Java

collezioni di oggetti

G. Prencipe
prencipe@di.unipi.it

Introduzione

- In genere, i programmi creano oggetti basandosi su criteri noti solo a run-time
- Cioè, non è possibile in genere conoscere a priori il numero di oggetti che verranno creati durante l'esecuzione
- Quindi c'è bisogno di un modo per gestire dinamicamente gli oggetti creati
- Java offre gli *array* come tipo predefinito per contenere oggetti
- Oltre ad essi offre le **Collection**

Arrays

- È la struttura dati più semplice per contenere oggetti
- Ci sono due motivi per volerli usare rispetto ad altri *contenitori* d'oggetti
  - Efficienza
  - Tipo

Arrays

- *Efficienza*
  - È il modo più efficiente offerto da Java per memorizzare e accedere direttamente oggetti
  - Sono sequenze lineari, e quindi gli accessi sono veloci
  - Le dimensioni sono però fisse e non possono essere cambiate una volta definite
  - Si può pensare di estendere un array quando si riempie
    - Questo è quello che fa la classe **ArrayList**, che vedremo
    - Risulta comunque meno efficiente degli array

Arrays

- Inoltre sugli array vengono effettuati controlli sui limiti
  - Se si accedono elementi oltre i limiti definiti, si ottiene una eccezione
  - Cosa che non avviene in C++

Arrays

- *Tipo*
  - A differenza degli altri contenitori offerti in Java, gli array sono definiti per contenere oggetti dello *stesso* tipo
  - Si ottiene errore in compilazione se si cerca di inserire in un array elementi di tipo diverso da quello definito

Arrays

- L'identificatore di array è un riferimento a un oggetto (sull'heap) che contiene riferimenti ad altri oggetti
- Parte dell'oggetto array è una *variabile* di sola lettura **length** che memorizza il numero di elementi contenuti nell'array
- L'unico altro modo per accedere l'oggetto array è tramite **[]**

Arrays

- Gli array possono contenere direttamente tipi primitivi
  - A differenza degli altri contenitori che possono contenere solo riferimenti a **Object**
  - In questo caso per contenere tipi primitivi bisogna fare riferimento alle loro classi relative (es. **Integer** per **int**)

Arrays

- È possibile avere come tipo di ritorno di un metodo un array

```
public String[] restituisciArray(int n){
  String[] a = new String[n];
        ….
  return a;
}
```

La classe **Arrays**

- In **java.util** si trova la classe **Arrays**, che fornisce una serie di metodi **static**i che eseguono funzioni di utilità per array
- Ci sono 4 metodi di base
  - **equals()**, per confrontare due array
  - **fill()**, per riempire un array con un certo valore
  - **sort()**, per ordinare l'array
  - **binarySearch()**, per cercare un elemento in un array ordinato
- Essi sono definiti (overload) per tutti i tipi primitivi e per **Object**

La classe **Arrays**

- Inoltre c'è un metodo **asList()** che prende un array come argomento e lo trasforma in un contenitore **List**, che vedremo
- Non ci sono tutti i metodi d'utilità che ci si aspetta, come ad esempio la stampa del contenuto di un array
  - Eviterebbe la scrittura di un **for** ogni volta
  - Quindi si può pensare di estendere la classe **Arrays** con le funzioni d'utilità che pensiamo necessarie
  - Bisogna stare attenti però a definire le funzioni per *tutti* i tipi primitivi (overload)

La classe **Arrays**

- Riempire un array: **static Arrays.fill()**
  - Questo metodo semplicemente riempie un array con un valore passato come argomento
  - È possibile passare oltre al valore gli estremi di un intervallo dell'array da riempire: vengono riempite solo le posizioni incluse nell'intervallo
  - Con **a** è un array di **boolean**, e **b** di **String**he,

```
Arrays.fill(a, true);
Arrays.fill(b, 3, 5, "ciao");
```

La classe **Arrays**

- Per copiare un array, la libreria standard di Java offre il metodo statico **System.arraycopy()**
- Anche in questo caso, c'è overloading per gestire tutti i tipi

  System.arraycopy(a1,offset1,a2,offset2,numEl);
  - **a1** e **a2** sono gli array sorgente e destinazione
  - **offset1** e **offset2** sono gli scostamenti in **a1** e **a2** da dove iniziare a leggere e in cui iniziare a scrivere, rispettivamente
  - **numEl** è il numero di elementi da copiare
    - Ogni violazione nei limiti degli array produce una eccezione

La classe **Arrays**

- Si possono copiare sia array contenenti tipi primitivi che oggetti
- Comunque, nel caso di oggetti, vengono copiati *solo* i riferimenti e *non* gli oggetti
  - *Shallow copy*

