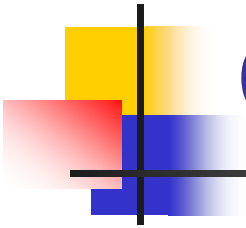
Linguaggi

Corso di Laurea in Ingegneria delle Telecomunicazioni
A.A. 2010-2011

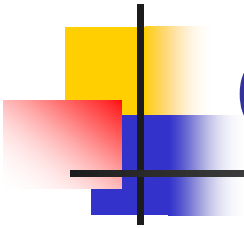
Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it

Collezioni in Java



Collections Framework

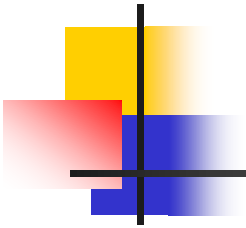
- Cos'è una collezione?
- Un oggetto che raggruppa un gruppo di elementi in un singolo oggetto.
- Uso: memorizzare, manipolare e trasmettere dati da un metodo ad un altro.
- Tipicamente rappresentano gruppi di elementi naturalmente collegati:
  - Una collezione di lettere
  - Una collezione di numeri di telefono



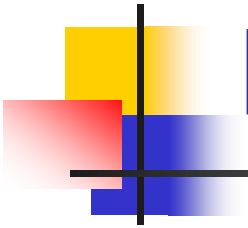
Cosa è l'ambiente "Collections"?

- E' una architettura unificata per manipolare collezioni.
- Un framework di collection contiene tre elementi:
  - Interfacce
  - Implementazioni (delle interfacce)
  - Algoritmi
- Esempi
  - C++ Standard Template Library (STL)
  - Smalltalk's collection classes.
  - Java's collection framework

Interfacce in un Collections Framework

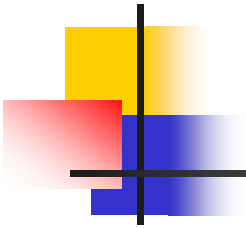


- Abstract data types rappresentanti le collezioni.
- Permentono di manipolare le collezioni indipententemente dai dettagli delle loro implementazioni.
- In un linguaggio object-oriented come Java, queste interfacce formano una gerarchia



Algoritmi in un Collections Framework

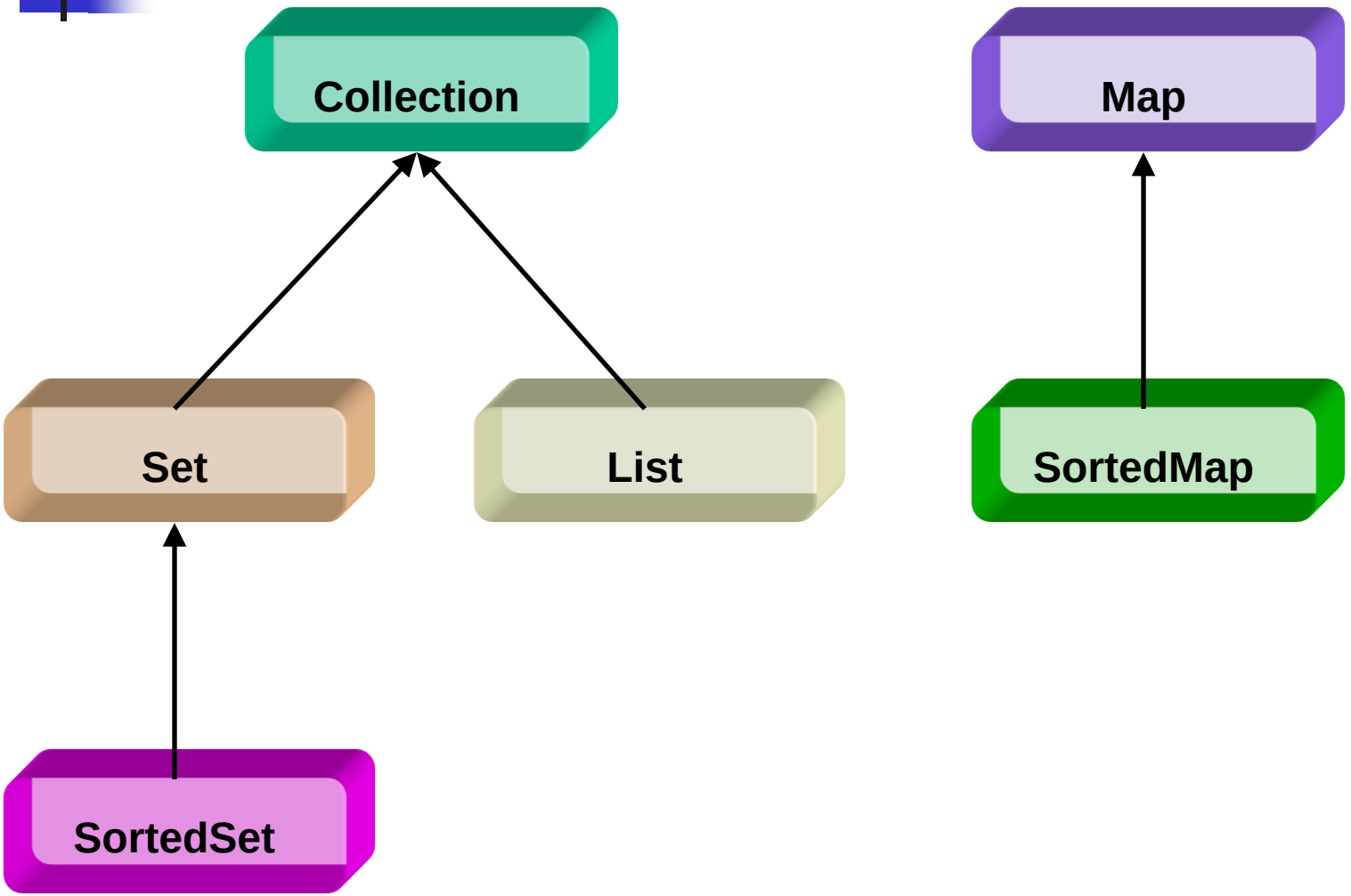
- Metodi che eseguono utili computazioni come ricerca, ordinamento sugli oggetti che implementano le interfaccie
- Questi algoritmi sono polimorfi in quanto lo stesso metodo può essere utilizzato su molte differenti implementazioni della stessa interfaccia.

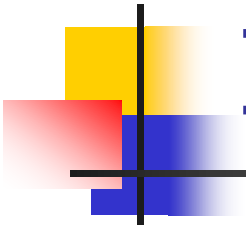


Beneficio di un Collections Framework

- Riduce lo sforzo di programmazione
- Accresce la velocità e la qualità di programmazione
- Interoperabilità fra API scorrelate
- Riduce lo sforzo di imparare ed utilizzare nuove API
- Riduce lo sforzo per progettare nuove API
- Riuso del software

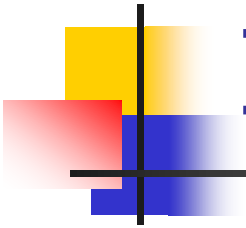
Interfacce di Collections





Interfaccia Collection

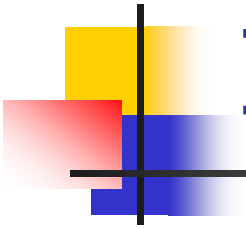
- Radice della gerarchia delle collezioni
- Rappresenta un gruppo di oggetti detti elementi della collezione.
- Alcune implementazioni di `Collection` consentono la duplicazione degli elementi mentre altre no. Alcune sono ordinate altre no.
- `Collection` è utilizzato per passare collezioni e manipolarle con la massima generalità.



Interfaccia Collection

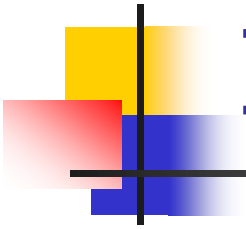
- Major methods:

```
int size();
boolean isEmpty();
boolean contains(Object);
Iterator iterator();
Object[] toArray();
Object[] toArray(Object []);
boolean add(Object);
boolean remove(Object);
void clear();
```



Interfaccia Collection

- boolean **add**(Object o)
  Ensures that this collection contains the specified element
- boolean **addAll**(Collection c)
  Adds all of the elements in the specified collection to this collection
- void **clear**()
  Removes all of the elements from this collection
- boolean **contains**(Object o)
  Returns true if this collection contains the specified element.
- boolean **containsAll**(Collection c)
  Returns true if this collection contains all of the elements in the specified collection.

