Programmazione Orientata agli Oggetti

Collezioni:
Mappe Generiche

Sommario

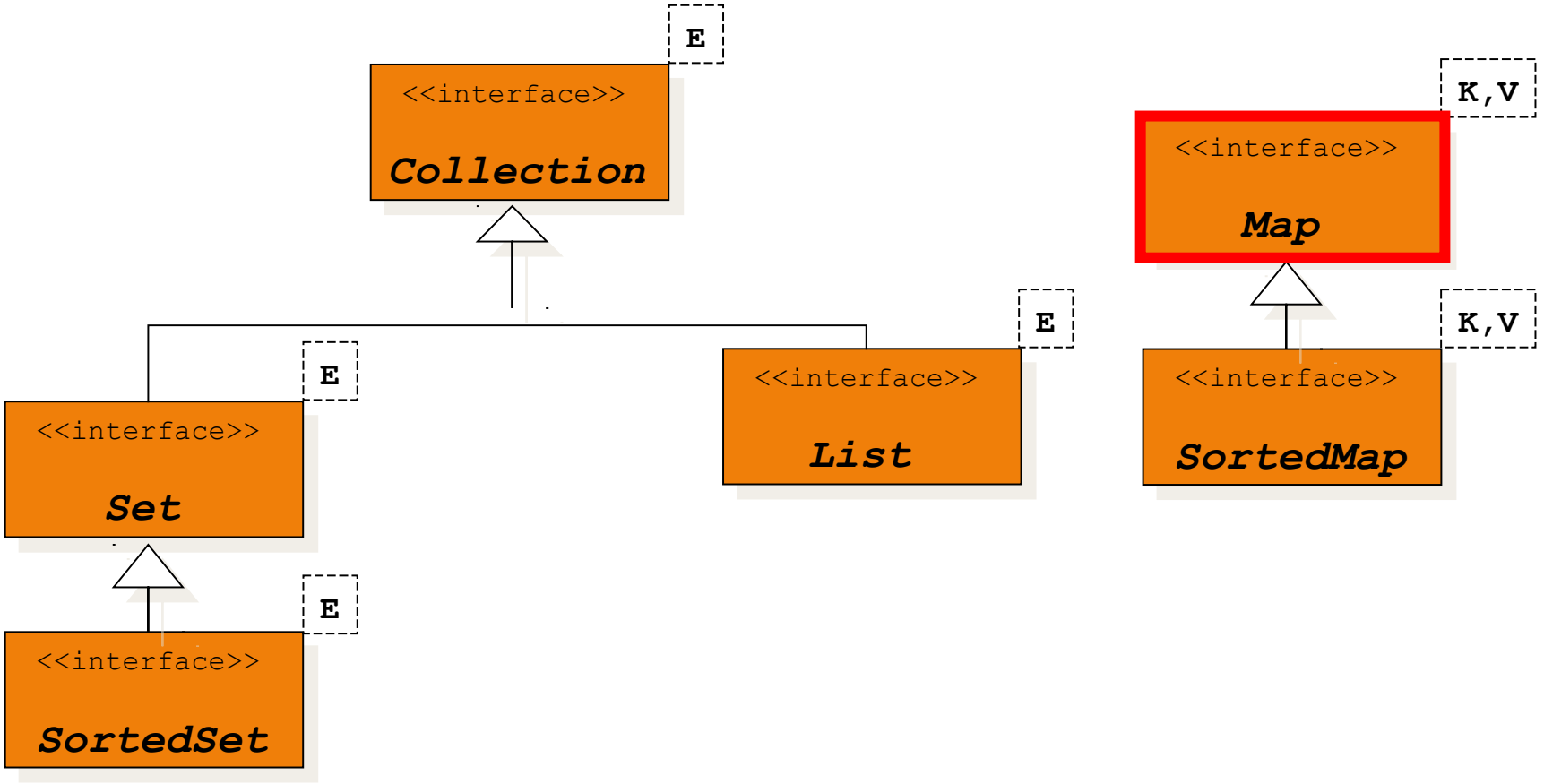
- Interface **`Map<K,V>`**
  - Operazioni tipiche sulle mappe
- Implementazioni di **`Map<K,V>`**
  - unicità delle chiavi

Sommario

- Interface **`Map<K,V>`**
  - Operazioni tipiche sulle mappe
- Implementazioni di **`Map<K,V>`**
  - unicità delle chiavi