La classe **Arrays**

- Confrontare array: **static Arrays.equals()**
  - Questo metodo confronta due array
  - Affinché due array risultino uguali, devono avere lo stesso numero di elementi, e ogni elemento nel primo array deve essere *equivalente* al corrispondente elemento nel secondo array
    - I confronti sono effettuati invocando **equals()** per ogni coppia di elementi
    - Per i tipi primitivi, viene invocato **equals()** sulla classe relativa
      - Es.: **Integer.equals()** per **int**

La classe **Arrays**

- La situazione si complica quando si vogliono effettuare operazioni più complesse, tipo l'ordinamento
- Il problema è dato dal fatto che i confronti tra **Object** non sono ovvi
- Per permettere di poter confrontare oggetti di qualsiasi tipo, bisogna implementare l'interfaccia **Comparable**

Interfaccia **comparable**

- L'interfaccia **java.lang.Comparable** ha un solo metodo: **compareTo()**
- Questo metodo prende come argomento un altro **Object** e restituisce
  - Un valore negativo, se l'oggetto corrente è *minore* di quello passato come argomento
  - Zero, se i due oggetti sono uguali
  - Un valore positivo, se l'oggetto corrente è *maggiore* di quello passato come argomento

Interfaccia **Comparable**

- Implementando questa interfaccia, è possibile invocare il metodo **Arrays.sort()** che permette di ordinare arrays di oggetti
  - Per conoscere gli algoritmi utilizzati per ordinare, consultare la documentazione
- Infatti, questo metodo si basa sul fatto che tutti gli oggetti contenuti nell'array implementino **compareTo()**
  - Per i tipi primitivi non è necessario
  - Per altri oggetti, bisogna definirlo, altrimenti otteniamo errore in compilazione

Esercizio

- Creare una classe **Comp** che implementa **Comparable**
- **Comp** definisce una coppia di interi, **h** e **k**
- **compareTo()** realizza l'ordinamento lessicografico a coppie
  - (h,k)<(u,v) ⇔ (h<u) o (h=u e k<v)
  - (h,k)=(u,v) ⇔ (h=u) e (k=v)
  - (h,k)>(u,v) ⇔ (h>u) o (h=u e k>v)
- Nel main, creare un array di oggetti **Comp**, e poi ordinarlo utilizzando **Arrays.sort()**

Interfaccia **Comparator**

- Se gli oggetti contenuti in un array non hanno implementata l'interfaccia **Comparable** (perché ad esempio non sono tipi definiti da noi), allora si ricorre all'interfaccia **Comparator**
  - Si definisce una nuova classe che implementa **Comparator**
- Questa interfaccia ha due metodi
  - **compare()** e **equals()**
  - Non è necessario implementare **equals()**, dato che essa può essere ereditata da **Object**

Esercizio -- **Comparator**

- Entrambi questi metodi prendono come argomento *due* **Object**, e li confrontano
- Provare a scrivere una classe **CompComparator** che implementa **Comparator** e confronta due oggetti di tipo **Comp** definito nell'esercizio precedente
  - È sufficiente implementare **compare()**
- Testare la classe ordinando l'array definito nell'esercizio precedente invocando l'opportuno metodo di **sort()**

La classe **Arrays**

- Quindi, il metodo **Arrays.sort()** può prendere un array (in questo caso gli oggetti devono implementare **Comparable**) o un array e un **Comparator**
- Se invochiamo **Arrays.sort()** su un array di **String**he, l'ordinamento ottenuto è *lessicografico*
  - Prima le parole che iniziano con lettere maiuscole e poi quelle che iniziano con le minuscole
  - Se volessimo ordinare indipendentemente dal *case* (ordinamento *alfabetico*), bisognerebbe definire un **Comparator** che fa questo

La classe **Arrays**

- Per cercare in un array ordinato: **static Arrays.binarySearch()**
  - Prende come argomenti l'array in cui cercare e la chiave (elemento) da cercare
  - Restituisce la posizione nell'array dove si trova l'elemento cercato, se esso esiste nell'array
  - Altrimenti restituisce un valore negativo che rappresenta la posizione in cui dovrebbe trovarsi l'elemento, mantenendo l'array ordinato
    - Restituisce **-(punto inserimento)-1**, dove **punto inserimento** è l'indice del primo elemento più grande della chiave da cercare, o **a.size()** se tutti gli elementi sono più piccoli della chiave

La classe **Arrays**

- Se nell'array vi sono elementi duplicati, non viene fornita alcuna garanzia su quale di essi verrà trovato
- Per gestire una lista ordinata di elementi non duplicati, bisogna usare **TreeSet** o **LinkedHashSet**, che vedremo dopo
- Se un array di oggetti è stato ordinato con un **Comparator**, bisogna includere lo stesso **Comparator** per effettuare la **binarySearch()**
  - Altrimenti, il metodo si aspetta che sia implementato **Comparable**

