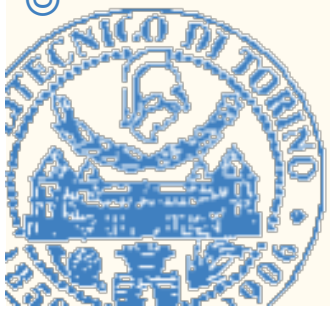
Collezioni di dati in Java

Programmazione
ad Oggetti

A.A. 2004-05



Introduzione

- Spesso un programma ha la necessità di memorizzare insiemi di dati tra loro variamente strutturati
- Il package java.util mette a disposizione
  - **modelli concettuali** per organizzare i dati (sotto forma di interfacce)
  - un certo numero di **implementazioni concrete** di tali modelli, ciascuno dotato di pregi e difetti

java.util.Collection

- Interfaccia radice della gerarchia di ereditarietà
  - Modella un gruppo di oggetti, dotato o meno di duplicati, dotato o meno di un ordine interno
  - Di solito, non viene implementata direttamente (si implementano le sue sotto-interfacce)
  - Non può contenere direttamente tipi elementari (interi, reali, caratteri, booleani)
- Due gruppi di metodi
  - Accesso ai singoli elementi
  - Modifica del contenuto

Accedere agli elementi

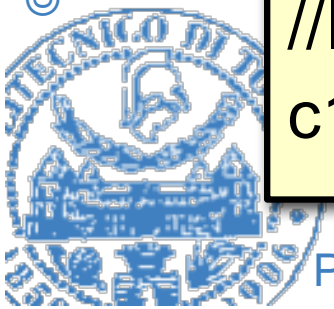
- Iterator iterator()
  - Restituisce un oggetto che consente di esaminare, ad uno ad uno, i singoli elementi contenuti
- Object[] toArray()
  - Restituisce un array contenente i riferimenti a tutti i diversi elementi contenuti
- int size()
  - Restituisce il numero di oggetti contenuti nella collezione
- boolean contains(Object o)
  - Indica se nella è presente un elemento “e” che soddisfa la relazione e.equals(o) oppure, nel caso in cui “o” valga null, se la lista contiene un altro elemento nullo