Collezioni: Interface

- Le principali interface del package **`java.util`**



- Per ognuna di queste interface il package offre diverse implementazioni

Mappe

- Una mappa (o dizionario, o array associativo) è una collezione di coppie chiave-valore (entry)
- Le chiavi, in quanto tali, sono uniche
  - In una mappa non possono esistere due chiavi eguali
- Ad ogni chiave può essere associato un solo valore
  - ✓ ma il valore può essere una collezione

- Esempi:
  - rubrica telefonica: a ciascuna persona (chiave) è associato un numero di telefono (il numero è il valore)
  - indice analitico di un libro: ad ogni voce (chiave) è associato un insieme di numeri di pagine (l'insieme è il valore)

Mappe: Operazioni Principali

- Operazioni tipiche su una mappa sono:
  - Inserimento di una coppia chiave-valore
    - Inserisci nella rubrica: "Paolo Rossi" -> 0288821212
  - Richiesta del valore associato ad una chiave
    - Dammi il numero di telefono di "Elena Bruni"
  - Aggiornamento di un valore
    - Cambia il numero di telefono di "Paolo Rossi" in 063232122
  - Rimozione di una coppia chiave-valore
    - Rimuovi "Mario Verdi" dalla mia rubrica
  - Verifica di esistenza di una chiave
    - C'è "Chiara Scuri" nella mia rubrica?
  - Richiesta dell'insieme delle chiavi
    - Dammi i nomi di tutte le persone della mia rubrica

I Metodi Base di `Map<K,V>`

- **`V put(K chiave, V valore);`**
  inserisce la coppia chiave-valore nella mappa;
  - se la chiave esiste già, allora il valore viene aggiornato e il metodo ritorna il valore vecchio
  - se la chiave non esiste viene inserita una nuova coppia, e il metodo ritorna **`null`**
- **`V get(Object chiave);`**
  restituisce il valore associato alla chiave, **`null`** se la chiave non esiste nella mappa
- **`V remove(Object chiave);`**

  rimuove la coppia associata alla chiave e lo ritorna il valore
- **`boolean containsKey(Object chiave);`**

  verifica se la chiave è presente nella mappa

Metodi Bulk e Viste di `Map<K,V>`

- **`void putAll(Map<K,V> mappa)`**
  inserisce nella mappa tutte le coppie della mappa passata come parametro
- **`void clear()`**

  elimina tutte le coppie dalla mappa
- **`Set<K> keySet()`**
  restituisce in un insieme tutte le chiavi
- **`Collection<V> values()`**

  restituisce in una collezione tutti i valori

Sommario

- Interface **`Map<K,V>`**
  - Operazioni principali
- Implementazioni di **`Map<K,V>`**
  - unicità delle chiavi

Mappe ed Insiemi

- Nel Java Collection Framework **`Map<K,V>`** e **`Set<K>`** si basano su concetti molti simili
  - ✓ In realtà l'implementazione di **`HashSet<K>`** si basa su una istanza di **`HashMap<K,?>`**

- Abbiamo diverse implementazioni di **`Map<K,V>`**, e **`SortedMap<K,V>`**, rispettivamente:
  - **`HashMap<K,V>`**
  - **`TreeMap<K,V>`**

- I meccanismi per la definizione dei criteri di equivalenza sulle chiavi sono gli stessi già visti per **`Set<K>`**, **`SortedSet<K>`**

Mappe: Implementazioni

- Per definire i criteri di equivalenza esistono due meccanismi:
  - l'unicità delle chiavi in **`HashMap<K,V>`** è definita attraverso i metodi **`hashCode()`** e **`equals()`** della classe delle chiavi
  - l'unicità delle chiavi in **`TreeMap<K,V>`** è «indotta» dal criterio di ordinamento delle chiavi stesse:
    - Ordinamento «Naturale»: attraverso il metodo **`compareTo()`** delle chiavi (e le chiavi devono quindi implementare **`Comparable<K>`**), *oppure*
    - *Ordinamento* «Esterno»: tramite il metodo **`compare()`** di un comparatore esterno **`Comparator<K>`** (passato al costruttore della mappa)

Esercizio

- Scrivere una batteria di metodi di test per comprendere a fondo la semantica di tutti i metodi citati nelle slide precedenti con riferimento alle due implementazioni di