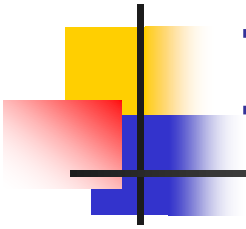


Interfaccia Collection

- boolean **equals**(Object o)
  Compares the specified object with this collection for equality.
- int **hashCode**()
  Returns the hash code value for this collection.
- boolean **isEmpty**()
  Returns true if this collection contains no elements.
- Iterator **iterator**()
  Returns an iterator over the elements in this collection.
- boolean **remove**(Object o)
  Removes a single instance of the specified element from this collection, if it is present (optional operation).

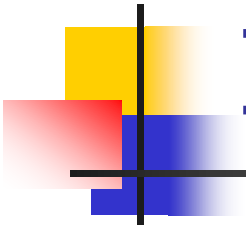
Interfaccia Collection

- boolean **removeAll**(Collection c)
  Removes all this collection's elements that are also contained in the specified collection
- boolean **retainAll**(Collection c)
  Retains only the elements in this collection that are contained in the specified collection
- int **size**()
  Returns the number of elements in this collection.
- Object[] **toArray**()
  Returns an array containing all of the elements in this collection.
- Object[]**toArray**(Object[] a)
  Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.



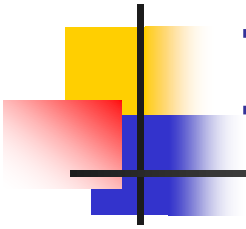
Interfaccia Collection

- **All Known Subinterfaces:**
  - BeanContext, BeanContextServices, List, Set, SortedSet
- **All Known Implementing Classes:**
  - AbstractCollection, AbstractList, AbstractSet, ArrayList, BeanContextServicesSupport, BeanContextSupport, HashSet, LinkedHashSet, LinkedList, TreeSet, Vector



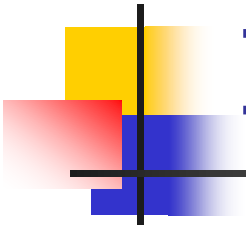
Interfaccia Set

- Interface `Set` extends `Collection`
- Modella l'astrazione matematica di insieme
  - Collezione non ordinata di object
- No elementi duplicati
- Gli stessi metodi di `Collection`
  - Semantica differente



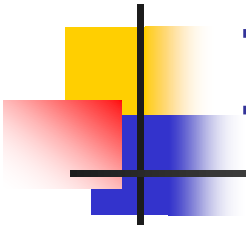
Interfaccia Set

- **All Superinterfaces:**
  - Collection
- **All Known Subinterfaces:**
  - SortedSet
- **All Known Implementing Classes:**
  - AbstractSet, HashSet, LinkedHashSet, TreeSet



Interfaccia SortedSet

- Interface `SortedSet` extends `Set`
- Un insieme ordinato
- Tutti gli elementi inseriti in un sorted set devono implementare l'interfaccia Comparable

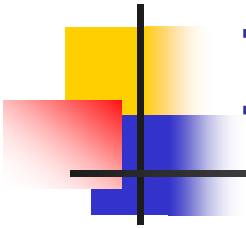


Interfaccia SortedSet

- **All Superinterfaces:**
  - Collection, Set
- **All Known Implementing Classes:**
  - TreeSet

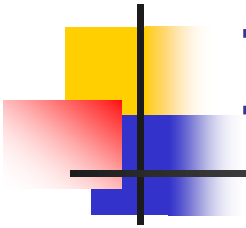
Interfaccia SortedSet

- Comparator **comparator**()
  Returns the comparator associated with this sorted set, or null if it uses its elements' natural ordering.
- Object **first**()
  Returns the first (lowest) element currently in this sorted set.
- SortedSet **headSet**(Object toElement)
  Returns a view of the portion of this sorted set whose elements are strictly less than toElement.
- Object **last**()
  Returns the last (highest) element currently in this sorted set.
- SortedSet **subSet**(Object fromElement, Object toElement)
  Returns a view of the portion of this sorted set whose elements range from fromElement, inclusive, to toElement, exclusive.
- SortedSet**tailSet**(Object fromElement)
  Returns a view of the portion of this sorted set whose elements are greater than or equal to fromElement



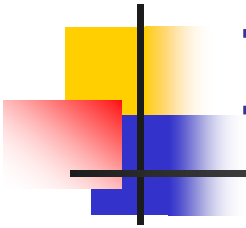
Interfaccia List

- Una collezione ordinata (chiamata anche sequenza).
- Può contenere elementi duplicati
- Consente il controllo della posizione nella lista in cui un elemento è inserito
- L'accesso agli elementi è eseguito rispetto al loro indice intero (posizione).



Interfaccia List

- **All Superinterfaces:**
  - Collection
- **All Known Implementing Classes:**
  - AbstractList, ArrayList, LinkedList, Vector



Interfaccia List

void **add**(int index, Object element)

Inserts the specified element at the specified position in this list

boolean **add**(Object o)

Appends the specified element to the end of this list

boolean **addAll**(Collection c)

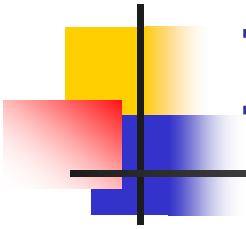
Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional method).

boolean **addAll**(int index, Collection c)

Inserts all of the elements in the specified collection into this list at the specified position

void **clear**()

Removes all of the elements from this list (optional method).



Interfaccia List

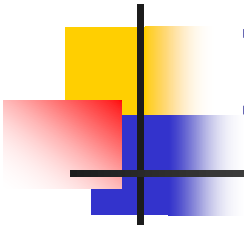
boolean **contains**(Object o)
Returns true if this list contains the specified element.

boolean **containsAll**(Collection c)
Returns true if this list contains all of the elements of the specified collection.

boolean **equals**(Object o)
Compares the specified object with this list for equality.

Object **get**(int index)
Returns the element at the specified position in this list.

int **hashCode**()
Returns the hash code value for this list.



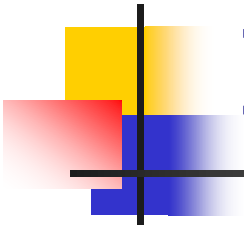
Interfaccia List

int **indexOf**(Object o)
Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.

boolean **isEmpty**()
Returns true if this list contains no elements.

Iterator **iterator**()
Returns an iterator over the elements in this list in proper sequence.

Int **lastIndexOf**(Object o)
Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element.



Interfaccia List

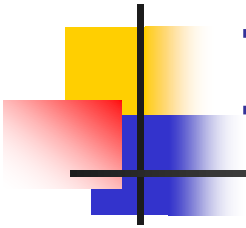
Object **remove**(int index)
Removes the element at the specified position in this list (optional operation).

boolean **remove**(Object o)
Removes the first occurrence in this list of the specified element (optional operation).

boolean **removeAll**(Collection c)
Removes from this list all the elements that are contained in the specified collection (optional operation).

boolean **retainAll**(Collection c)
Retains only the elements in this list that are contained in the specified collection (optional operation).

Object **set**(int index, Object element)
Replaces the element at the specified position in this list with the specified element (optional operation).



Interfaccia List

List **subList**(int fromIndex, int toIndex)
Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

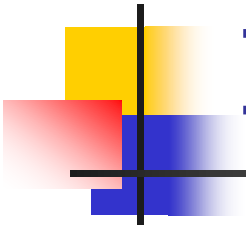
Object[]**toArray**()
Returns an array containing all of the elements in this list in proper sequence.

Object[]**toArray**(Object[] a)
Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array.

int **size**()
Returns the number of elements in this list.

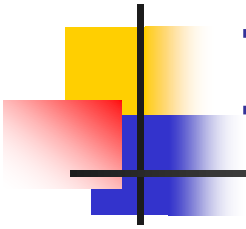
ListIterator **listIterator**()
Returns a list iterator of the elements in this list (in proper sequence).

ListIterator **listIterator**(int index)
Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.