Modificare una collezione

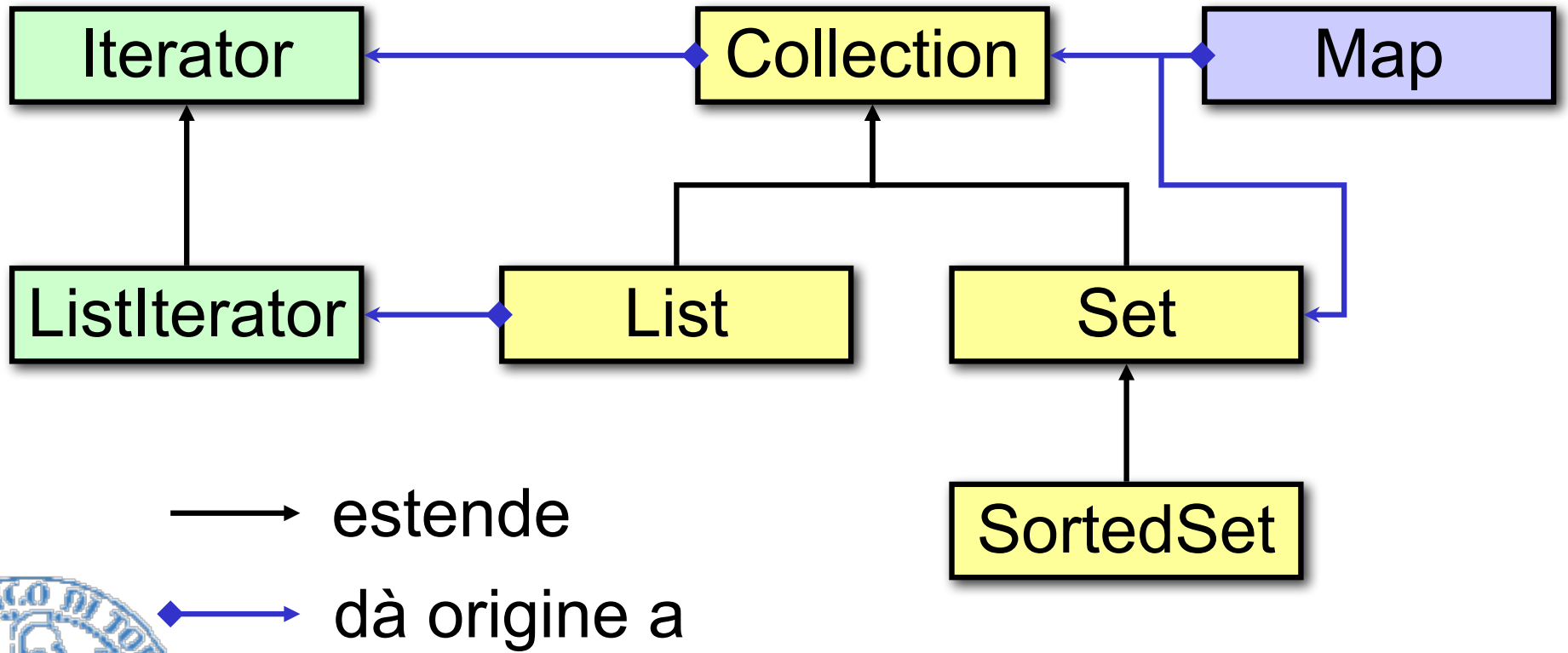
- Insieme di operazioni elementari per aggiungere ed eliminare oggetti
  - Le singole implementazioni possono implementarne il funzionamento o lanciare UnsupportedOperationException
- Inserimento
  - boolean add(Object o)
  - boolean addAll(Collection c)
- Cancellazione
  - void clear()
  - boolean remove(Object o)
  - boolean removeAll(Collection c)
  - boolean retainAll(Collection c)

Operazioni sulle collezioni

```
Collection c1,c2,c3,c4;
//...

//Aggiungo gli elementi di c2 a c1
c1.addAll(c2);

//Elimino gli elementi presenti in c3
c1.removeAll(c3);

//Interseco c1 con c4
c1.retainAll(c4);
```

Gerarchia di ereditarietà



java.util.List (1)

- Collezione ordinata
  - Può contenere duplicati e/o elementi nulli
  - Permette l'accesso agli elementi contenuti anche in base alla loro posizione
  - Estende il concetto di iteratore (ListIterator) per garantire un accesso efficiente alla sequenza contenuta
- Accesso agli elementi
  - ListIterator listIterator()
  - ListIterator listIterator(int index)
  - Object get(int index)
  - int indexOf(Object o)
  - List subList(int fromIndex, int toIndex)

java.util.List (2)

- Modificare gli elementi
  - void add(int index, Object o)
  - void addAll(int index, Collection c)
  - Object remove(int index)
  - Object set(int index, Object o)