Introduzione ai contenitori

- I *contenitori* aumentano significativamente la potenza programmativa
- In Java2 la libreria di contenitori gestisce modi per *mantenere oggetti*, e si divide in due filoni
  - **Collection**: gruppo di elementi individuali, con spesso associata una regola applicata ad essi
    - Es.: un Set non può avere elementi duplicati
  - **Map**: un gruppo di oggetti definiti come coppie *chiave-valore*
    - Come l'array associativo
- Tutti i contenitori hanno un metodo **toString()** che permette loro di essere stampati senza problemi
  - A differenza degli array

Contenitori

- Le **Collection** contengono un elemento in ogni locazione
  - Elementi individuali
  - Ne sono esempi **List**, **Set**
  - Per aggiungere elementi a qualsiasi **Collection** si utilizza il metodo **add()**
- Le **Map** contengono coppie *chiave-valore*, come un piccolo database
  - Ne è un esempio l'**HashMap**
  - Per inserire elementi in una **Map** si utilizza il metodo **put()**

Contenitori

- Il metodo **fill()** riempie sia **Collection** che **Map**
- Il metodo **toString()** serve per convertire in **String**a sia **Collection** che **Map**
  - La **String**a per una **Collection** è racchiusa da parentesi quadre, con gli elementi separati da virgole
  - La **String**a per una **Map** è racchiusa da parentesi graffe, con ogni coppia rappresentate dalla stringa *chiave=valore*

Contenitori -- **fill()**

- **fill()** è un metodo **static**o della classe **Collection**, e semplicemente duplica un elemento in una collezione
  - Esiste per **List**, ma non per **Set** o **Map**
  - Inoltre serve solo a *rimpiazzare* elementi che già esistono
    - Cioè, *non* aggiunge nuovi elementi

Contenitori

- Lo svantaggio nell'utilizzo dei contenitori in Java è legato al fatto che si perdono le informazioni sul tipo degli oggetti quando essi vengono inseriti
- Infatti, i contenitori sono progettati per contenere riferimenti a **Object**, per renderli generici e flessibili
- In questo modo
  - Non ci sono restrizioni sui tipi degli oggetti da inserire in un contenitore
  - Quando si estraggono elementi da un contenitore, bisogna eseguire un *cast* per recuperare la specializzazione perduta

Contenitori e **String**he

- Quindi, se sbagliamo il *cast*, otteniamo una eccezione
- Per le **String**he, il compilatore ci semplifica le cose
- Infatti, se il compilatore si aspetta una **String**a e non ne ottiene una, automaticamente invoca **toString()**, che esiste anche in **Object**

Contenitori

- Per ottenere l'effetto di poter controllare i tipi degli oggetti che inseriamo in un contenitore, dobbiamo definirne uno che esplicitamente accetta oggetti di un certo tipo
- In questo modo i controlli vengono fatti a tempo di compilazione
- Vediamo un esempio

Esempio

```
public class TipoSpecList {
  private List list = new ArrayList();
  public void add(MioTipo m) { list.add(m); }
  public MioTipo get(int index) {
    return (MioTipo)list.get(index);
  }
  public int size() { return list.size(); }
} ///:~
```

- Nota: non viene esteso **ArrayList()**, altrimenti sarebbe sempre permesso di aggiungere **Object**
  - Usiamo invece composizione

Iteratori

- I due compiti fondamentali dei contenitori sono di inserire elementi ed estrarli
- Consideriamo l'esempio di **ArrayList**
  - Si inserisce con **add()**
  - Un modo per estrarre e **get()**
- Se però a un certo punto vogliamo cambiare implementazione e sostituire **ArrayList** con **Set**, **get()** non va più bene
- Per rendere le cose più flessibili si utilizzano gli *iteratori*

Iteratori

- Un *iteratore* è un oggetto il cui compito è quello di muoversi lungo una sequenza di oggetti e selezionare ogni elemento di quella sequenza, senza che il programmatore si preoccupi della struttura di quella sequenza
- Sono oggetti "light-weight": sono cioè economici da creare
  - Hanno quindi varie limitazioni
    - Es.: alcuni iteratori possono scorrere la sequenza in una sola direzione

Iteratori

- L'interfaccia **java.util.Iterator** è un esempio di iteratori con varie limitazioni
  - Un **Iterator** viene restituito da un contenitore invocando il metodo **iterator()** (del contenitore). Esso è pronto a restituire il primo elemento nella sequenza alla prima invocazione del suo metodo **next()**
  - Il succesivo elemento nella sequenza si ottiene con **next()**
  - **hasNext()** controlla se ci sono successivi elementi nella sequenza
  - **remove()** elimina l'ultimo elemento restituito dall'iteratore

