

Collections



Collections in Java

- Una **Collection** è un oggetto che raggruppa elementi multipli in una singola entità
 - A differenza degli array, gli elementi possono essere eterogenei e 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)

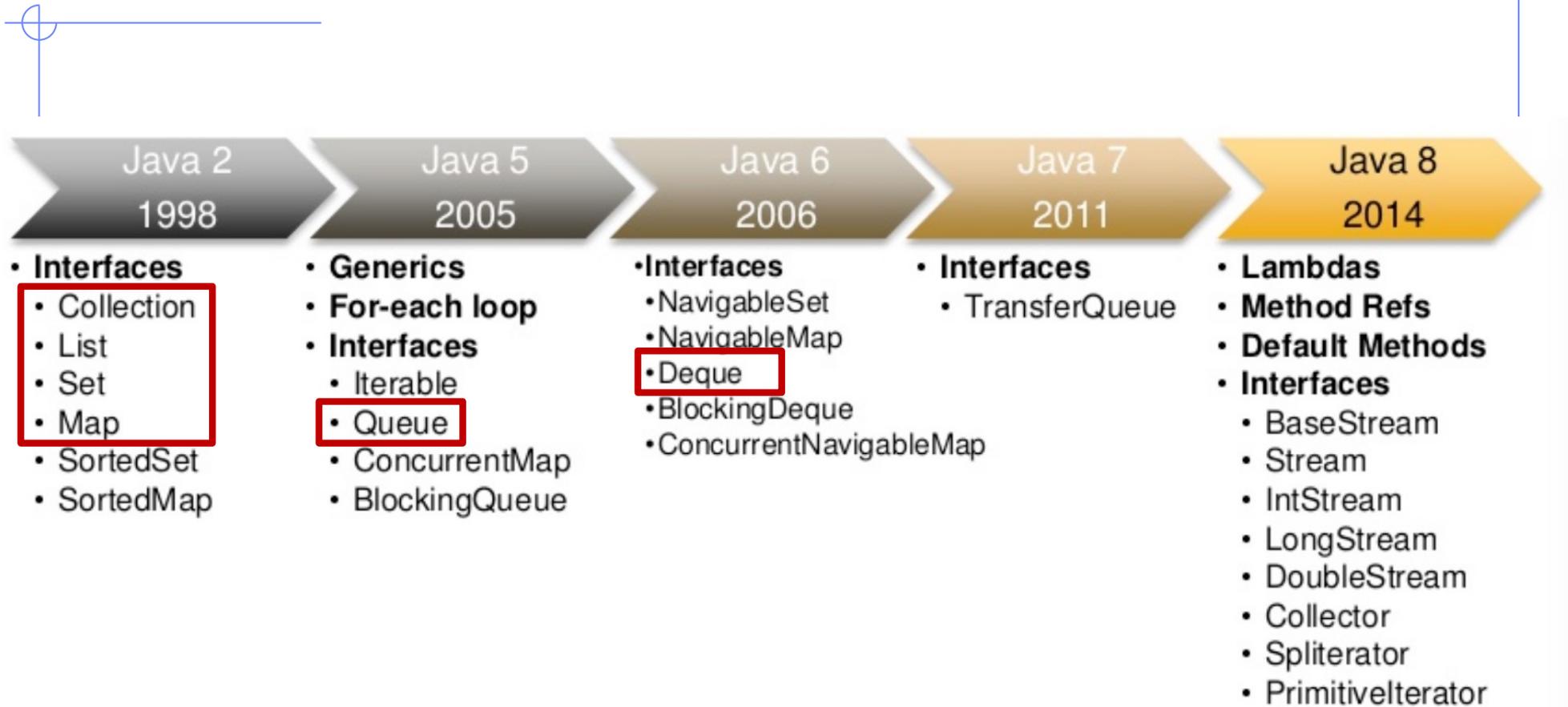
Collections Framework

- Il Java Collection Framework contiene tre tipi di elementi:
 - **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 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

