STRUTTURE DINAMICHE

DIFFERENZE RISPETTO AD ARRAY

- Finora le collezioni di dati sono state dichiarate come array (di stringhe, di interi, ecc.) la cui dimensione devono essere specificate con l'istruzione new e non possono cambiare
- Nelle strutture dinamiche, le dimensioni delle collezioni possono cambiare nel tempo
- Ad esempio, nella progettazione di una struttura dati atta a rappresentare una melodia non si può sapere a priori da quante note essa sarà composta

PREMESSA: GENERICS

- Spesso occorre definire nella dichiarazione di classi, metodi, ecc. dei comportamenti che sono indipendenti dal tipo di dato con cui si sta lavorando
- Ad esempio:
  - si vuole implementare una classe Pair, che contenga una coppia di valori dello stesso tipo (ma di qualunque tipo)
  - una soluzione percorribile è – per mantenere generico il tipo di dato da trattare – dichiarare i due valori come Object:

```
class Pair {
	public Object first;
	public Object second;
	public Pair(Object a, Object b) {
		first = a;
		second = b;
	}
}
```

- Con l'implementazione precedente, supponiamo di voler usare la classe in due porzioni di codice, con tipi interi e con stringhe:

```
Pair a = new Pair(1,2);
System.out.println(a.first + a.second);
// Errore in compilazione: sono Object!
System.out.println((int)a.first + (int)a.second);
// Funziona, ma definisco un'altra coppia con:
Pair b = new Pair("a","b");
System.out.println ((int)b.first + (int)b.second); //???
```

- La compilazione non genera errori, ma a run-time viene sollevata un'eccezione non gestita

GENERICS

- Con l'uso dei generics è possibile specificare dei tipi come parametri quando si dichiarano classi, interfacce e metodi
- La classe precedente può essere ridefinita come:

```
class PairGen<T1,T2> {
	public T1 first;
	public T2 second;
	public PairGen(T1 a, T2 b) {
		first = a;
		second = b;
	}
}
```

- All'interno delle parentesi angolari vengono dichiarati i tipi T1 e T2 come parametri della classe

GENERICS: ESEMPIO

Si può scrivere:
PairGen<Integer, Integer>(1,2)
PairGen<>(1,2)
PairGen(1,2)

- Per usare la classe come prima, ora si scrive:

```
PairGen<Integer, Integer> a = new PairGen<>(1,2);
System.out.println(a.first + a.second); // OK: 3
PairGen<String, String> b = new PairGen("a","b");
System.out.println(b.first + b.second); // OK: ab
```

- Nella dichiarazione di tipo non possono essere usati tipi primitivi (int, float, char, ecc.)
- Pregi:
  - L'astrazione sul concetto di "coppia" di valori viene mantenuta
  - Non è necessario effettuare cast
  - Gli errori di utilizzo errato vengono generati in fase di compilazione (ad es.: sostituendo nelle stampe degli esempi precedenti i + con * l'ultima riga del codice sopra genera errore di compilazione)

COLLECTIONS

- Le interfacce Java dedicate a strutture dati dinamiche si dividono in 2 gruppi, basate rispettivamente sulle interfacce generiche java.util.Collection e java.util.Map

- java.util.Collection permette di lavorare con liste di singoli valori, es. List<Integer> lista;

- java.util.Map rappresenta invece coppie chiave/valore, es. Map<Integer, String> map;

- Le interfacce che vengono implementate (sotto-interfacce di java.util.Collection e java.util.Map) determinano una serie di classi specializzate con precise caratteristiche

INSIEMI E LISTE: CLASSI PRINCIPALI

- HashSet<E>
  - Insieme di elementi non ordinati (l'ordine di iterazione sugli elementi non è garantito)
  - Operazioni generiche (aggiunta, rimozione, controllo contenuto) avvengono in tempo costante
- TreeSet<E>
  - Insieme di elementi ordinati per valore
  - Operazioni generiche (aggiunta, rimozione, controllo contenuto) avvengono in tempo logaritmico
- ArrayList<E>
  - Insieme di elementi ordinati per ordine di inserimento

OPERAZIONI SU LISTE

Metodi comuni alle collezioni presentate:

- `public boolean add(E e)`
  aggiunge l'elemento e
- `public boolean remove(Object o)`
  rimuove l'elemento o, se esiste
- `public boolean contains(Object o)`
  controlla se l'elemento specificato è presente nella lista
- `public void clear()`
  cancella tutti gli elementi dalla lista
- `public int size()`
  restituisce il numero di elementi nella lista
- `public boolean isEmpty()`
  controlla se la lista è vuota

METODI AGGIUNTIVI "INTERESSANTI"

- TreeSet
  - `public E first()`
    restituisce il primo elemento (il più piccolo in lista)
  - `public E last()`
    restituisce l'ultimo elemento (il più grande in lista)

- ArrayList
  - `public boolean add(int index, E e)`
    inserisce l'elemento `e` nella posizione `index`
  - `public E set(int index, E e)`
    sostituisce l'elemento alla posizione `index`
  - `public E get(int index)`
    restituisce l'elemento alla posizione `index`
  - `public int indexOf(Object o)`
    restituisce la prima posizione in cui viene trovato l'elemento `o` (-1 se non presente)
  - `public E remove(int index)`
    rimuove l'elemento alla posizione `index`