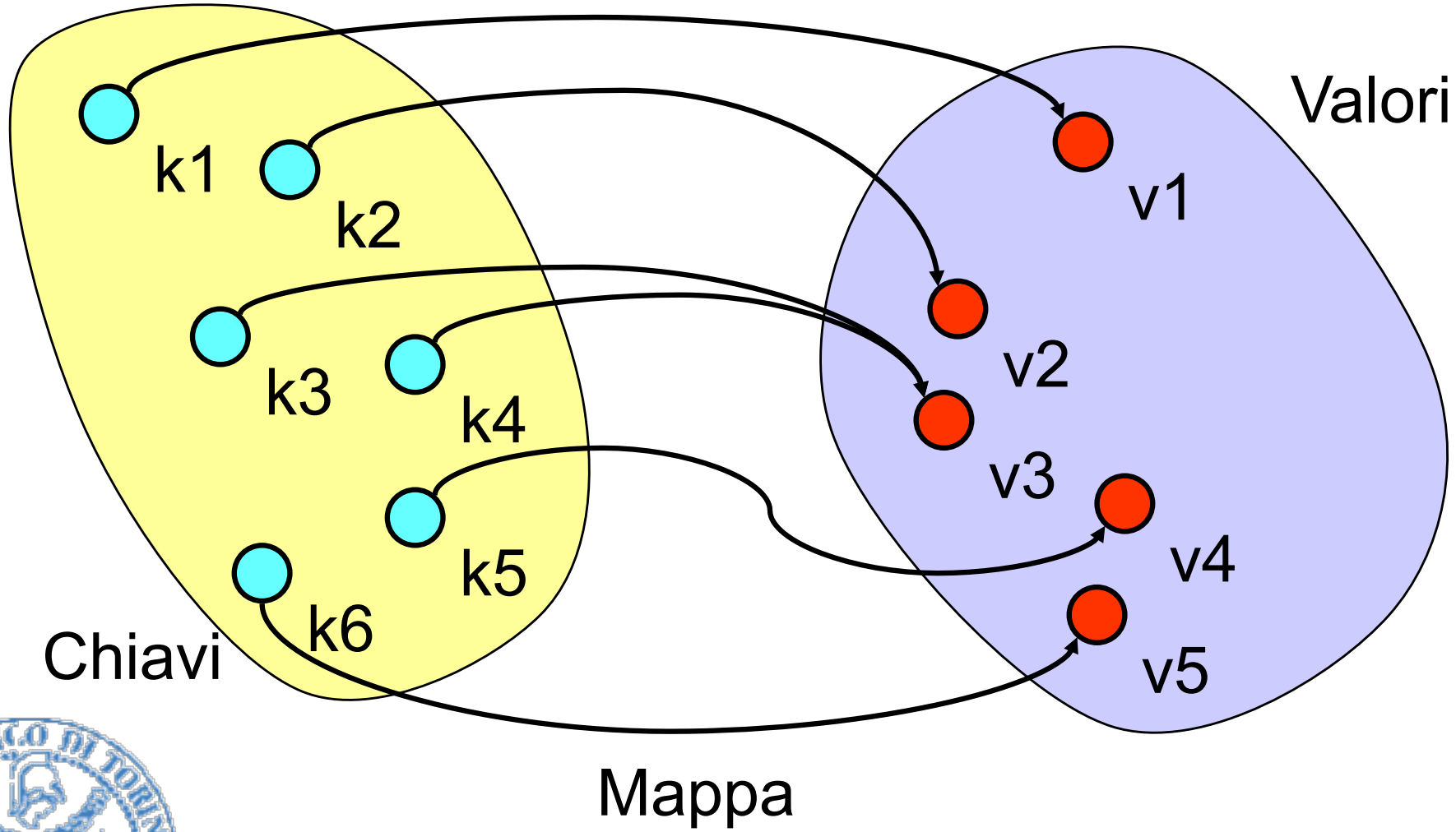
java.util.Set

- Collezione priva di duplicati
  - Modella il concetto matematico di insieme
  - Aggiunge solo vincoli su quali oggetti siano inseribili e sulla semantica delle operazioni
  - Non ha operazioni differenti rispetto a Collection
- SortedSet: estende il concetto di insieme
  - Il suo iteratore permette di visitare i singoli elementi in ordine lessicografico (o nell'ordine definito da un oggetto di tipo java.util.Comparator)

java.util.Map (1)

- Modella il concetto di relazione iniettiva tra due insiemi di oggetti, detti rispettivamente chiavi e valori
  - La mappa associa, ad ogni chiave, uno ed un solo valore
  - La chiave dovrebbe essere un oggetto immutabile
  - In base alle implementazioni, può essere lecito o meno che chiavi e valori valgano *null*

java.util.Map (2)



Operazioni su una mappa

- Accesso agli elementi
  - Object get(Object key)
  - boolean containsKey(Object key)
  - boolean containsValue(Object value)
- Accesso agli insiemi
  - Set entrySet()
  - Set keySet()
  - Collection values()
  - boolean isEmpty()
  - int size()