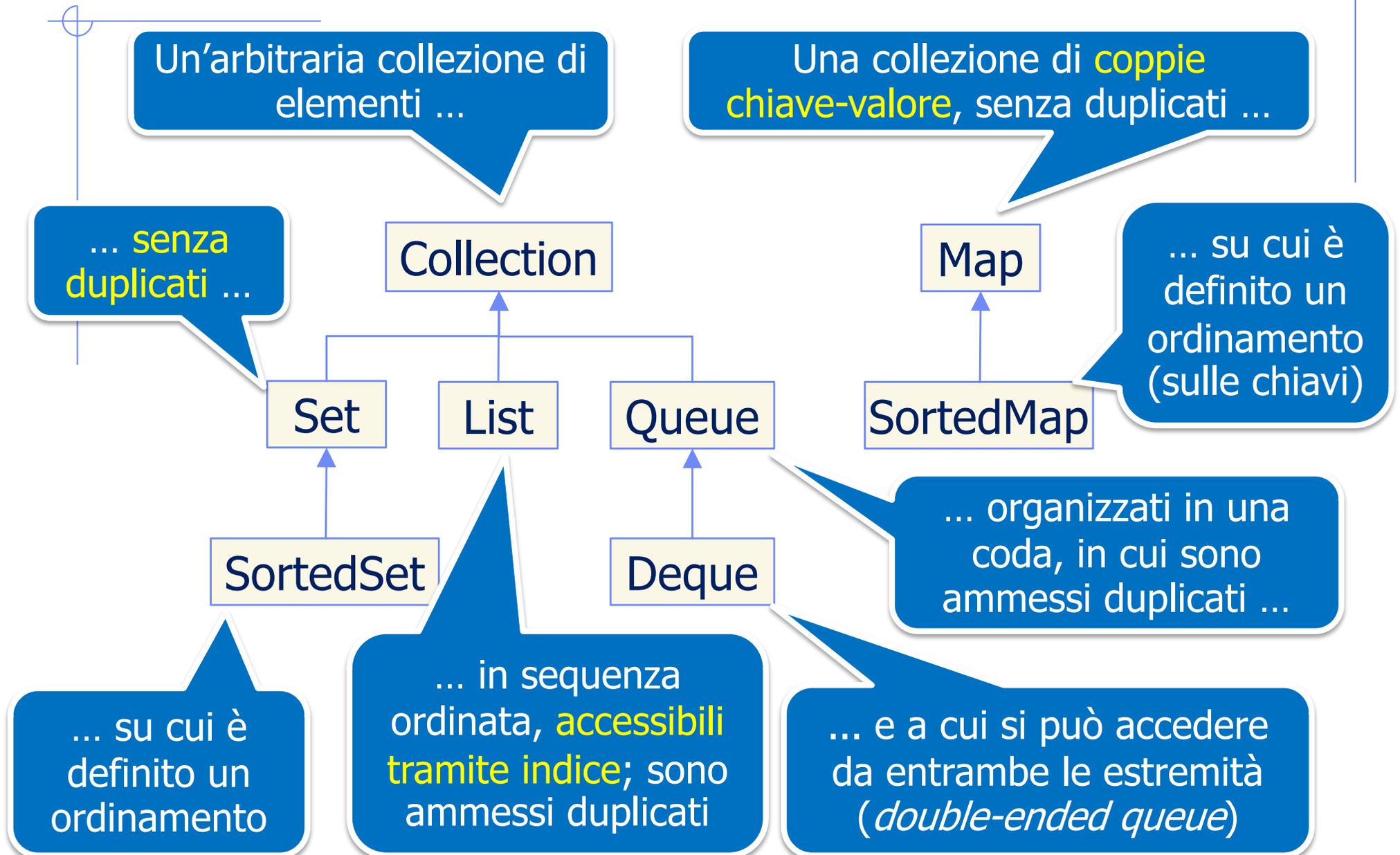
Un po' di storia...