Esempio

```
public class Printer {
  static void printAll(Iterator e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
} ///:~
```

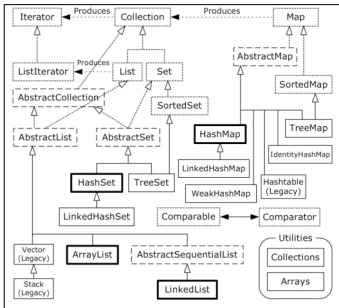
- **Printer** serve a stampare tutti gli elementi in una certa sequenza
- Non si ha alcuna informazione sui tipi contenuti nella sequenza
- Si ha solo un iteratore
- Notare le invocazioni automatiche a **toString()** nella **println**

Attenzione

- Attenzione a ridefinire la **toString()**
- Se nella sua ridefinizione si invoca **this** in una **String**a, il compilatore invoca automaticamente **toString()** e si ottiene un effetto di ricorsione infinita

```
public class InfiniteRecursion {
  public String toString() {
    return " InfiniteRecursion address: " + this + "\n";
  }
```

Tassonomia contenitori



- Rettangoli
  - Puntini: interfacce
  - Trattini: abstract
  - Solido: concrete
- Frecce
  - Puntini: classe implementa interfaccia
  - Solido: classe produce oggetti della classe a cui punta la freccia

- Le interfacce che servono a contenere oggetti sono **Collection**, **List**, **Set** e **Map**

Contenitori

- Generalmente si scrive codice per queste interfacce, e si specifica il tipo che si utilizza al momento della creazione List x = new LinkedList()
- Si può definire anche List x = new LinkedList()
- Ma lo scopo (è la bellezza) delle interfacce (cioè, in questo caso, di definire **List**) è di mantenere il codice generico
- Infatti in questo modo è possibile modificare successivamente l'implementazione (usando ad esempio un **ArrayList** invece di una **LinkedList**) senza modificare nient'altro
  - Esaminare il grafico di prima per convincersene

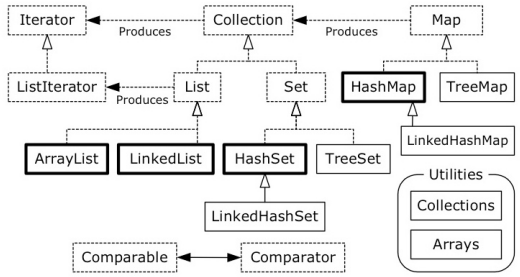
Contenitori

- Nella gerarchia ci sono una seria di classi il cui nome inizia con **Abstract**
- Sono classi che implementano parzialmente delle interfacce
- Quindi, se vogliamo implementare un **Set**, possiamo *non* implementare *tutti* i metodi, ma si può ereditare da **AbstractSet** e fare meno lavoro

Contenitori

- Comunque, nel diagramma di prima, in genere siamo interessati solo alle interfacce e alle classi concrete
- Tipicamente infatti si costruisce un oggetto di una classe concreta, poi si fa upcast alla corrsipondente interfaccia, e poi si utilizza l'interfaccia

Tassonomia semplificata



Semplice esempio

```
public class SimpleCollection {
  public static void main(String[] args) {
    // Upcast perché vogliamo
    // lavorare con le Collection
    Collection c = new ArrayList();
    for(int i = 0; i < 10; i++)
      c.add(Integer.toString(i));
    Iterator it = c.iterator();
    while(it.hasNext())
      System.out.println(it.next());}}///:~
```

Funzionalità di **Collection**

- Come mostrato nel grafico precedente, ogni cosa che si può fare con una **Collection**, può essere fatta da un **Set** o **List**
- **Map** non è ereditato da **Collection**, e quindi deve essere trattato separatamente

boolean add(Object)	Ensures that the container holds the argument. Returns false if it doesn't add the argument. (This is an "optional" method, described later in this chapter.)
boolean addAll(Collection)	Adds all the elements in the argument. Returns **true** if any elements were added. ("Optional.")
void clear( )	Removes all the elements in the container. ("Optional.")
boolean contains(Object)	**true** if the container holds the argument.
boolean containsAll(Collection)	**true** if the container holds all the elements in the argument.
boolean isEmpty( )	**true** if the container has no elements.
Iterator iterator( )	Returns an **Iterator** that you can use to move through the elements in the container.
boolean remove(Object)	If the argument is in the container, one instance of that element is removed. Returns **true** if a removal occurred. ("Optional.")
boolean removeAll(Collection)	Removes all the elements that are contained in the argument. Returns **true** if any removals occurred. ("Optional.")
boolean retainAll(Collection)	Retains only elements that are contained in the argument (an "intersection" from set theory). Returns **true** if any changes occurred. ("Optional.")
int size( )	Returns the number of elements in the container.
Object[] toArray( )	Returns an array containing all the elements in the container.
Object[] toArray(Object[] a)	Returns an array containing all the elements in the container, whose type is that of the array a rather than plain **Object** (you must cast the array to the right type).