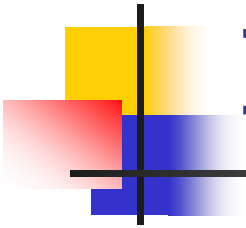
Interfaccia Map

- Interface Map (non estende Collection)
- Un object corrisponde (maps) ad almeno una chiave
- Non contiene chiavi duplicate; ogni chiave corrisponde al più un valore
- Sostituisce la classe astratta java.util.Dictionary
- L'ordine può essere fornito da classi che implementano l'interfaccia



Interfaccia Map

- **All Known Subinterfaces:**
  - SortedMap
- **All Known Implementing Classes:**
  - AbstractMap, Attributes, HashMap, Hashtable, IdentityHashMap, RenderingHints, TreeMap, WeakHashMap

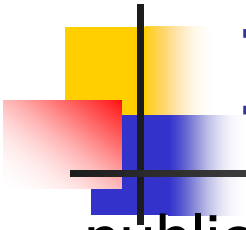


Interfaccia Map

- public int **size**()
  - Returns the number of key-value mappings in this map.
- public boolean **isEmpty**()
  - Returns true if this map contains no key-value mappings.
- public boolean **containsKey**(Object key)
  - Returns true if this map contains a mapping for the specified key.
- public boolean **containsValue**(Object value)
  - Returns true if this map maps one or more keys to the specified value.
- public Object **get**(Object key)
  - Returns the value to which this map maps the specified key. Returns null if the map contains no mapping for this key.

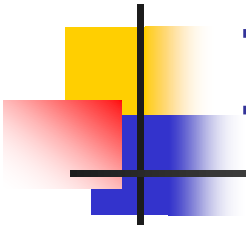
Interfaccia Map

- public Object **put**(Object key, Object value)
  - Associates the specified value with the specified key in this map (optional operation).
- public Object **remove**(Object key)
  - Removes the mapping for this key from this map if it is present (optional operation).
- public void **putAll**(Map t)
  - Copies all of the mappings from the specified map to this map (optional operation).
- public void **clear**()
  - Removes all mappings from this map (optional operation).
- public Set **keySet**()
  - Returns a set view of the keys contained in this map.



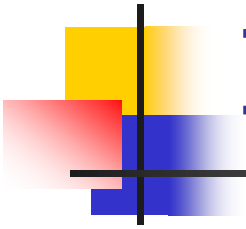
Interfaccia Map

- public Collection values()
  - Returns a collection view of the values contained in this map.
- public Set entrySet()
  - Returns a set view of the mappings contained in this map.
- public boolean equals(Object o)
  - Compares the specified object with this map for equality. Returns true if the given object is also a map and the two Maps represent the same mappings.
- public int hashCode()
  - Returns the hash code value for this map. The hash code of a map is defined to be the sum of the hashCodes of each entry in the map's entrySet view.



Interfaccia SortedMap

- public interface **SortedMap** extends Map
- un map che garantisce l'ordine crescente.
- Tutti gli elementi inseriti in un sorted map devono implementare l'interfaccia Comparable



Interfaccia SortedMap

- **All Superinterfaces:**
  - Map
- **All Known Implementing Classes:**
  - TreeMap

Implementazioni nel Framework Collections

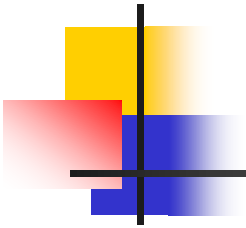
- Le implementazioni concrete dell'interfaccia collection sono strutture dati riutilizzabili

Implementazioni fondamentali

- per le liste: *ArrayList,LinkedList, Vector*
- per le tabelle hash: *HashMap, HashSet, Hashtable*
- per gli alberi: *TreeSet, TreeMap (implementano* rispettivamente SortedSet e SortedMap)

Due parole sulle classi di implementazione:

- le *linked list* sono liste realizzate con puntatori
- i *resizable array* sono liste di array
- i *balanced tree* sono alberi realizzati con puntatori
- le *tabelle hash* sono tabelle realizzate tramite *funzioni hash,* ossia funzioni che associano una entità a un valore tramite una funzione matematica.

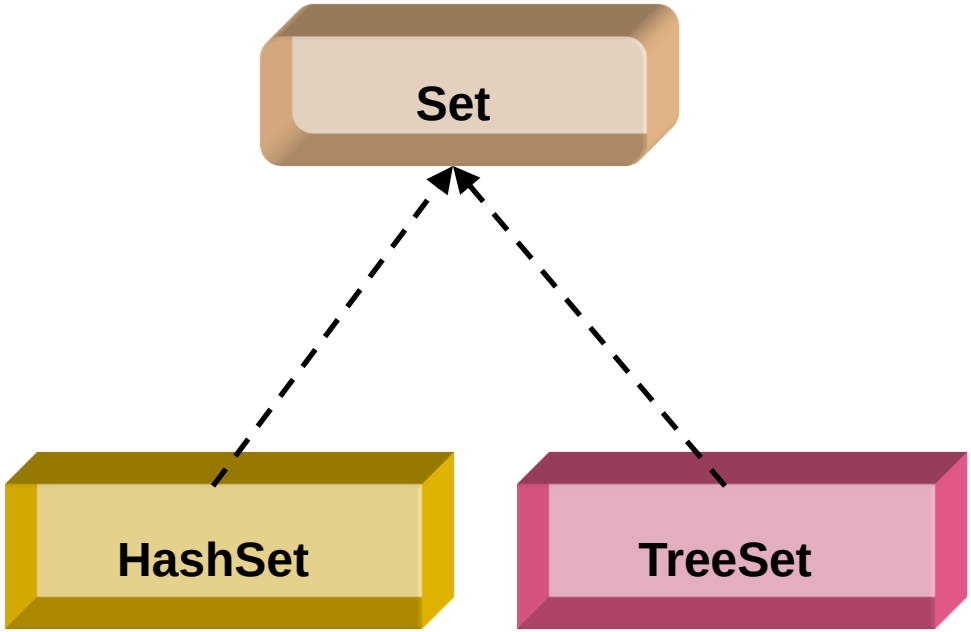


Implementazioni nel Framework Collections

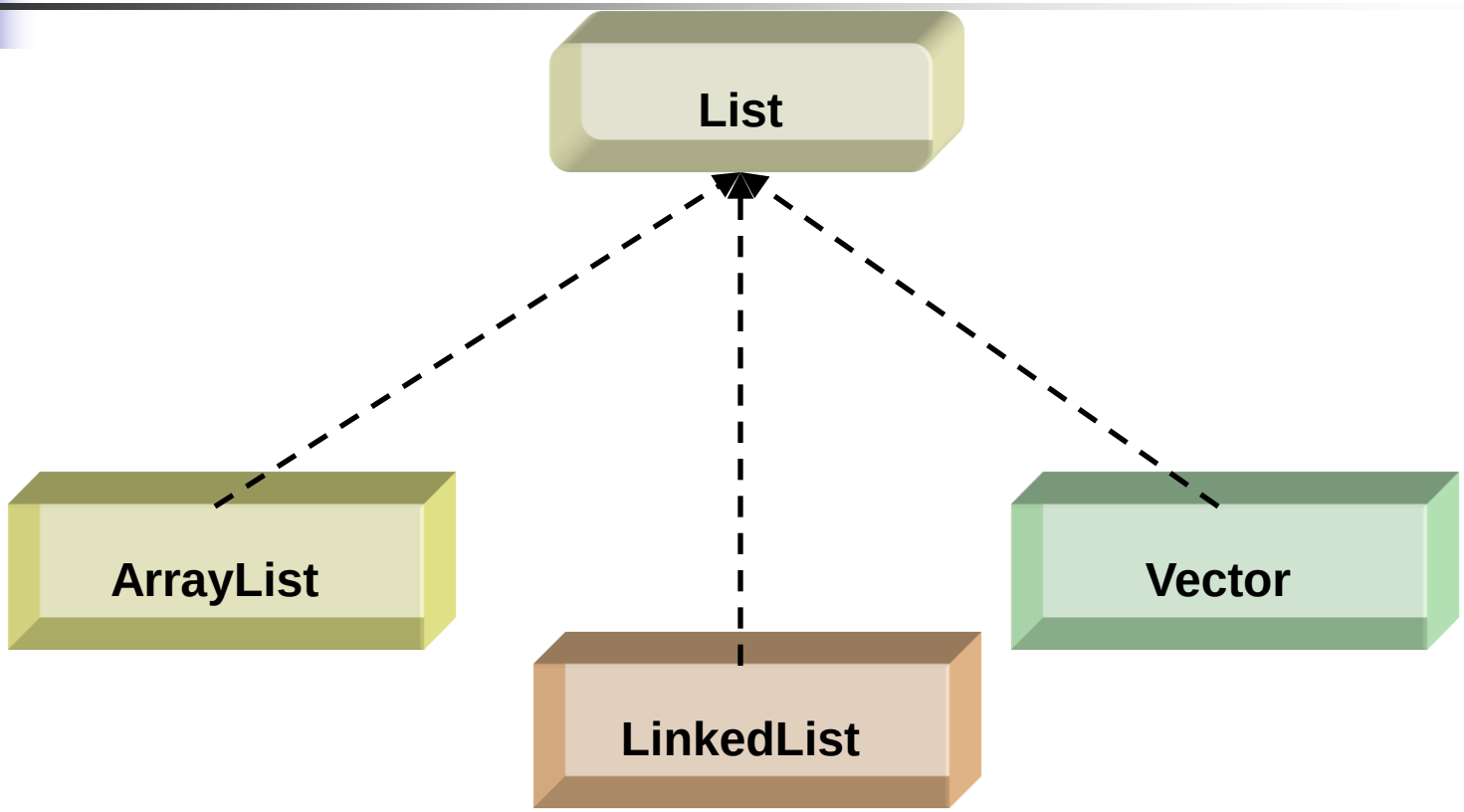
- Scelta di fondo fino alla JDK 1.4: uso del tipo generico Object come mezzo per ottenere contenitori generici
  - i metodi che aggiungono / tolgono oggetti dalle collezioni prevedono un parametro di tipo Object
  - i metodi che cercano / restituiscono oggetti dalle collezioni prevedono un valore di ritorno Object
- Conseguenza:
  - si possono aggiungere/togliere alle/dalle collezioni oggetti di qualunque tipo, TRANNE i tipi primitivi
  - questi ultimi devono prima essere rispettivamente incapsulati in un oggetto (BOXING) / estratti da un oggetto che li racchiuda (UNBOXING)