Per usare il Java Collections Framework:

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

Interfacce (*core*)



Collection: operazioni base

```
int size();
```

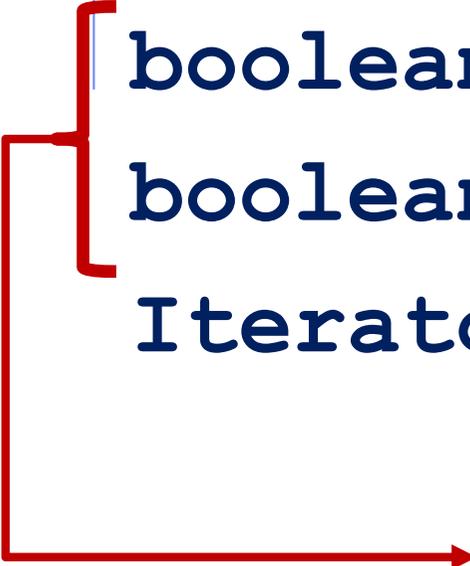
```
boolean isEmpty();
```

```
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

Collection: *bulk operations*

- Consentono di effettuare operazioni su più elementi contemporaneamente

```
boolean containsAll(Collection c);
```

```
boolean addAll(Collection c);
```

```
boolean removeAll(Collection c);
```

```
boolean retainAll(Collection c);
```

```
void clear();
```

- Il metodo **toArray()** consente di ottenere il contenuto di una collezione in un array, es.:

```
Object[] a = c.toArray();
```

- Utile per interagire con API basate su array anziché collection

List: operazioni base (addizionali)

```
Object get(int index);
```

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

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

```
Object remove(int index);
```

```
int indexOf(Object o);
```

```
int lastIndexOf(Object o);
```

Map: associazione chiave-valore

La chiave deve essere univoca!

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();
```

Implementazioni (alcune)

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

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`?

Coda

```
public class Coda extends ArrayDati {  
    public int estrai() {  
        Number x = null;  
        if (this.size()==0)  
            System.out.println("Tentativo di estrarre  
                                da una Coda vuota");  
  
        System.exit(1);  
    }  
    x=this.remove(1);  
    return x.getInt();  
}  
}
```

Problema: questo é
formalmente
un Object!

Coda

```
public class Coda extends ArrayDati {  
    public int estrai() {  
        Number x = null;  
        if (this.size()==0)  
            System.out.println("Tentativo di estrarre  
                                da una Coda vuota");  
        System.exit(1);  
    }  
    return ((Number) (this.remove(1))).getInt();  
}  
}
```

Il cast
risolve il problema

Pila

```
public class Coda extends ArrayDati {
    public int estrai() {
        Number x = null;
        if (this.size()==0)
            System.out.println("Tentativo di estrarre
                                da una Pila vuota");
            System.exit(1);
        }

    return ( (Number) (this.remove(size())) ).getInt();
}
}
```

main

```
public static void main(String[] args) {  
    ArrayDati s = new Coda();  
    //ArrayDati s = new Pila();  
    s.inserisci(1);  
    s.inserisci(2);  
    s.inserisci(3);  
    for(int k=0;k<=4;k++){  
        int v = s.estrai();  
        System.out.println(v);  
    }  
}
```

1
2
3

Tentativo di estrarre da
una Coda vuota

3
2
1

Tentativo di estrarre da
una Pila vuota

Ma che noia questo cast!

```
List listaDiNumeri=new LinkedList();  
listaDiNumeri.add(new Number(x));  
listaDiNumeri.add(new Number(x2));  
Number n1 =(Number) listaDiNumeri.get();  
Number n2 =(Number) listaDiNumeri.get();
```

```
myList.add(new Papera());
```

...

```
Number n =(Number)myList.get();
```



**Late Error
Detection!**

L'elemento che trovo è una Papera,
non posso farne un cast su Number!

Possiamo fare di meglio?