Elementi di una mappa

```
Map map;
//...
Iterator it = map.keySet().iterator();
while (it.hasNext()) {
        Object key = it.next();
        //...
}
```

```
Map map;
//...
Iterator it = map.values().iterator();
while (it.hasNext()) {
        Object value = it.next();
        //...
}
```

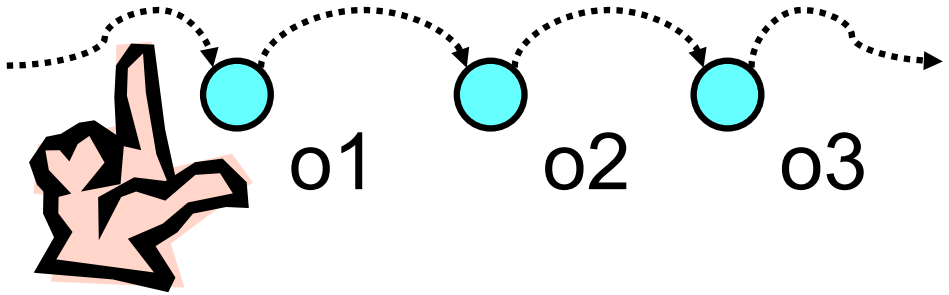

Coppie di una mappa

```
Map map;
//...
Iterator it = map.entrySet().iterator();
while (it.hasNext()) {
        Map.Entry entry= (Map.Entry) it.next();
        Object key = entry.getKey();
        Object value = entry.getValue();
        //...
}
```


java.util.Iterator (1)

- Modella il concetto di visita (eventualmente distruttiva) di una collezione
  - Permette di esaminare uno alla volta i singoli elementi appartenenti, supportando il concetto di eliminazione del singolo elemento
  - Sostituisce la classe java.util.Enumeration che non definisce una semantica per l'eliminazione
- Metodi
  - boolean hasNext()
  - Object next()
  - void remove()

java.util.Iterator (2)



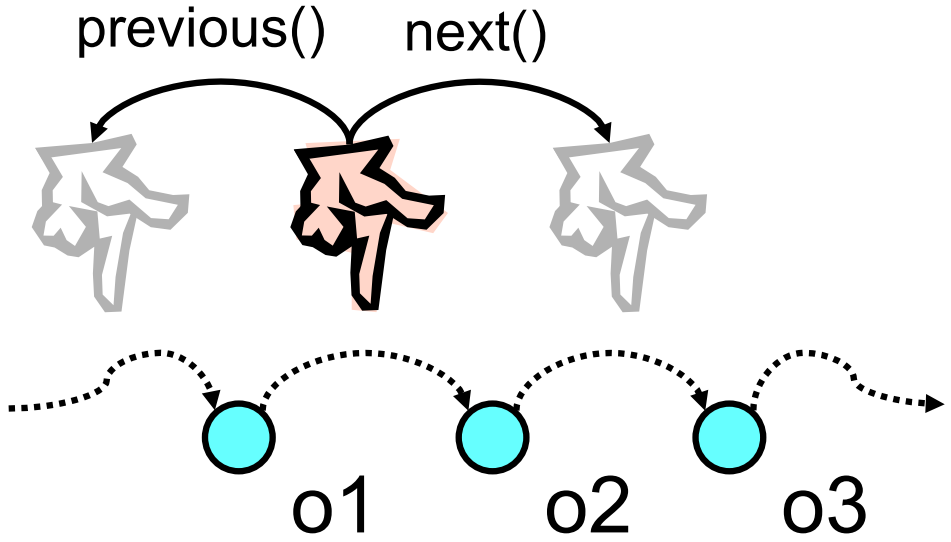
- Può essere immaginato come un indice posto tra due elementi consecutivi
  - All'atto della costruzione, si trova prima del primo elemento
  - La chiamata a next() provoca l'avanzamento alla posizione successiva (se esiste) o la generazione di un'eccezione
  - remove() elimina l'ultimo elemento restituito (quello che precede il puntatore, se esiste)

Elementi di una collezione

```
Collection c;
//...
Iterator it = c.iterator();
while (it.hasNext() ) {
        Object element = it.next();
        //...
}
```


java.util.ListIterator (1)