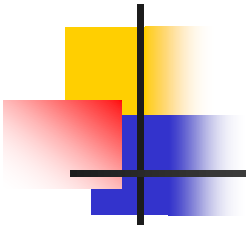
Set Implementations

- HashSet
  - un insieme associato con una hash table
- TreeSet
  - Implementazione di un albero binario bilanciato
  - Impone un ordine nei suoi elementi



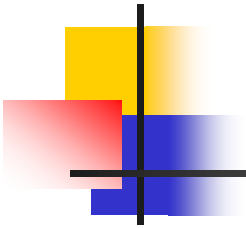
List Implementations





List Implementations

- `ArrayList`
  - Un array ridimensionabile
  - Asincrono (richiede sincronizzazione esplicita)
- `LinkedList`
  - Una lista doppiamente concatenata
  - Può avere performance migliori di `ArrayList`
- `Vector`
  - Un array ridimensionabile
  - Sincrono (i metodi sono già sincronizzati, da preferire ad array per applicazioni multithreaded)

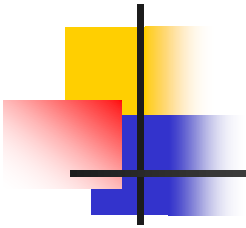


List Implementations

- la rappresentazione con array facilita l'accesso agli elementi data la posizione ma penalizza gli inserimenti e le cancellazioni in mezzo alla lista
- è necessario spostare gli elementi su o giù

Viceversa

- LinkedList è più lenta nell'accesso agli elementi data la posizione
- accedere l'elemento in posizione i richiede la scansione di i riferimenti
- ma è più veloce negli inserimenti e nelle cancellazioni (approssimativamente costano quanto la scansione)

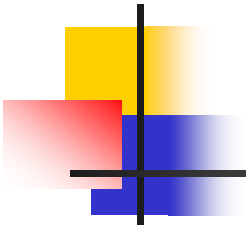


List Implementations

- Test di operazioni
- Basato su circa 1000 operazioni
- tempi misurati in millisecondi

Type	Get	Insert	Remove
array	172	na	na
ArrayList	281	328	30484
LinkedList	5828	109	16
Vector	422	360	30781

fonte: Thinking in Java

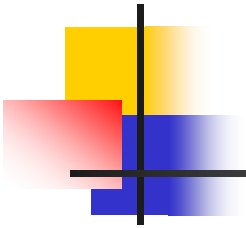


List Implementations

- gli array sono i più veloci, ma non consentono inserimenti e cancellazioni
- ArrayList è veloce nell'accesso agli elementi, lenta in inserimenti e cancellazioni in mezzo
- LinkedList è più lenta nell'accesso, ma decisamente più veloce in inserimenti e canc.
- Vector è più lenta di entrambe e non dovrebbe essere utilizzata

Di conseguenza

- nel caso di liste a bassa dinamica, per ridurre i tempi di scansione è opportuno usare ArrayList
- per liste ad alta dinamica, con frequenti inserimenti e cancellazioni conviene utilizzare LinkedList
- E che succede se devo cambiare tipo ? es: passare da ArrayList a LinkedList



List Implementations

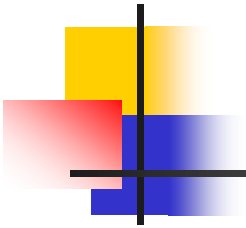
- è opportuno programmare con le interfacce invece che con le implementazioni
- le interfacce riducono l'accoppiamento tra le classi e semplificano i cambiamenti

Nel caso delle liste

- è opportuno utilizzarle per quanto possibile attraverso riferimenti di tipo java.util.List

In questo modo

- le modifiche sono semplificate
- basta cambiare le poche istruzioni in cui gli oggetti di tipo lista sono creati cambiando la classe usata per l'implementazione
- il resto dell'applicazione resta intatta
- i metodi si comportano polimorficamente e viene utilizzata la nuova implementazione



List Implementations

Attenzione

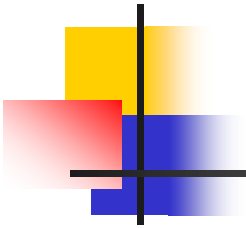
- in questo approccio, quando manipolo la lista devo tenere in considerazione che l'implementazione potrebbe cambiare

In particolare

- devo fare attenzione a non basare la scrittura del codice su una o l'altra delle implementaz.
- Un'operazione critica: la scansione

Il modo tipico di scandire una lista utilizzando indici interi

```
for (int i = 0; i < lista.size(); i++) {
    Object o = lista.get(i);
    // operazioni su o
    }
```



List Implementations

- Questo tipo di scansione
- è particolarmente adatto ad ArrayList (il metodo get viene eseguito rapidamente)
- ma disastrosamente lenta su LinkedList

Perchè ?

- perchè come detto l'accesso all'elemento in posizione i di una LinkedList richiede di scandire i elementi (i operazioni circa)

Detta n la dimensione della lista 1 + 2 + 3 + 4 + ... + n

- pari circa a n(n +1)/2, ovvero dell'ordine di n2
  - es: per una lista di 100 elementi: 5000
  - nel caso di ArrayList: circa 100 operazioni
- In casi normali
  - il problema non sorge (liste piccole)
  - ma in alcuni casi si tratta di un costo di calcolo che può diventare inaccettabile

List Implementations

la scansione attraverso indici interi NON è la scansione più naturale per LinkedList

ArrayList

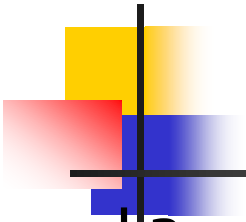
- implementazione basata su indici >> scansione naturale basata su indici

LinkedList

- implementazione basata su riferimenti >> scansione naturale basata su riferimenti

Idealmente

- vorrei che per ciascuna tipologia di lista potesse essere utilizzata automaticamente la scansione più adatta
- senza che il programmatore se ne debba preoccupare
- in questo caso il polimorfismo da solo non basta