Tipizzazione parametrica

```
List<Number> listaDiNumeri=new LinkedList();  
listaDiNumeri.add(new Number(x));  
listaDiNumeri.add(new Number(x2));  
Number n =listaDiNumeri.get();  
Number n =listaDiNumeri.get();
```

```
listaDiNumeri.add(new Papera());
```

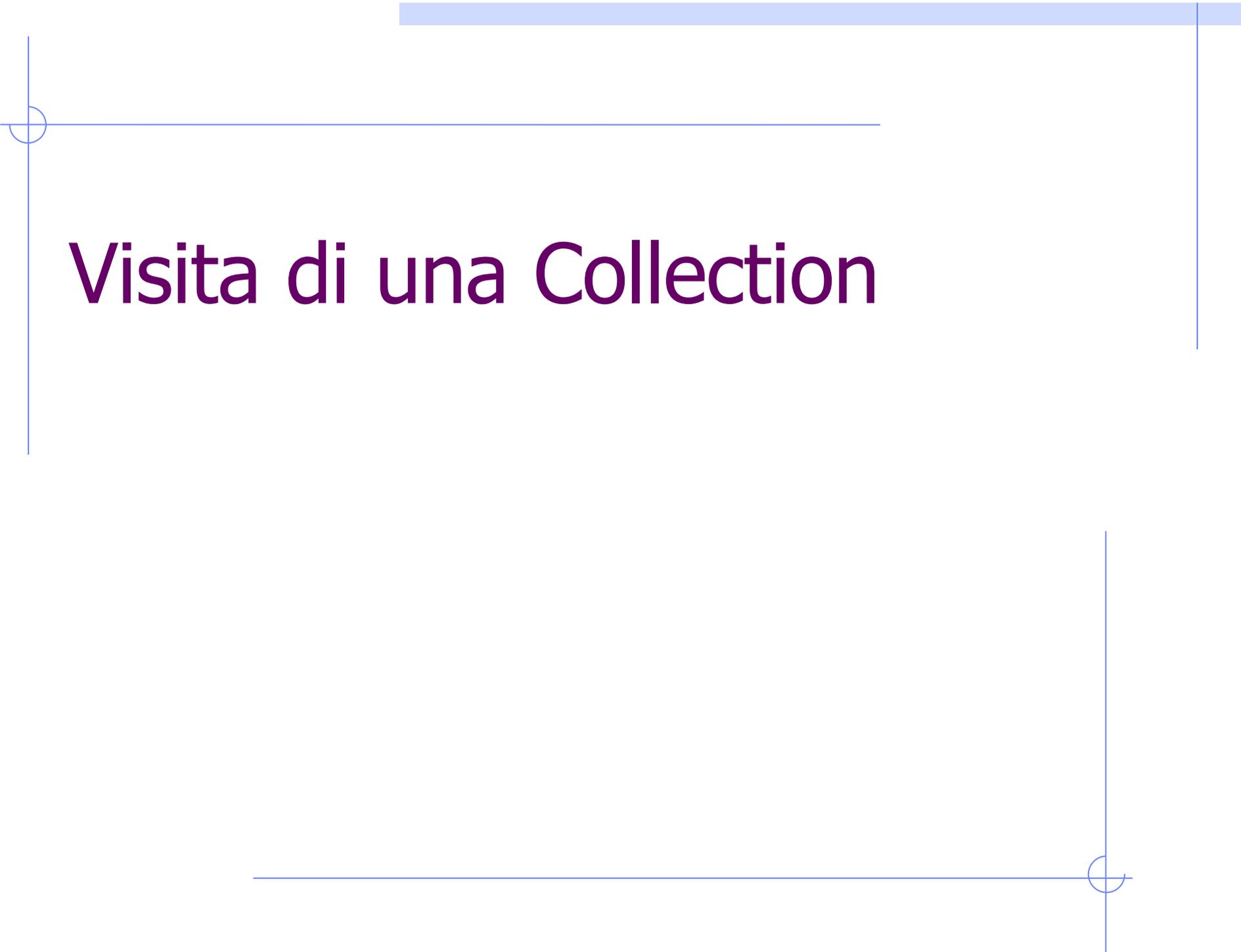
...