- Estende il concetto di Iterator, permettendo la visita in entrambe le direzioni
  - Oltre alla visita ed all'eliminazione, può permettere le operazioni di inserimento e sostituzione
  - Nel caso in cui si visiti una lista alternando le direzioni (procedendo avanti ed indietro) all'atto del cambio viene ritornato lo stesso elemento due volte consecutive



java.util.ListIterator (2)

- Metodi supportati
  - void add(Object o)
  - boolean hasNext()
  - boolean hasPrevious()
  - Object next()
  - int nextIndex()
  - Object previous()
  - int previousIndex()
  - void remove()
  - void set(Object o)

Implementazioni

- All'interno del package java.util sono presenti un certo numero di classi che implementano le interfacce precedenti
  - Una data interfaccia può essere implementata da più classi, con algoritmi differenti
  - A parità di comportamento funzionale, le prestazioni possono essere sensibilmente differenti
- Per generalità, le variabili di tipo collezione, si dichiarano come appartenenti alla relativa interfaccia e si fa riferimento alla classe concreta solo all'atto della costruzione:
  - List l = new LinkedList();
- Per lo più, le implementazioni non sono sincronizzate

Principali classi (1)

- LinkedList
  - Rappresenta una lista doppiamente collegata
  - Oltre alle funzionalità base definite da List, offre metodi per l'inserimento e la cancellazione di elementi in testa ed in coda (implementazione di pile e code)
  - I tempi di accesso al singolo elemento interno crescono al crescere della dimensione
- ArrayList
  - Lista basata su un array dinamicamente riallocato
  - Ha tempi di accesso ad un elemento arbitrario costanti ma tempi di inserimento proporzionali al numero di elementi da spostare

Principali classi (2)

- HashSet
  - Implementa il concetto di insieme appoggiandosi su una tabella “hash”
  - Ha tempi di inserimento, cancellazione, accesso mediamente costanti
  - Presuppone che gli oggetti inseriti dispongano di un’implementazione adeguata dei metodi int hashCode() e boolean equals(Object o) definiti nella classe Object

Principali classi (3)

- HashMap
  - Implementa il concetto di mappa appoggiandosi su una tabella “hash”
  - Ha tempi costanti per le operazioni base (get e put), ma l’iterazione sugli elementi richiede un tempo proporzionale alla dimensione
  - Due fattori influenzano le prestazioni e di cui si dovrebbe tener conto all’atto della creazione:
    - Capacità iniziale:
      numero di elementi previsti nella tabella di hash
    - Fattore di carico:
      indica la percentuale massima di elementi occupati prima che la capacità sia incrementata automaticamente (operazione di rehash)

java.util.Collections (1)