List Implementations

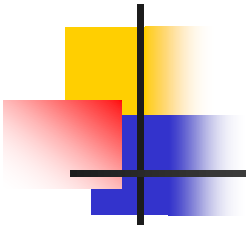
- La scansione della lista è un'operazione che deve necessariamente essere effettuata da un oggetto diverso dalla lista, non posso quindi semplicemente sovrascrivere il metodo "scandisciti()" e utilizzarlo polimorficamente devo necessariamente definire altri oggetti la cui responsabilità è quella di scandire la lista

Soluzione

- utilizzare un oggetto "iteratore"

Iteratore

- oggetto specializzato nella scansione di una lista
- fornisce al programmare un'interfaccia per effettuare la scansione in modo indipendente dalla strategia di scansione concreta (indici, puntatori, ecc.)
- implementa la scansione in modo ottimale per ciascun tipo di lista



List Implementations

L'utilizzo in java.util

- interfaccia java.util.Iterator, che prevede i seguenti metodi
  - Object next() per spostarsi in avanti
  - boolean hasNext() per fermarsi
- esiste poi una implementazione per ArrayList
- ed una implementazione per LinkedList

Iteratore per ArrayList

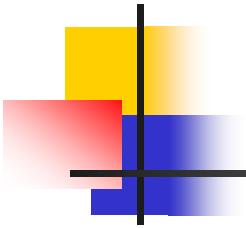
- utilizza indici interi

Iteratore per LinkedList

- scandisce la lista usando i riferimenti

Come si ottiene l'iteratore ?

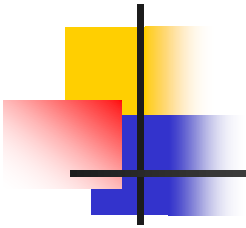
- utilizzando il metodo Iterator iterator() di java.util.List



List Implementations

Dettagli sugli iteratori di java.util

- sostanzialmente si basano sui metodi next() e previous() forniti dalle due liste
- sono però più complicati di quanto si pensa dal momento che consentono anche di modificare la lista durante la scansione attraverso il metodo remove()
- senza doversi preoccupare della consistenza dei riferimenti



List Implementations

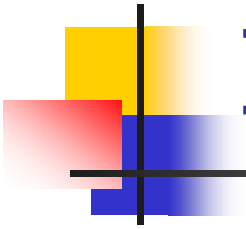
Una novità di J2SE 1.5

- il ciclo for migliorato (“enhanced for loop”)
- un modo sintatticamente compatto per utilizzare un iteratore su una collezione

Sintassi

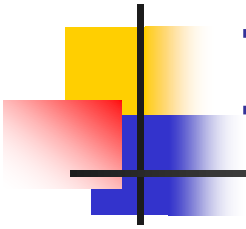
for (<Tipo> <riferimento> : <Collezione>) {

    <operazioni su <riferimento>>}



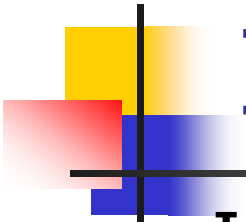
Interfaccia Iterator

- Rappresenta un loop
- Creato da `Collection.iterator()`
- Simile a `Enumeration`
  - Nome di metodi migliorati
  - Permette una operazione `remove()` sul item corrente
- Metodi
  - `boolean hasNext()`
  - `Object next()`
  - `void remove()`
    - `Rimuove l'elemento dalla collezione`



Interfaccia Iterator

- **All Known Subinterfaces:**
  - ListIterator
- **All Known Implementing Classes:**
  - BeanContextSupport.BCSIterator

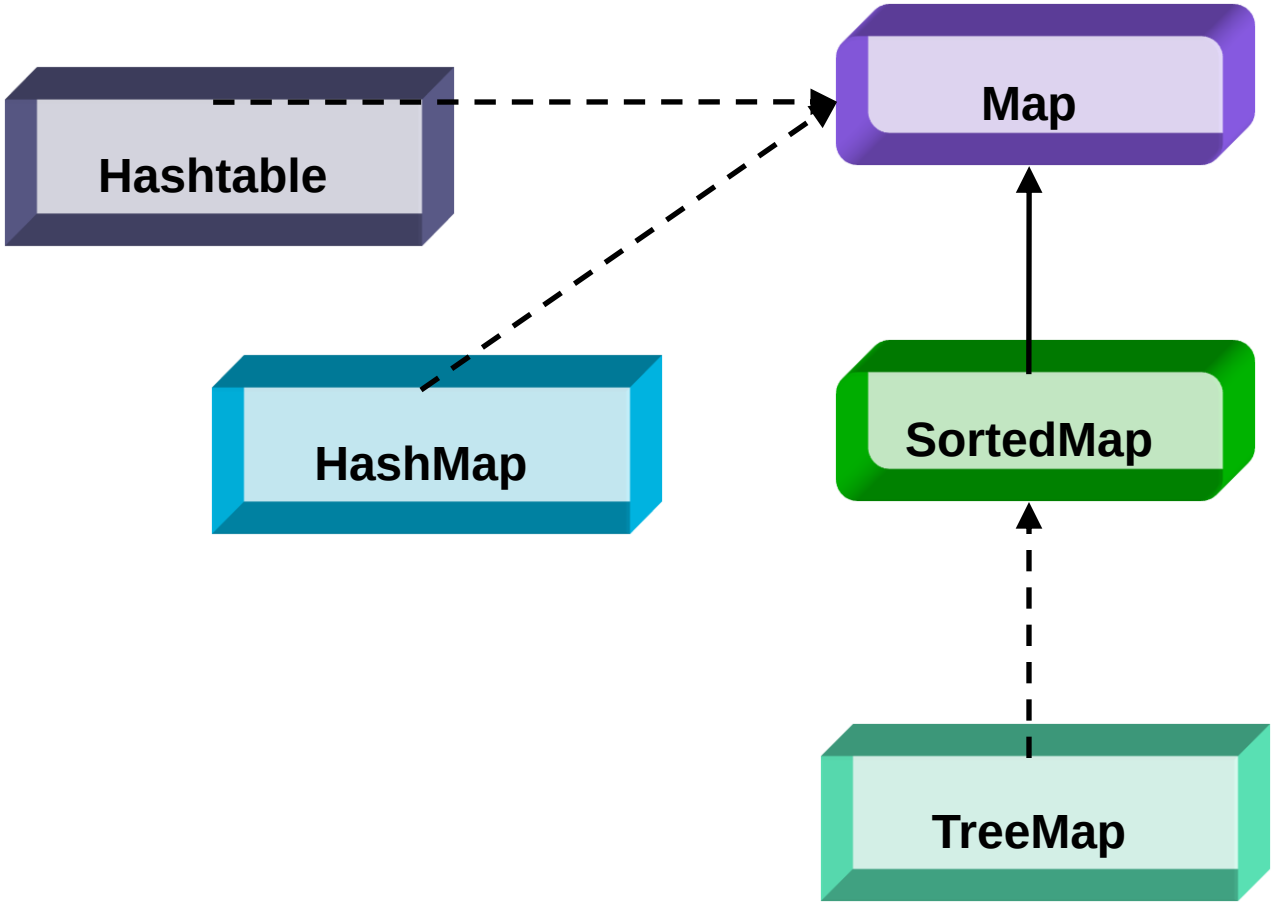


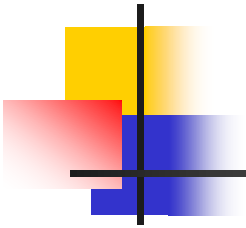
Interfaccia ListIterator

- Interface `ListIterator` extends `Iterator`
- Creata da `List.listIterator()`
- Aggiunge i metodi
  - Attraversa la `List` in ogni direzione
  - Modifica `List` durante l'iterazione
- Methods added:

```
public boolean hasPrevious()
public Object previous()
public int nextIndex()
public int previousIndex()
public void set(Object)
public void add(Object)
```