Senza cast!

Non posso aggiungere una Papera
a una lista di Number!

Early Error
Detection!

```
Number n = listaDiNumeri.get();
```



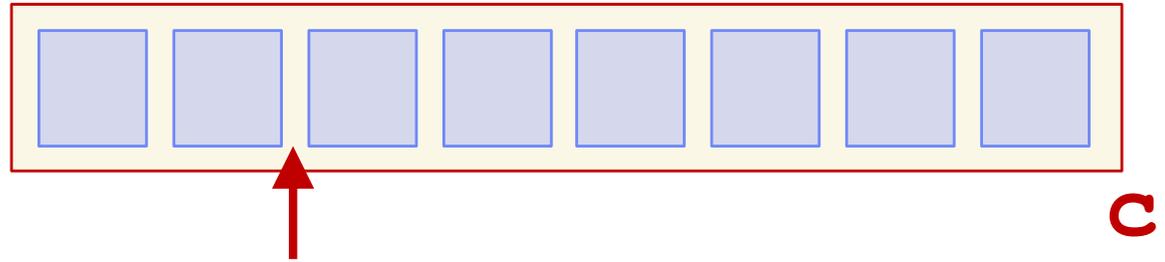
Visita di una Collection

Ciclo for generalizzato

```
List<Number> listaDiNumeri=new LinkedList();  
...  
for (Number n : listaDiNumeri) {  
    System.out.println(n);  
}
```

**Attenzione: non posso modificare
La struttura mentre la visito:
no remove()**

Iteratori



A ogni oggetto di tipo `Collection` è associato un oggetto di tipo `Iterator`

```
Iterator<Number> i = c.iterator();
```

