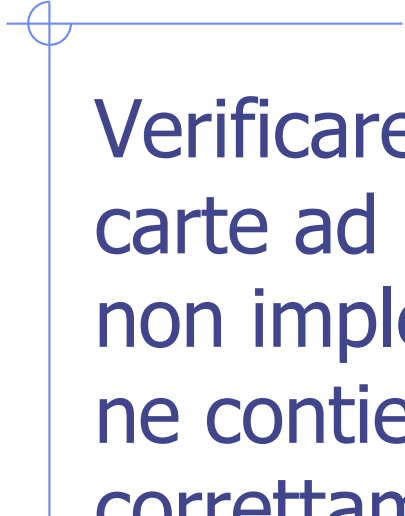
Mescolare il mazzo in modo casuale e mostrare le prime N carte (default N=10).

Controllare se tra le carte c'è una doppia e mostrare una finestra, che dirà "hai vinto" se la si è trovata, "hai perso" altrimenti.

Varianti

- Rendere N scegliibile dall'utente, e validarne la risposta.
- Dare, nella finestra finale, una rappresentazione grafica semplificata della carta vincente.
- Dopo la eventuale vittoria, scegliere a caso una carta dal mazzo. Se questa ha lo stesso valore della vincente (indipendentemente da seme), la vittoria vale doppio. Dare comunicazione all'utente dell'esito di tale operazione.

Set vs. List



Verificare che il mazzo creato aggiungendo le carte ad un Set contiene 104 elementi se Carta non implementa la equals e la hashCode, mentre ne contiene 52 se Carta implementa correttamente equals ed hashCode.

Spiegarne le ragioni.