  `Map<K,V>`

  - **`HashMap<K,V>`**
  - **`TreeMap<K,V>`**

Esempio: Rubrica

```
import java.util.*;
public class Rubrica {
    private Map<String,Integer> rubrica;

    public Rubrica() {
        this.rubrica = new HashMap<String,Integer>();
//      this.rubrica = new HashMap<>(); // solo da Java7…
    }

    public void inserisci(String nome, Integer numero) {
        this.rubrica.put(nome, numero);
    }

    public void rimuovi(String nome) {
        this.rubrica.remove(nome);
    }

    public Set<String> nomiInRubrica() {
        return this.rubrica.keySet();
    }

    public Integer dammiIlNumeroDi(String nome) {
        return this.rubrica.get(nome);
    }
…
```

Esempio: Rubrica (cont.)

```
...
    public Integer dammiIlNumeroDi(String nome) {
        return this.rubrica.get(nome);
    }

    public Integer aggiornaNumero(String nome, Integer numero) {
        return this.rubrica.put(nome, numero);
    }

    @Override
    public String toString(){
        StringBuilder str = new StringBuilder();
        str.append("----------------------------\n");
        str.append("Rubrica\n");
        Set<String> nomi = this.rubrica.keySet();
        for(String s : nomi) {
           str.append(s);
           str.append(": ");
           str.append(this.rubrica.get(s));
           str.append("\n");
        }
        str.append("----------------------------\n");
        return str.toString();
    }
...
```

Esempio: Rubrica (cont.)

```
…  public static void main(String[] args) {
        Rubrica r = new Rubrica();
        String s1 = new String("Paolo"), s2 = new String("Fabio");
        String s3 = new String("Anna"),  s4 = new String("Carla");

        r.inserisci(s1, 390112461); // inserisco Paolo->390112461 in rubrica
        r.inserisci(s2, 390108361); // inserisco Fabio->390108361 in rubrica
        r.inserisci(s3, 39062888);  // inserisco Anna->39062888 in rubrica
        System.out.println(r.toString()); // stampo la rubrica
        // verifico se ho il numero di Carla
        Integer numeroCercato = r.dammiIlNumeroDi(s4);
        if (numeroCercato == null)
                System.out.println("Il numero di "+s4+" non esiste");
        else
                System.out.println("Il numero di "+s4+" è "+numeroCercato);

        r.rimuovi(s2);  // tolgo i dati relativi a Fabio dalla rubrica

        // cambio il numero di Paolo (e mi faccio stampare il numero vecchio)
        Integer nuovoNumero = 39066777;
        Integer vecchioNumero = r.aggiornaNumero(s1,nuovoNumero);
        System.out.println("Aggiornato il numero di "+s1+
                           " da "+vecchioNumero+" a "+nuovoNumero);

        System.out.println(r.toString()); // stampo la rubrica
    }
}
```

Esercitazione: Caso di Studio diadia

- Possiamo usare le mappe per semplificare il codice del nostro studio di caso
- Possiamo gestire in maniera molto più semplice l'implementazione della classe **`Stanza`**, in particolare per ciò che riguarda l'associazione

  `direzione` → `stanzaAdiacente`

  - attualmente è implementata tramite due array

Esercitazione: Testing & Refactoring

1. Rieseguire i test già accumulati prima di ogni refactoring
2. Cambiare la rappresentazione interna di **`Stanza`**
3. Quindi si rieseguano gli stessi test e si verifichi che tutto continua a funzionare nonostante le modifiche
   - In presenza di fallimenti, utilizzare i test per localizzare e rimuovere velocemente, ed a basso costo, i bug introdotti

Riscriviamo il Codice di Stanza

```
public class Stanza {
    private String descrizione;
    private List<Attrezzo> attrezzi;
    private Map<String,Stanza> stanzeAdiacenti;

    public Stanza(String descrizione){
        this.uscite = new HashMap<>();
        this.attrezzi = new ArrayList<>();
        this.descrizione = descrizione;
    }

    public Stanza getStanzaAdiacente(String direzione) {
       return this.stanzeAdiacenti.get(direzione);
    }

    void impostaStanzaAdiacente(String direzione,
                                Stanza stanzaAdiacente) {
       this.stanzeAdiacenti.put(direzione,stanzaAdiacente);
    }
    …
```

Esercizio Mappe

- Date le classi **`Studente`** e **`Aula`**

```
public class Studente {
    private String matricola;
    private String meseDiNascita;
    public Studente(String matricola, String meseDiNascita) {
       this.matricola = matricola;
       this.meseDiNascita = meseDiNascita;
    }

    public String getMatricola()    { return this.matricola;              }

    public String getMeseDiNascita() { return this.meseDiNascita;      }
    @Override
    public int hashCode()           { return this.matricola.hashCode();}
    @Override
    public boolean equals(Object o) {
       Studente that = (Studente)o;
       return this.getMatricola().equals(that.getMatricola());
    }
}
```