- Consente di scandire gli elementi della collezione uno a uno

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

true ci sono altri elementi da scandire

ritorna l'elemento successivo

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

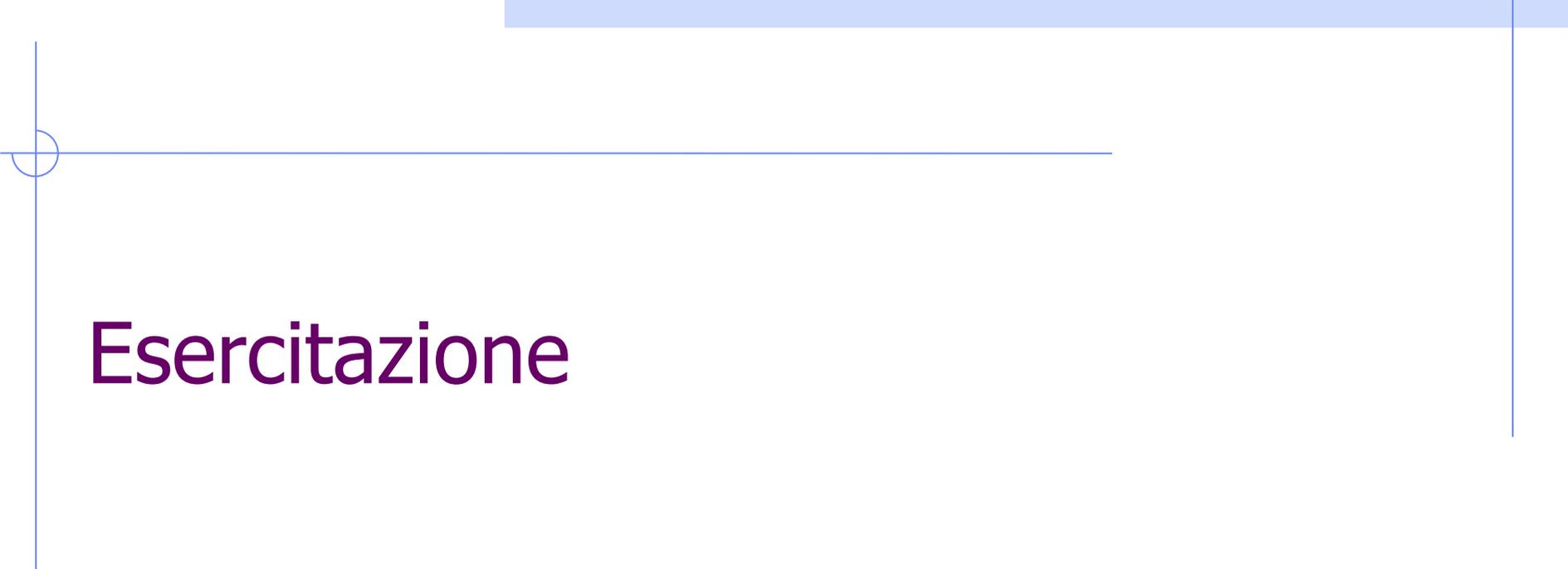
Iteratori: esempio d'uso

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

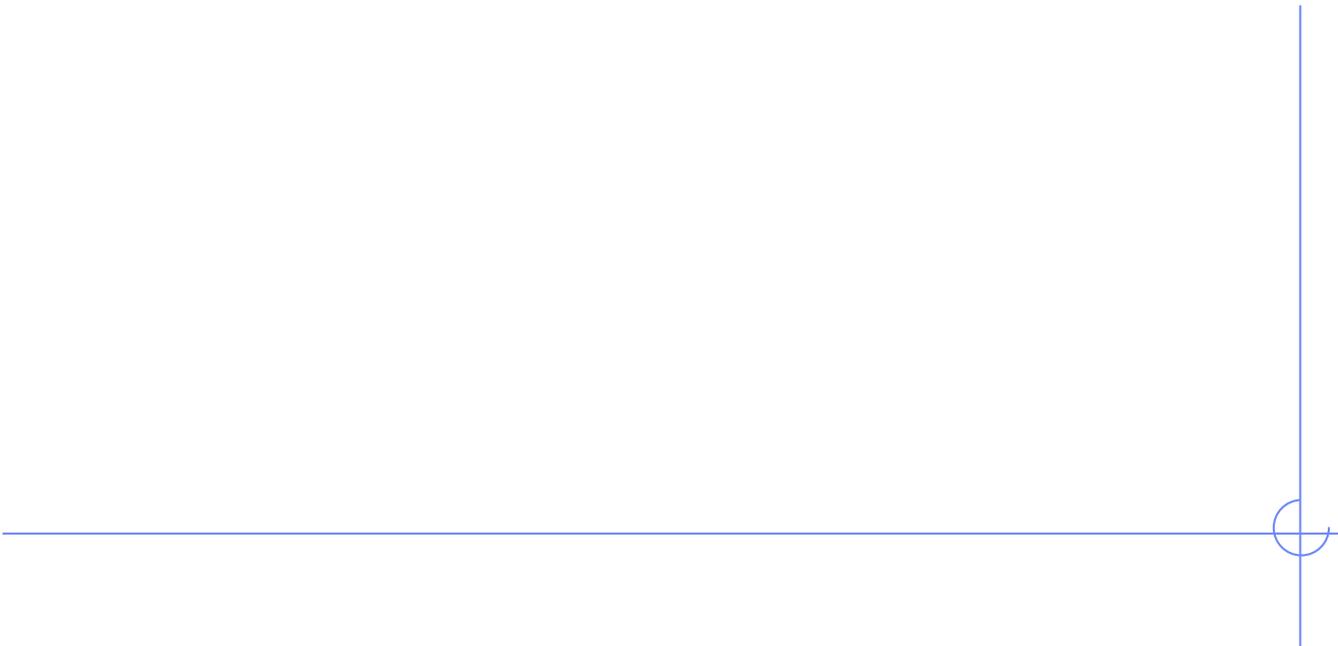
Rimuove dalla collezione gli oggetti che non soddisfano una condizione data:

boolean cond(Number n) (definita altrove).

E' un po' più potente del for generalizzato.



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.