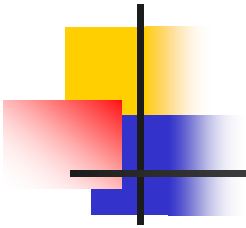
Map Implementations





Map Implementations

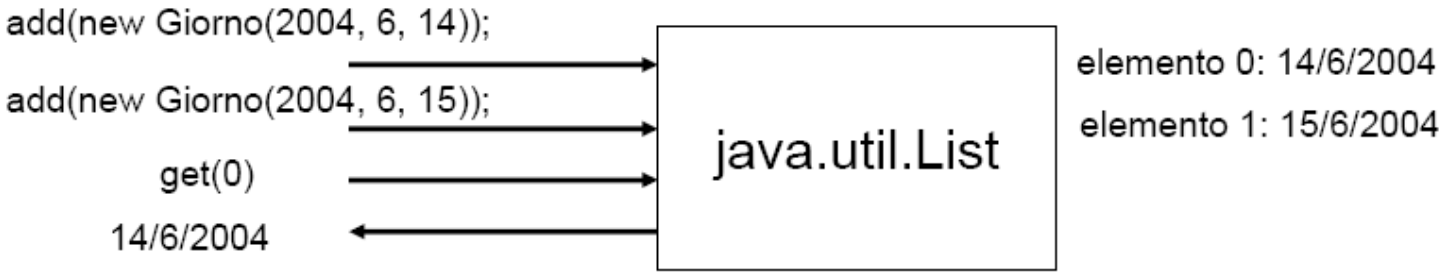
- `HashMap`
  - Un hash table implementazione di `Map`
  - Come `Hashtable`, ma supporta null keys & values
- `TreeMap`
  - Un albero binario bilanciato
  - Impone un ordine nei suoi elementi
- `Hashtable`
  - hash table sincronizzato Implementazione dell'interfaccia `Map`.

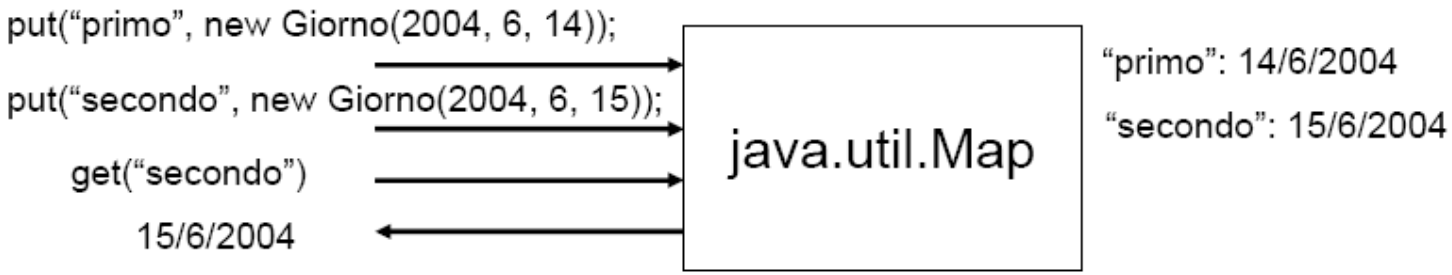


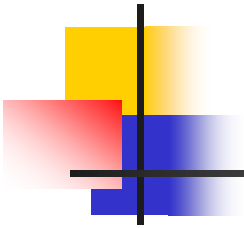
Mappe

- Una mappa, ovvero un dizionario associativo
  - classe java.util.HashMap
  - implementa l'interfaccia java.util.Map
- Dizionario associativo
  - collezione in cui i riferimenti sono salvati con un "nome", detto chiave
  - tipicamente una stringa
  - possono successivamente essere recuperati rapidamente utilizzando la chiave

Mappe


add(new Giorno(2004, 6, 14));
add(new Giorno(2004, 6, 15));
get(0)
14/6/2004
java.util.List
elemento 0: 14/6/2004
elemento 1: 15/6/2004


put("primo", new Giorno(2004, 6, 14));
put("secondo", new Giorno(2004, 6, 15));
get("secondo")
15/6/2004
java.util.Map
"primo": 14/6/2004
"secondo": 15/6/2004



public interface Map<K,V>

Principali metodi

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

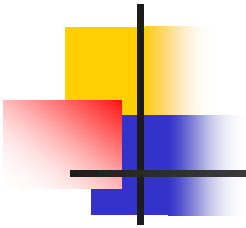
void **putAll**(Map<? extends K,? extends V> m)

V **get**(Object key)

V **remove**(Object key)

Implementazioni

- java.util.HashMap
- java.util.TreeMap
- java.util.Hashtable



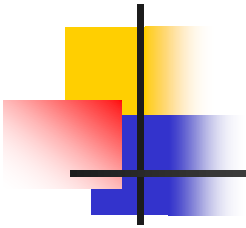
Mappe

Differenze rispetto alle liste

- in una mappa non sono significative le posizioni degli elementi ma le chiavi
- le ricerche sulla base della chiave sono enormemente facilitate (nella lista richiederebbero una scansione)
- utilizzata tipicamente quando più che le scansioni sono importanti le ricerche

Attenzione però

- ad ogni chiave può essere associato un unico oggetto
- put successive con la stessa chiave sostituiscono i valori precedenti
- non può essere usata quando possono esserci più valori per la stessa chiave



HashMap

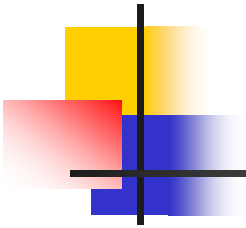
- Un requisito fondamentale per le mappe
- la rapidità di inserimento e cancellazione

L'implementazione fondamentale

- HashMap
  - di gran lunga la più veloce

La tecnica sottostante

- tecnica di hashing ovvero basata su "funzioni di hashing"



HashMap

Funzione di hash

- funzione che trasforma un valore (“chiave”) di lunghezza variabile in uno di lunghezza fissa (“hash” della chiave)

Caratteristica tipica di una funzione hash

- l’hash deve essere calcolabile rapidamente

Classificazione delle funzioni hash

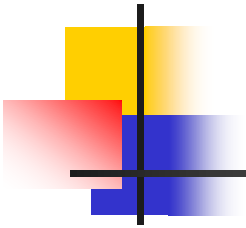
- funzioni con collisioni o prive di collisioni

Funzione priva di collisione

- non ci sono due valori per cui l’hash è uguale
- possibile solo se i valori sono finiti