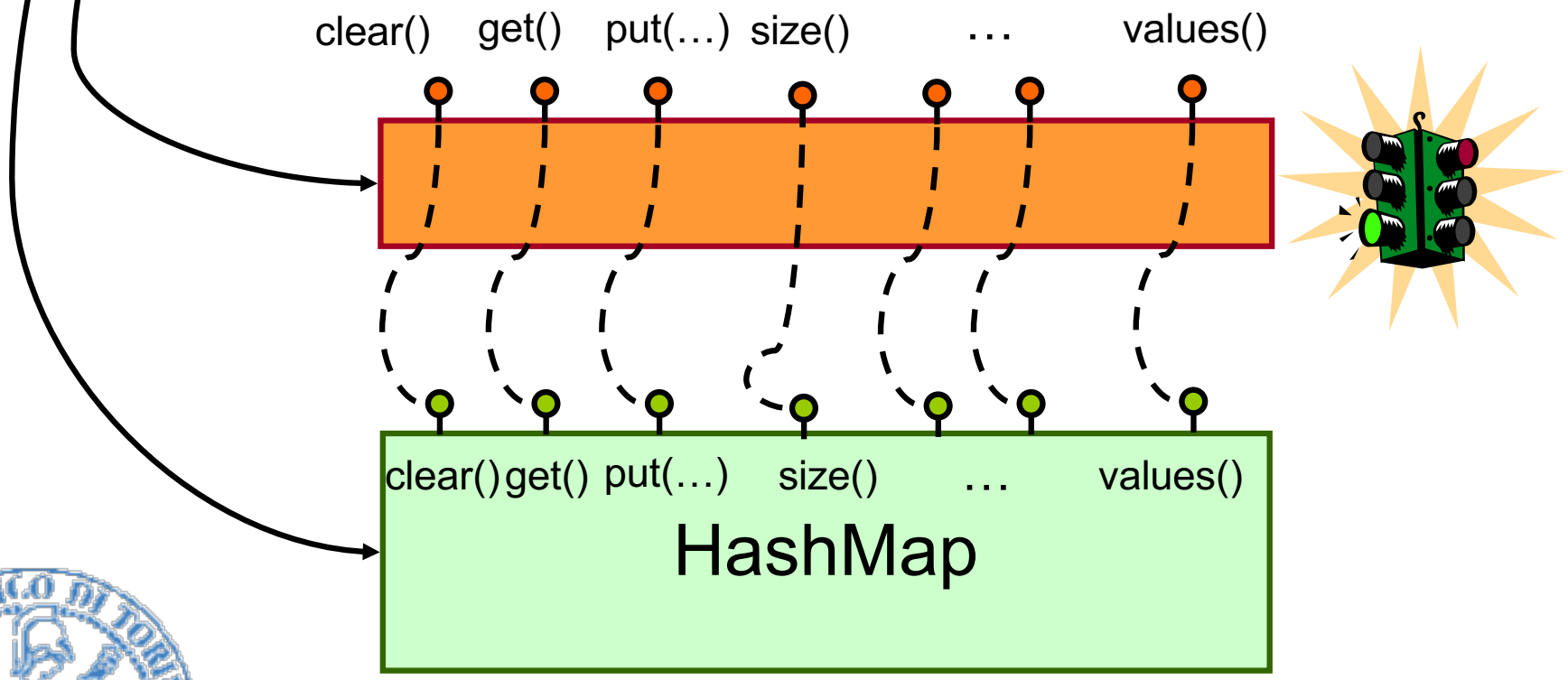
- Fornisce un insieme di metodi statici di utilità per la gestione delle collezioni
- Ricerca binaria, ordinamento, copia di una intera lista, ritrovamento del elemento minimo o massimo, sostituzione di un elemento, …
  - static int binarySearch(…)
  - static void sort(…)
  - static void copy(List dest, List src)
  - static void fill(List list, Object obj)
  - static Object max(…)
  - static Object min(…)
  - static boolean replaceAll(List list, Object old, Object new)
  - …

java.util.Collections (2)

- Tutte le classi viste precedentemente non sono sincronizzate:
  - L'accesso da parte di più thread deve essere gestito dal programmatore
  - In alternativa si possono utilizzare i metodi forniti da Collections per la creazione di oggetti che incapsulano la sincronizzazione
    - static Collection synchronizedCollection(Collection c)
    - static List synchronizedList(List list)
    - static Map synchronizedMap(Map m)
    - static Set synchronizedSet(Set s)
    - static SortedMap synchronizedSortedMap(SortedMap m)
    - static SortedSet synchronizedSortedSet(SortedSet s)
  - Modifiche concorrenti restituiscono l'eccezione ConcurrentModificationException

Accesso sincronizzato

```
Map m1=new HashMap();
Map m2=Collections.synchronizedMap(m1);
```



Accesso in sola lettura (1)

- Collections mette a disposizione un insieme di metodi che rendono l'oggetto non modificabile:
  - static Collection unmodifiableCollection(Collection c)
  - static List unmodifiableList(List list)
  - static Map unmodifiableMap(Map m)
  - static Set unmodifiableSet(Set s)
  - static SortedMap unmodifiableSortedMap(SortedMap m)
  - static SortedSet unmodifiableSortedSet(SortedSet s)
- Un eventuale tentativo di modifica genera l'eccezione UnsuppotedOperationException

Accesso in sola lettura (2)