Esercizio Mappe (cont.)

```
public class Aula {
    private Set<Studente> studenti;

    public Aula() {
        this.studenti = new HashSet<Studente>();
    }

    public boolean addStudente(Studente studente) {
        studenti.add(studente);
    }

    public Map<String, List<Studente>> meseDiNascita2studenti() {
        // scrivere il codice di questo metodo
    }
}
```

- Scrivere il codice del metodo **`meseDiNascita2studenti()`**:
  - ritorna una mappa che associa a ciascun mese una lista con tutti e soli gli studenti nati in quel mese

Esercizio Mappe (cont.)

- Esempio:
  - Usiamo <"00","gen"> per denotare un oggetto **`Studente`** di matricola "00" e mese di nascita gennaio
  - se un'aula contiene un insieme con i seguenti studenti:

    {<"00","gen">, <"11","gen">, <"54","ott">,<"24","mar">,<"32", "mar">}

  allora il metodo **`meseDiNascita2studenti()`** deve restituire una mappa con le seguenti coppie chiave → valore :

  "gen" → {<"00", "gen">,<"11", "gen">}

  "ott" → {<"54", "ott">}

  "mar" → {<"32", "mar">,<"24", "mar">}

Una Soluzione

```
public Map<String, List<Studente>> meseDiNascita2studenti() {
    List<Studente> tmp;
    Map<String, List<Studente>> mappa;
    mappa = new HashMap<String, List<Studente>>();

    for(Studente studente : this.studenti){
        tmp = mappa.get(studente.getMeseDiNascita());
        if (tmp==null)
            tmp = new ArrayList<Studente>();
        tmp.add(studente);
        mappa.put(studente.getMeseDiNascita(), tmp);
    }
    return mappa;
}
```

Un'Altra Soluzione

```
public Map<String, List<Studente>> meseDiNascita2studenti() {
    List<Studente> tmp;
    Map<String, List<Studente>> mappa;
    mappa = new HashMap<String, List<Studente>>();
    for(Studente studente : this.studenti) {
        tmp = mappa.get(studente.getMeseDiNascita());
        if (tmp==null) {
            tmp = new ArrayList<Studente>();
            mappa.put(studente.getMeseDiNascita(),tmp);
        }
        tmp.add(studente);
    }
   return mappa;
}
```

Un'Altra Soluzione Ancora!

```
public Map<String, List<Studente>> meseDiNascita2studenti() {
    List<Studente> tmp;
    Map<String, List<Studente>> mappa;
    mappa = new HashMap<String, List<Studente>>();
    for(Studente studente : this.studenti){
      if (mappa.containsKey(studente.getMeseDiNascita())) {
        tmp = mappa.get(studente.getMeseDiNascita());
         tmp.add(studente);
      }
      else {
         tmp = new ArrayList<Studente>();
         tmp.add(studente);
         mappa.put(studente.getMeseDiNascita(), tmp);
      }
    }
    return mappa;
}
```

Mappe, Liste, Insiemi

- Le mappe risultano spesso più comode di liste ed insiemi
  - ✓ N.B. non si può accedere all'elemento di un insieme se non disponendo di un suo duplicato!
- Riconsideriamo la classe **`Borsa`**: e la rappresentazione della collezione degli attrezzi nella borsa
- Confrontiamo le due alternative rappresentazioni:
  - Lista: **`List<Attrezzo>`**, VS
  - Mappa: **`Map<String, Attrezzo>`**,

    `nomeAttrezzo` → **`Attrezzo`**
- N.B.: non è possibile memorizzare oggetti **`Attrezzo`** distinti ma con lo stesso nome
- L'accesso per nome risulta agevole

Esercizio: Mappe & Boxing/Unboxing (1)

```
public Map<String, Integer> countWordOccurrences(String text){

    Map<String, Integer> word2count = new HashMap<>();

    Scanner scanner = new Scanner(text);

    while (scanner.hasNext()) {
        String word = scanner.next();
        int count = word2count.get(word);
        if (count==0) {
            count = 1;
        }
        else {
            count = count + 1;
        }
        word2count.put(word, count);
    }
    return word2count;

}
```

✔ Dov'è l'errore ?