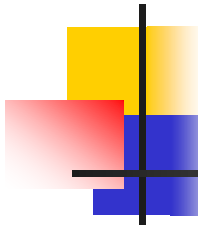
Funzione con collisione

- più valori possono avere lo stesso valore di hash
- caso tipico

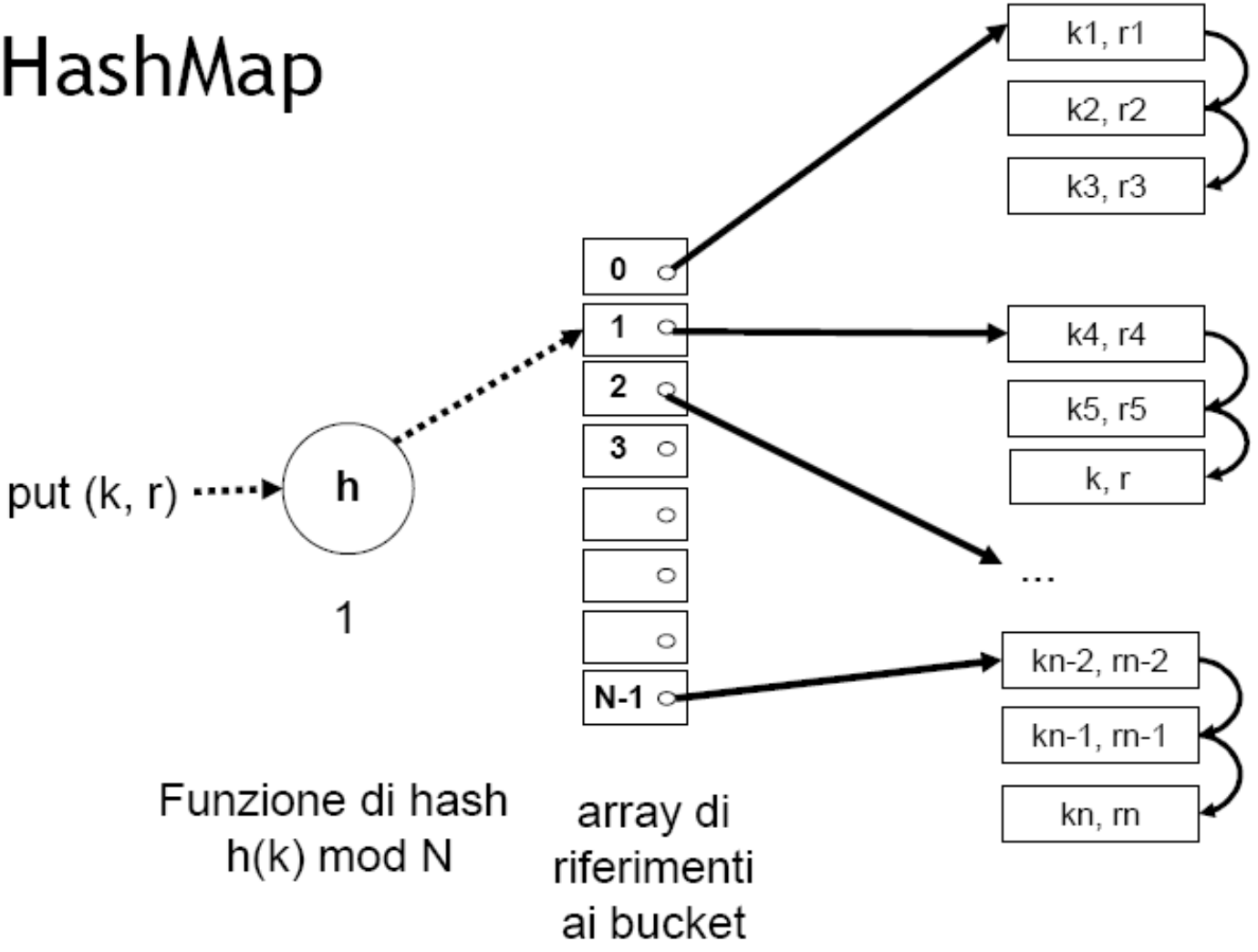


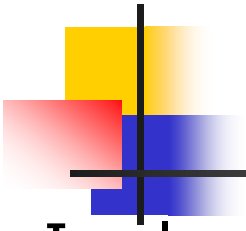
HashMap

- Implementazione di put() nella HashMap
  - la mappa mantiene gli oggetti in un array di N riferimenti a liste (dette "bucket")
  - ogni elemento della lista memorizza una coppia <chiave, riferimento>
  - viene calcolato il valore della funzione di hash sulla chiave e poi viene applicato un operatore modulo per ridurlo ad un numero tra 0 e N – 1
  - in questo modo si ottiene un indice nell'array; la coppia <chiave, riferimento> viene aggiunta in coda al bucket della posizione ottenuta



HashMap


k1, r1
k2, r2
k3, r3
0
1
2
3
k4, r4
k5, r5
k, r
...
N-1
kn-2, rn-2
kn-1, rn-1
kn, rn
put (k, r)
h
1
Funzione di hash
h(k) mod N
array di
riferimenti
ai bucket



HashMap

Implementazione di get() in HashMap

- viene calcolato il valore di hash della chiave per risalire al bucket (indice nell'array)
- viene scandito il bucket e la chiave viene confrontata con ogni chiave; se viene trovata una chiave identica a quella cercata, viene restituito il riferimento, altrimenti viene restituito null

Due operazioni fondamentali

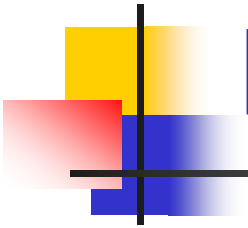
- il calcolo della funzione di hash
- il confronto tra le chiavi

Calcolo della funzione di hash

- viene usato il metodo hashCode() ereditato da Object

Confronto tra le chiavi

- viene utilizzato il metodo equals() ereditato da Object



HashMap

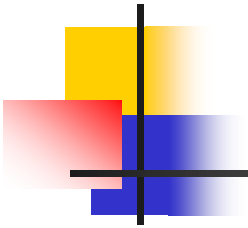
- le implementazioni standard sono basate sull'indirizzo in memoria
- potrebbero non essere quelle adatte a fare hashing in alcuni casi

Nelle classi principali della piattaforma

- sono ridefinite opportunamente quando è necessario

Di conseguenza

- è opportuno utilizzare come chiave per le mappe oggetti di classi note es: String, Integer, ecc.
- Nel caso in cui questo non sia possibile per la classe di oggetti da utilizzare come chiavi è necessario ridefinire opportunamente hashCode() ed equals()



public class **HashMap<K,V>** extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable

- C*apacity: numero di* buckets nella hash table,
- *Load factor* e' la misuradi quanto piena puo ' essere l'hash table prima di un resize (0,75)
- Costruttori

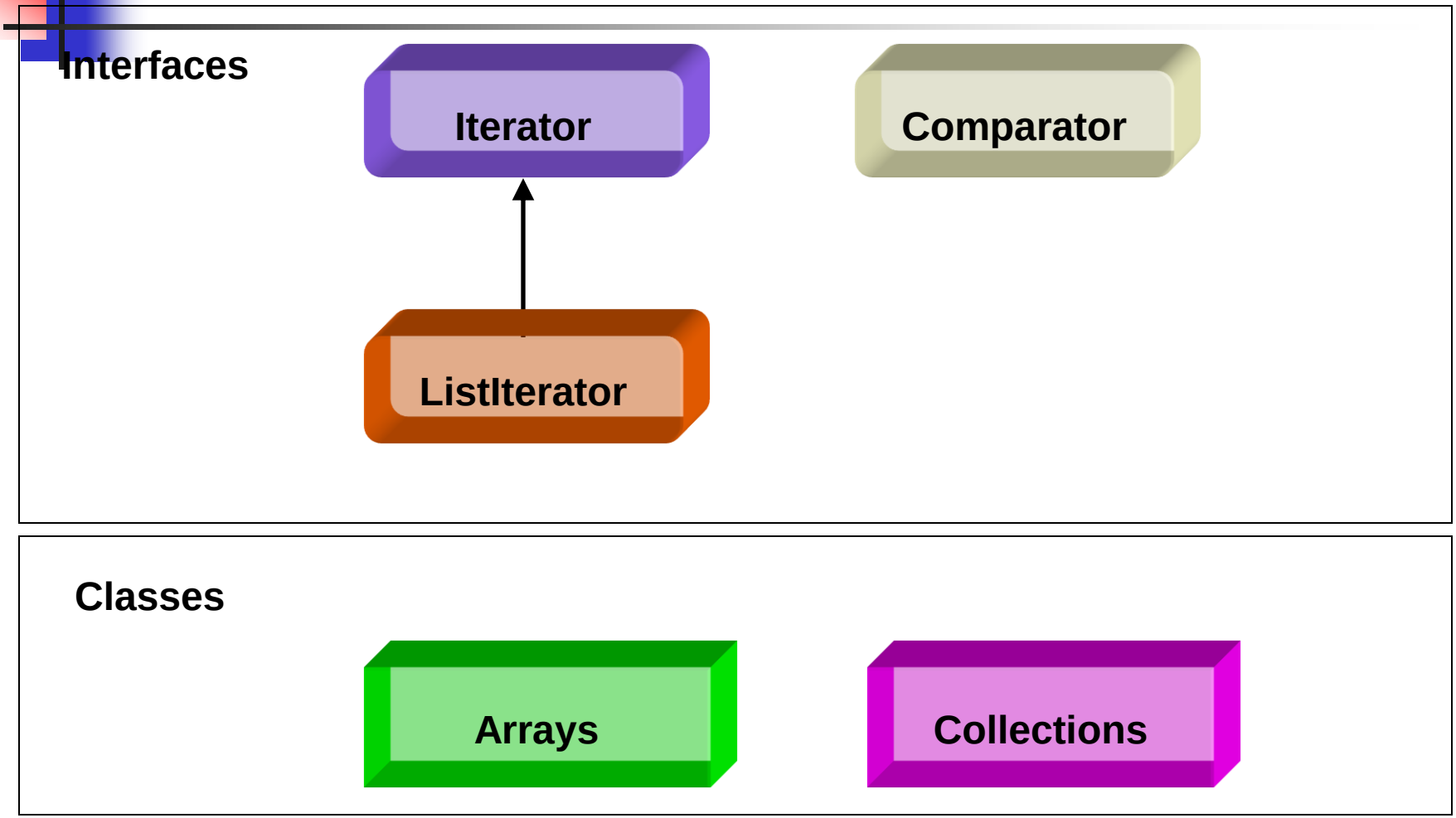
HashMap() costruisce una HashMap vuota con capacity =16 e load factor = 0.75

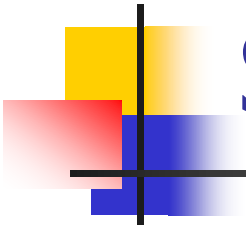
.**HashMap**(int initialCapacity) costruisce una HashMap vuota con capacity = initialCapacity e load factor = 0.75

HashMap(int initialCapacity, float loadFactor) costruisce HashMap con i valori specificati

HashMap(Map<? extends K,? extends V> m) costruisce una HashMap con lo stesso mappings della mappa scecificata.