Collection

- Non c'è **get()** per l'accesso diretto agli elementi
  - Dato che **Collection** include anche **Set**
- Per esaminare gli elementi di una **Collection** bisogna usare un iteratore

Funzionalità di **List**

- Tipicamente in una **List** si inseriscono elementi con **add()** e si estraggono con **get()**
- Troviamo implementate già due tipi di **List**
  - **ArrayList**
  - **LinkedList**

Differenti **List**

List (interface)	Order is the most important feature of a **List**; it promises to maintain elements in a particular sequence. **List** adds a number of methods to **Collection** that allow insertion and removal of elements in the middle of a **List**. (This is recommended only for a **LinkedList**.) A **List** will produce a **ListIterator**, and using this you can traverse the **List** in both directions, as well as insert and remove elements in the middle of the **List**.
ArrayList*	A **List** implemented with an array. Allows rapid random access to elements, but is slow when inserting and removing elements from the middle of a list. **ListIterator** should be used only for back-and-forth traversal of an **ArrayList**, but not for inserting and removing elements, which is expensive compared to **LinkedList**.
LinkedList	Provides optimal sequential access, with inexpensive insertions and deletions from the middle of the **List**. Relatively slow for random access. (Use **ArrayList** instead.) Also has **addFirst( )**, **addLast( )**, **getFirst( )**, **getLast( )**, **removeFirst( )**, and **removeLast( )** (which are not defined in any interfaces or base classes) to allow it to be used as a stack, a queue, and a deque.

- **LinkedList** è più potente di un **ArrayList**

Esercizio -- creare pile e code

- Una pila offre metodi **push**, **pop** e **top** per inserire e eliminare elementi
- Una coda inserisce elementi in testa (**put**) e li estrae dalla coda (**get**)
- Entrambe possono essere implementate da una **LinkedList**
- Provare a implementare pile e code

Funzionalità di **Set**

Set (interface)	Each element that you add to the **Set** must be unique; otherwise the **Set** doesn't add the duplicate element. **Object**s added to a **Set** must define **equals( )** to establish object uniqueness. **Set** has exactly the same interface as **Collection**. The **Set** interface does not guarantee it will maintain its elements in any particular order.
HashSet*	For **Set**s where fast lookup time is important. **Object**s must also define **hashCode( )**.
TreeSet	An ordered **Set** backed by a tree. This way, you can extract an ordered sequence from a **Set**.
LinkedHashSet (JDK 1.4)	Has the lookup speed of a **HashSet**, but maintains the order that you add the elements (the insertion order), internally using a linked list. Thus, when you iterate through the **Set**, the results appear in insertion order.

- Set ha la stessa interfaccia di **Collection**, ma ha comportamenti diversi
- Un **Set** infatti non contiene oggetti duplicati

Funzionalità di **Set**

- **TreeSet**
  - Tiene gli elementi ordinati utilizzando una albero red-black
- **HashSet**
  - Tiene gli elementi ordinati utilizzando una funzione hash
- **LinkedHashSet**
  - Usa *hashing* per velocizzare la ricerca
- Quindi queste tre strutture dati conservano gli elementi in modo diverso, quindi i risultati delle estrazioni non sono gli stessi

SortedSet

- Con un **SortedSet** (**TreeSet** è l'unico disponibile nella libreria) gli elementi sono tenuti ordinati
- Quindi ci sono metodi aggiuntivi
  - **Comparator comparator()**
    - Restituisce il **comparator** utilizzato per il **Set**, o **null** per l'ordinamento naturale degli oggetti
  - **Object first()**, **Object last()**
  - **SortedSet subSet(fromEl,toEl)**
  - **SortedSet headSet(toElement)**
    - Tutti gli elementi fino a **toElement**
  - **SortedSet tailSet(fromElement)**
    - Tutti gli elementi da **fromElement**

Funzionalità di **Map**

- Con un **ArrayList** possiamo selezionare oggetti tramite un indice (intero)
- Se vogliamo selezionare oggetti utilizzando un criterio diverso bisogna ricorrere all'interfaccia **Map**
  - Noti anche come *dizionari* o *array associativi*
  - Permettono di selezionare oggetti utilizzando un altro oggetto

Funzionalità di **Map**

- Con **put(Object chiave, Object valore)** si inserisce un **valore** associandolo a una **chiave**
- Con **get(Object chiave)** si ottiene il **valore** associato alla **chiave**
- È possibile controllare se una **Map** contiene una chiave o un valore con **containsKey()** e **containsValue()**, rispettivamente