ESERCIZIO
(TestMelody.java)

- Si implementi la classe Melody, che contiene un elenco dinamico di istanze delle classi Note e Rest (NoteRest)
- La classe dovrà mettere a disposizione metodi basilari di analisi della melodia, quali:
  - un contatore delle pause
  - un contatore delle note
  - due metodi che restituiscono il pitch più alto e quello più basso (se esistono…)
  - due metodi che restituiscono la durata più lunga e più breve per i simboli musicali

Osservazione: per la descrizione di note e pause è possibile ripartire dal file Note.java

Hint: per sapere se un oggetto è istanza di una certa classe si usa l'operatore instanceof:

```
boolean isClasse = (oggetto instanceof Classe);
```

DIZIONARI: CLASSI PRINCIPALI

- Le collezioni che implementano l'interfaccia java.util.Map sono i dizionari: non si accede al singolo elemento per posizione ma per chiave

- HashMap<K,V>
  - Insieme di chiavi non ordinate (l'ordine di iterazione sugli elementi non è garantito)
  - Operazioni generiche (aggiunta, rimozione, controllo contenuto) avvengono in tempo costante

- TreeMap<K,V>
  - Insieme di chiavi ordinate per valore
  - Operazioni generiche (aggiunta, rimozione, controllo contenuto) avvengono in tempo logaritmico

OPERAZIONI SU DIZIONARI

Metodi comuni alle collezioni presentate:

- `public V put(K k, V v)`
  associa alla chiave k l'elemento v
- `public V remove(Object k)`
  rimuove la mappatura della chiave k, se esiste
- `public boolean containsKey(Object k)`
  controlla se esiste una mappatura della chiave specificata
- `public boolean containsValue(Object v)`
  controlla se esiste almeno una chiave che mappi sull'elemento specificato

OPERAZIONI SU DIZIONARI

Metodi comuni alle collezioni presentate:

- `public Set<Map.Entry<K,V>> entrySet()`
  restituisce l'insieme delle mappature del dizionario
- `public Set<K> keySet()`
  restituisce l'insieme delle chiavi
- `public Collection<V> values()`
  restituisce una collezione dei valori
- `public void clear()`
  cancella tutti gli elementi dalla lista
- `public int size()`
  restituisce il numero di elementi nella lista
- `public boolean isEmpty()`
  controlla se la lista è vuota

ALCUNI METODI AGGIUNTIVI SPECIFICI

Metodi propri del dizionario TreeMap:

- `public Map.Entry<K,V> firstEntry()`
  restituisce la coppia chiave/valore associata alla chiave più piccola

- `public K firstKey()`
  restituisce la chiave più piccola

- `public Map.Entry<K,V> lastEntry()`
  restituisce la coppia chiave/valore associata alla chiave più grande

- `public K lastKey()`
  restituisce la chiave più grande

CICLARE SUGLI ELEMENTI DI COLLEZIONI

- Liste:
  - `for (E element : list)`
    (element è l'elemento corrente)
  - `for (int i = 0; i < list.size(); i++)`
    list.get(i) è l'elemento corrente (ArrayList)

- Dizionari
  - `for (Map.Entry<K, V> entry: map.entrySet())`
    entry è la coppia chiave/valore corrente
  - `for (K k : map.keySet())`
    k è la chiave corrente, map.get(k) è il valore corrente
  - `for (V v : map.values())`
    v è il valore corrente

STRUTTURE ANNIDATE

- All'interno di liste e dizionari i tipi di dato possono essere complessi a piacere, ad esempio:

  ```
  TreeMap<Integer, HashSet<ArrayList<Note>>> chords;
  ```

  potrebbe costituire in un brano un dizionario (TreeMap) che mappa rispetto al numero di note (Integer) contenute negli accordi un certo numero di accordi (HashSet) costituiti da una lista (ArrayList) di note (Note)

- `chords.get(4)` restituisce la lista di accordi costituiti da 4 note
- `for (ArrayList<Note> chord : chords.get(4))` cicla sugli accordi di 4 note
- All'interno del ciclo precedente `chord.get(0)` restituisce la prima nota del singolo accordo
- (Attenzione: negli esempi non sono state effettuati controlli, ma chords.get(4), chord, chord.get(0) potrebbero essere null)

ESERCIZIO
(TestMelodyChord.java)

- Prendendo spunto dalla precedente classe Melody, si aggiunga la classe MelodyChord, che contiene un elenco dinamico di istanze delle classi Note (ArrayList<ArrayList<Note>>), rappresentante una sequenza di accordi
- La classe dovrà mettere a disposizione metodi basilari di analisi della melodia, quali:
  - un contatore del numero totale delle note
  - due metodi che restituiscono il pitch più alto e quello più basso (se esistono…)
  - due metodi che restituiscono la durata più lunga e più breve delle note
  - un metodo che restituisca la struttura TreeMap<Integer, HashSet<ArrayList<Note>>> vista nella slide precedente, in cui gli accordi sono ordinati per numero di note che li costituiscono