Esercizio: Mappe & Boxing/Unboxing (2)

- ✓ Suggerimento: il codice precedente, una volta compilato, è equivalente al seguente

```
public Map<String, Integer> countWordOccurrences(String text){
    Map<String, Integer> word2count = new HashMap<>();
        Scanner scanner = new Scanner(text);
    while (scanner.hasNext()) {
        String word = scanner.next();
        int count = word2count.get(word).intValue();
        if (count==0) {
            count = 1;
        }
        else {
            count = count + 1;
        }
        word2count.put(word, new Integer(count));
    }
    return word2count;
}
```

NavigableMap<K,V>: da Java 6

- In Java 6 le funzionalità offerte da **SortedMap<K,V>** sono state rinconsiderate ed estese
- E' stata introdotta una specializzazione di **SortedMap<K,V>** denominata **NavigableMap<K,V>**
  - ✓ Implementata anche da **TreeMap<K,V>**
- Alcuni nuovi metodi risultano molto utili, ad es.:

```
NavigableMap<K,V> subMap(K fromKey,
                         boolean fromInclusive,
                         K toKey,
                         boolean toInclusive)
```

- Valgono tutte le stesse considerazioni già fatte per **NavigableSet<E>** (>>)

NavigableMap<K,V> e Retrocompatibilità

- Alcune scelte sembrano discutibili. Sono state dettate dall'esigenza di mantenere la retro-compatibilità con le implementazioni e le interfacce già esistenti e diffuse

```
@Override
SortedMap<K> headMap(K toKey)
```

  - *Returns a view of the portion of this map whose keys are strictly less than **`toKey`**.*

```
NavigableMap<K> headMap(K toKey,
                        boolean inclusive)
```

  - *Methods **`subMap(K, K)`**, **`headMap(K)`**, and **`tailMap(K)`** are specified to return **`SortedMap`** to allow existing implementations of **`SortedMap`** to be compatibly retrofitted to implement **`NavigableMap`**, but extensions and implementations of this interface are encouraged to override these methods to return **`NavigableMap`**. Similarly, **`SortedMap.keySet()`** can be overriden to return **`NavigableSet`**.*

Sovrautilizzo del JCF

- Un ultimo invito alla cautela nell'uso immotivamente complicato delle collezioni nel JCF
- Ad es., se vi ritrovate ad utilizzare mappe eccessivamente nidificate, come la seguente...

```
Map<String, Map<String, Set<Studente>>>
```

...avete di fronte un segnale forte (anzi fortissimo!) che la distribuzione delle responsabilità tra le classi del vostro progetto necessita di essere riconsiderata

- ✓ State "evitando" la modellazione del dominio di interesse con l'utilizzo di opportuni tipi!

Collezioni vs Distribuzione delle Responsabilità

- Redistribuendo le responsabilità del progetto (tipicamente aggiungendo anche nuove classi di dominio come quelle utili a modellare esplicitamente le associazioni tra oggetti, cfr. **`Rubrica`**)
  - i livelli di nidificazione delle **`Map`** si «abbattono»
  - alcuni **`Set/List/Map`** diventano variabili di istanza di classi di dominio e non tipi attuali di **`Map`** generiche eccessivamente nidificate e difficili da «visualizzare»
  - il testing diventa molto più agevole

Ridistribuzione delle Responsabilità vs Collezioni Nidificate

- ✓ Esistono pertanto sostanziali motivi per preferire ad una struttura dati del tipo...

```
Map<String, Map<String,Set<Studente>>>;
```

...una serie di classi di dominio come ad es.:

- Classe **`Ateneo`** [con variabile di istanza]:
  - `Map<String, Dipartimento> nome2dipartimento;`
- Classe **`Dipartimento`**
  - `Map<String, CorsoDiStudio> nome2cds;`
- Classe **`CorsoDiStudio`**
  - `Set<Studente> studentiIscritti;`

Modello di Dominio “Anemico”

- Spesso il «sintomo» precedente si affianca all’uso eccessivo di **`java.lang.String`** come strumento di modellazione universale

  http://c2.com/cgi/wiki?StringlyTyped

- Sebbene ogni concetto sia rappresentabile con collezioni di **`String`** opportune, questo non deve distrarci dai vantaggi che scaturiscono dalla scelta dei giusti tipi per la rappresentazione dei concetti di dominio (vedi APS >>)
  - ✓ rimandare eccessivamente un’adeguata modellazione di dominio fa solo “lievitare” il conto che prima o poi si sarà costretti a pagare
  - – ad es. il progetto passa ad altri che non riescono ad interpretare il codice: diventa più conveniente riscriverlo che comprenderlo