Funzionalità di **Map**

- Nella libreria standard di Java esistono differenti tipi di Map
  - **HashMap**, **TreeMap**, **LinkedHashMap**, **WeakHashMap**, **IdentityHashMap**
- Condividono tutti la stessa interfaccia di **Map**, ma differsicono nei comportamenti, incluso *efficienza*, *ordine* con cui le coppie sono memorizzate e presentate, per quanto *tempo* gli oggetti sono tenuti nella mappa, e come l'*uguaglianza* tra le chiavi è stabilita

Funzionalità di **Map**

Map (interface)	Maintains key-value associations (pairs), so you can look up a value using a key.
HashMap*	Implementation based on a hash table. (Use this instead of **Hashtable**.) Provides constant-time performance for inserting and locating pairs. Performance can be adjusted via constructors that allow you to set the *capacity* and *load factor* of the hash table.
LinkedHashMap (JDK 1.4)	Like a **HashMap**, but when you iterate through it you get the pairs in insertion order, or in least-recently-used (LRU) order. Only slightly slower than a **HashMap**, except when iterating, where it is faster due to the linked list used to maintain the internal ordering.
TreeMap	Implementation based on a red-black tree. When you view the keys or the pairs, they will be in sorted order (determined by **Comparable** or **Comparator**, discussed later). The point of a **TreeMap** is that you get the results in sorted order. **TreeMap** is the only **Map** with the **subMap( )** method, which allows you to return a portion of the tree.
WeakHashMap	A map of *weak keys* that allow objects referred to by the map to be released; designed to solve certain types of problems. If no references outside the map are held to a particular key, it may be garbage collected.
IdentityHashMap (JDK 1.4)	A hash map that uses == instead of **equals( )** to compare keys. Only for solving special types of problems; not for general use.

- Per aumentare l'efficienza nelle ricerche si utilizzano *codici hash*
- *Tutti* gli oggetti Java possono produrre un codice hash, tramite il metodo **hashCode()** in **Object**

Esercizio -- **HashMap**

- Scrivere un programma che verifichi che la classe **Random** genera effettivamente numeri casuali equamente distribuiti
- Generare 10000 numeri (**chiavi**) e inserirli in una **HashMap**
  - Se il numero generato già esiste, si incrementa un contatore (**valore**) associato al numero
  - Stampare la **HashMap**

SortedMap

- **SortedMap** (**TreeMap** è l'unica **SortedMap** disponibile) tiene le chiavi ordinate
- Questo permette l'aggiunta di alcune funzionalità (simili a quelle di **SortedSet**)
  - **Comparator comparator()**
    - Restituisce il **comparator** utilizzato per **Map**, o **null** per l'ordinamento naturale degli oggetti
  - **Object firstKey()**, **Object lastKey()**
  - **SortedMap subMap(fromKey,toKey)**
  - **SortedMap headMap(toKey)**
    - Tutti gli elementi con chiavi fino a **toKey**
  - **SortedMap tailMap(fromKey)**
    - Tutti gli elementi con chiave da **fromKey**

LinkedHashMap

- Utilizza hashing, e attraversando la struttura dati restituisce le coppie nell'ordine in cui sono state inserite (creando l'*impressione* di gestire una lista)
- Inoltre, nel costruttore si può specificare di utilizzare un algoritmo *least-recently used* (*LRU*) basato sugli accessi, che porta gli elementi non acceduti da più tempo in cima alla lista
  - Utile in situazioni dove si effettuano pulizie periodiche dei dati

Codici hash

- Come anticipato, ogni oggetto in Java ha associato un codice hash restituito dal metodo **hashCode()** in **Object**
- Per default questo metodo restituisce semplicemente l'indirizzo dell'oggetto
  - Quindi, due istanze diverse della stessa classe non producono *mai* (con il metodo di default) lo stesso codice hash
  - Questo non va bene se vogliamo usare oggetti di quella classe come chiavi
    - Infatti, tipicamente un oggetto utilizzato come chiave viene confrontato con un altro oggetto (ad esempio nella **get()**)
    - Con il metodo di dafault questo confronto non produrrà mai *true*

Codici hash

- Per le classi standard tutto va bene, perché esse hanno **hashCode()** riscritto (override) in modo che gli oggetti creati si possano comportare correttamente come chiavi
- Per implementare correttamente hashCode() nelle classi create da noi bisogna
  - Riscrivere **hashCode()** in modo da fargli produrre un codice hash significativo
  - Riscrivere anche **equals()**
    - Per default, infatti, anche **equals()** confronta gli indirizzi degli oggetti e non il loro contenuto
    - **HashMap** utilizza **equals()** per determinare se una chiave passata è uguale a un'altra presente nella tabella

Scegliere una implementazione

- Riassumendo, abbiamo visto che essenzialmente esistono solo tre contenitori in Java
  - **Map**, **List**, **Set**
- Esistono però diverse implementazioni per ognuno di essi
- Coma fare allora a decidere quale usare?
- Per poterlo fare bisogna conoscere i punti di forza e le debolezze di ognuna di esse

Scegliere tra le **List**e

- Il modo migliore per esaminare le differenze fra le varie implementazioni delle **List**e è tramite un test
- Si confrontano tra loro **ArrayList**, **LinkedList** e **Vector** rispetto a estrazione, iterazione, inserimento e cancellazione di elementi, e si misurano i tempi in millisecondi (**System.currentTimeMillis()**)

Type	Get	Iteration	Insert	Remove
array	172	516	na	na
ArrayList	281	1375	328	30484
LinkedList	5828	1047	109	16
Vector	422	1890	360	30781

Scegliere tra **Set**

- In questo caso è possibile scegliere tra **TreeSet**, **HashSet** e **LinkedHashSet**

Type	Test size	Add	Contains	Iteration
TreeSet	10	25.0	23.4	39.1
	100	17.2	27.5	45.9
	1000	26.0	30.2	9.0
HashSet	10	18.7	17.2	64.1
	100	17.2	19.1	65.2
	1000	8.8	16.6	12.8
LinkedHashSet	10	20.3	18.7	64.1
	100	18.6	19.5	49.2
	1000	10.0	16.3	10.0

Scegliere tra **Map**

Type	Test size	Put	Get	Iteration
TreeMap	10	26.6	20.3	43.7
	100	34.1	27.2	45.8
	1000	27.8	29.3	8.8
HashMap	10	21.9	18.8	60.9
	100	21.9	18.6	63.3
	1000	11.5	18.8	12.3
LinkedHashMap	10	23.4	18.8	59.4
	100	24.2	19.5	47.8
	1000	12.3	19.0	9.2
	10	20.3	25.0	71.9
Type	**Test size**	**Put**	**Get**	**Iteration**
IdentityHashMap	100	19.7	25.9	56.7
	1000	13.1	24.3	10.9
WeakHashMap	10	26.6	18.8	76.5
	100	26.1	21.6	64.4
	1000	14.7	19.2	12.4
Hashtable	10	18.8	18.7	65.7
	100	19.4	20.9	55.3
	1000	13.1	19.9	10.8

- La dimensione della **Map** è il fattore che influenza maggiormente le prestazioni

Ordinamento e ricerca in **Liste**

- I metodi per ordinare e cercare nelle **List**e hanno gli stessi nomi di quelli utilizzati per gli array, ma sono metodi statici di **Collections** invece che di **Arrays**

Utilità in **Collections**

max(Collection) **min(Collection)**	Produces the maximum or minimum element in the argument using the natural comparison method of the objects in the **Collection**.
max(Collection, Comparator) **min(Collection, Comparator)**	Produces the maximum or minimum element in the **Collection** using the **Comparator**.
indexOfSubList(List source, List target)	Produces starting index of the *first* place where **target** appears inside **source**.
lastIndexOfSubList(List source, List target)	Produces starting index of the *last* place where **target** appears inside **source**.
replaceAll(List list, Object oldVal, Object newVal)	Replace all **oldVal** with **newVal**.
reverse( )	Reverses all the elements in place.
rotate(List list, int distance)	Moves all elements forward by **distance**, taking the ones off the end and placing them at the beginning.
copy(List dest, List src)	Copies elements from **src** to **dest**.
swap(List list, int i, int j)	Swaps elements at locations **i** and **j** in **list**. Probably faster than what you'd write by hand.

fill(List list, Object o)	Replaces all the elements of list with **o**.
nCopies(int n, Object o)	Returns an immutable **List** of size n whose references all point to **o**.
enumeration(Collection)	Produces an old-style **Enumeration** for the argument.
list(Enumeration e)	Returns an **ArrayList** generated using the **Enumeration**. For converting from legacy code.

Collection e **Map** a sola-lettura

- Se serve creare versioni a sola-lettura di una Collection o una Map, ci viene in aiuto la classe Collections
- Il metodo **static**o **unmodifiableXXX()** prende un contenitore e ne ritorna uno a sola-lettura
- Esistono 4 versioni di questo metodo: per **Collection**, **List**, **Set** e **Map**

Da array a **Liste**

- Il metodo **static**o **Arrays.asList()** trasforma l'array passato come argomento in una lista che implementa alcuni dei metodi previsti in **List**, ma non tutti
- Infatti, la lista restituita è di dimensione *fissata* (si basa sull'array passato come argomento), quindi non ha senso, ad esempio, implementare **add()**
- Invocando metodi che non sono implementati, il compilatore non da errore, ma otteniamo a run-time **UnsupportedOperationException**

Contenitori presenti in Java 1

- Alcuni contenitori sono stati introdotti in *Java1* e per compatibilità mantenuti anche in *Java2*
- **Vector**: è come **ArrayList**
- **Enumerator**: corrisponde a **Iterator**
- **Hashtable**: simile (anche in performance) a **HashMap**
- **Stack** (ereditata da **Vector**)
  - Ha metodi **push**, **pop**, e **put**
  - Come visto, può essere implementata con una **LinkedList**

Contenitori presenti in Java 1

- **BitSet**: utilizzato per memorizzare efficientemente informazioni binarie
  - È efficiente solo per la dimensione
  - Negli accessi è leggermente più lento degli array
  - Inoltre, la dimensione minima del **BitSet** è **long** (64 bits). Memorizzare dati più piccoli produce spreco di memoria

Sommario

- Un array associa indici numerici agli oggetti, che devono essere tutti dello stesso tipo. Può essere multidimensionale
- Una **Collection** memorizza singoli elementi, mentre **Map** coppie
- Come una array, **List** associa indici numerici agli oggetti. **List** modifica la dimensione delle liste automaticamente. **List** può memorizzare solo riferimenti a **Object** (non possono tenere tipi primitivi direttamente)

Sommario

- Utilizzare un **ArrayList** se si fanno molti accessi diretti, e una **LinkedList** se si fanno molti inserimenti e rimozioni dal centro della lista
- Code e pile possono essere realizzate tramite **LinkedList**
- Una **Map** serve per associare oggetti ad altri oggetti
- **HashMap** mira a velocizzare l'accesso, mentre **TreeMap** tiene le chiavi ordinate (e quindi non è così veloce come **HashMap**). **LinkedHashMap** tiene gli elementi in ordine di inserimento, ma li può riordinare con un algoritmo LRU

Sommario

- Un **Set** evita le duplicazioni
- **HashSet** massimizza la velocità di ricerca, mentre **TreeSet** tiene gli elementi ordinati. **LinkedHashSet** tiene gli elementi in ordine di inserimento
- Non è necessario utilizzare **Vector**, **Hashtable** e **Stack** nel codice *Java2*

Esercizi

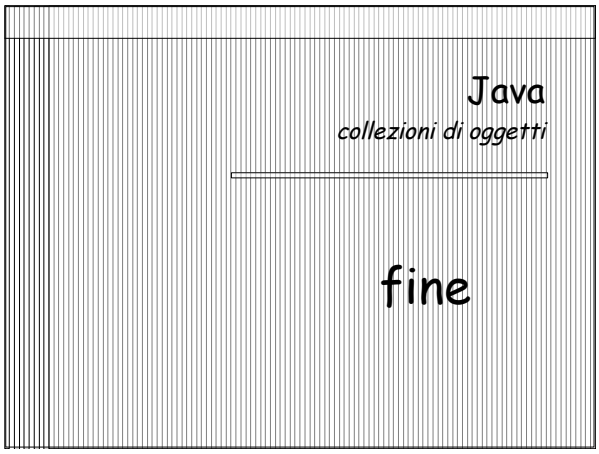
1. Creare una classe **Coniglio** con un **int numeroConiglio** che viene inizializzato nel costruttore
   1. Scrivere un metodo **salta()** che stampa il numero e la frase "salta!!"
   2. Creare una classe con un metodo **genConiglio()** che genera casualmente un **Coniglio**
   3. Utilizzando **genConiglio()**, creare una **ArrayList** con degli oggetti **Coniglio**
   4. Utilizzare **get()** per scorrere la lista e invocare **salta()** per ogni **Coniglio**
   5. Modificare il punto precedente utilizzando un **Iterator**
   6. Definire un **Comparator** per **Coniglio**, ordinare l'**ArrayList** e stampare la nuova lista così ottenuta

Esercizi

2. Inserire oggetti **Coniglio** in una **Map**, associando un nome (**chiave**) ad ogni **Coniglio** (**valore**)
   1. Ottenere un **Iterator** per **keySet()** e utilizzarlo per scorrere i conigli nella **Map**, stampare il loro nome e invocare **salta()**
   2. Stampare anche l'**hashCode()** di ogni **Coniglio**

Esercizi

3. Creare una classe con due **String**he, e renderla **Comparable** in modo che i confronti avvengano solo con la prima delle due **String**he
   1. Creare una classe con un metodo **genString()** che generi **String**he casuali
   2. Utilizzando **genString()**, riempire un array con oggetti di questa classe
   3. Ordinare la lista e stamparla
   4. Definire un **Comparator** che utilizza solo la seconda **String**a
   5. Ordinare l'array e eseguire una ricerca binaria utilizzando il **Comparator**


Java
collezioni di oggetti
fine