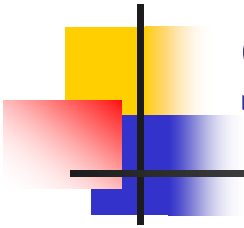
Utility di Collections Framework





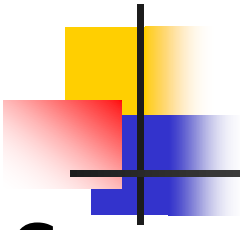
Sorting

- La classe Collections definisce un insieme di metodi statici di utilità generale per le collezioni, fra cui `Collections.sort(list)` metodo statico che utilizza l'ordinamento naturale per list
- `SortedSet`, `SortedMap` interfaces
  - `Collections con elementi ordinati`
  - `Iterators` attraversamento ordinato
- Implementazioni ordinate di `Collection`
  - `TreeSet`, `TreeMap`



Sorting

- Comparable interface
  - Deve essere implementata da tutti gli elementi in SortedSet
  - Deve essere implementata da tutte le chiavi in SortedMap
  - Metodo di comparable, che restituisce un valore minore, maggiore o uguale a zero a seconda che l'oggetto su cui è invocato sia minore, maggiore o uguale a quello passato; l'ordinamento utilizzato è quello naturale per la classe
    - `int compareTo(Object o)`
- Comparator interface
  - Puo utilizzare un ordinamento ad hoc, quando l'interfaccia Comparable non è implementata o l'ordinamento naturale da essa fornito non è adatto agli scopi
  - Metodo di comparator che implementa l'ordinamento ad hoc:
    - `int compare(Object o1, Object o2)`



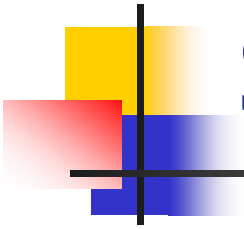
Sorting

Comparable

- A comparable object is capable of comparing itself with another object. The class itself must implements the java.lang.Comparable interface in order to be able to compare its instances.
- Si usa quando il criterio di ordinamento è unico quindi può essere inglobato dentro la classe

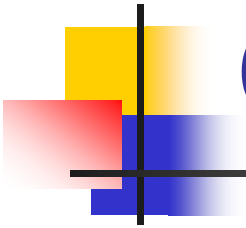
Comparator

- A comparator object is capable of comparing two different objects. The class is not comparing its instances, but some other class's instances. This comparator class must implement the java.lang.Comparator interface.
- Si usa quando si vogliono avere più criteri di ordinamento per gli stessi oggetti o quando l'ordinamento previsto per una data classe non è soddisfacente



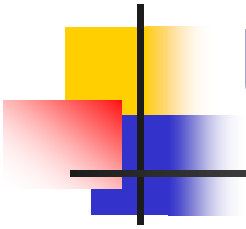
Sorting

- Ordinamento di Arrays
  - Uso di `Arrays.sort(Object[])`
  - Se l'array contiene Objects, essi devono implementare l'intefaccia Comparable
  - Metodi equivalenti per tutti i tipi primitivi
    - `Arrays.sort(int[])`, etc.



Operazioni non supportate

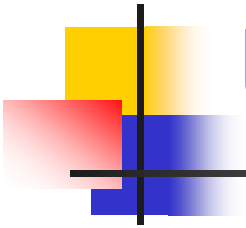
- Un classe può non implementare alcuni particolari metodi di una interfaccia
- UnsupportedOperationException è una runtime (unchecked) exception



Utility Classes - Collections

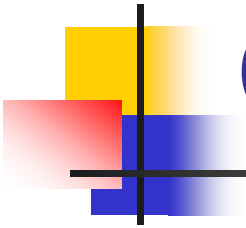
- La classe Collections definisce un insieme di metodi statici di utilità generale
- Static methods:

```
void sort(List)
int binarySearch(List, Object)
void reverse(List)
void shuffle(List)
void fill(List, Object)
void copy(List dest, List src)
Object min(Collection c)
Object max(Collection c)
```



Utility Classes - Arrays

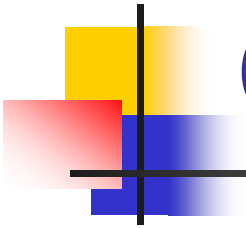
- `Arrays` class
- Metodi statici che agiscono su array Java
  - `sort`
  - `binarySearch`
  - `equals, deepequals` (array di array)
  - `fill`
  - `asList` – ritorna un `ArrayList` composto composta dagli elementi dell'array



Collezioni

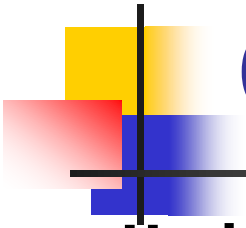
- Collezioni utilizzabili in Java 6.0 (vedere la docs):

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



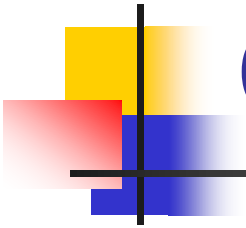
Collezioni

- HashSet class implements the Set interface, backed by a hash table.
- It makes no guarantees as to the iteration order of the set; in particular, **it does not guarantee that the order will remain constant over time**. This class permits the null element.
- This class offers **constant time performance for the basic operations** (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.



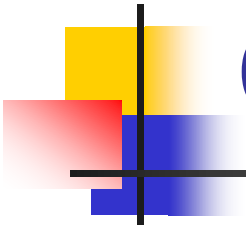
Collezioni

- **HashSet is not synchronized.** If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set: Set s = Collections.synchronizedSet(new HashSet(...));
- The iterators returned by this class's iterator method are **fail-fast**: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the Iterator throws a ConcurrentModificationException. Thus the iterator fails quickly and cleanly, rather than risking non-deterministic behavior.
- Note that the fail-fast behavior cannot be hardly guaranteed in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a **best-effort basis**. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.



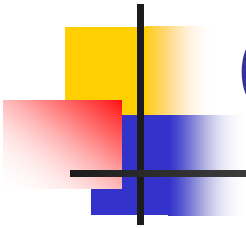
Collezioni

- **LinkedHashSet** is an Hash table and linked list implementation of the Set interface, with **predictable iteration order**.
- This implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion-order). Note that insertion order is not affected if an element is re-inserted into the set. (An element e is reinserted into a set s if s.add(e) is invoked when s.contains(e) would return true immediately prior to the invocation.)
- This implementation spares its clients from the unspecified, **generally chaotic ordering** provided by HashSet, without incurring the increased cost associated with TreeSet.



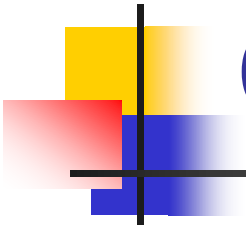
Collezioni

- **HashMap** is an Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and key.
- The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.
- This class makes **no guarantees as to the order of the map**; in particular, it does not guarantee that the order will remain constant over time.
- This implementation provides **constant-time performance for the basic operations** (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.



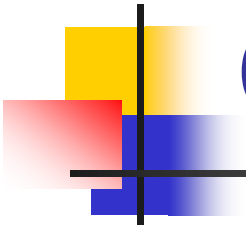
Collezioni

- An instance of HashMap hence has two parameters that affect its performance: **initial capacity** and **load factor**.
- The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created.
- The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.



Collezioni

- As a general rule, the **default load factor** (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.
- If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.



Collezioni

- The **Stack** class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.
- A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class.
- For instance, **ArrayDeque** is a resizable-array implementation of the Deque interface. Array deques have no capacity restrictions; they grow as necessary to support usage. They are not thread-safe; in the absence of external synchronization, they do not support concurrent access by multiple threads. Null elements are prohibited. This class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue.