

HOMEWORK 3

Java Collection Framework

Esercizio 0

- Chi non avesse concluso la scrittura dei test, lo faccia in questo homework, prima di fare le modifiche al codice
- E' importante partire da una versione del codice funzionante, con un sufficiente numero di test per garantirlo

Esercizio 1

- Sostituire tutti gli array utilizzati nelle classi **Stanza** e **Borsa** con opportune collezioni (**List**, **Set**, **Map**)
 - ✓ Assumere che non possano esistere due oggetti **Attrezzo** con lo stesso nome in stanze dello stesso **Labirinto**
 - Eliminare il vincolo che al max 10 attrezzi possano essere collocati nella borsa (ma mantenere quello sul peso max)
 - Provare ad usare (in alternativa) **List** e **Map** per implementare la collezione di attrezzi nella borsa. Quale risulta più semplice?
- ✓ Queste modifiche alterano l'implementazione ma non la logica: dopo averle effettuate confermare con i test sviluppati nei precedenti homework la correttezza del codice *anche* dopo le nuove modifiche

Esercizio 2

- Rivisitare il codice delle nuove tipologie di stanze introdotte nel precedente homework:
 - La stanza *buia*: se nella stanza non è presente un attrezzo con un nome particolare (ad esempio "lanterna") il metodo `getDescrizione()` di una stanza buia ritorna la stringa "*qui c'è un buio pesto*"
 - La stanza *bloccata*: una delle direzioni della stanza non può essere seguita a meno che nella stanza corrente non sia presente un oggetto di un certo nome (ad es. «piedediporco»)
- Queste sottoclassi subiscono modifiche consequenziali ai cambiamenti effettuati nella implementazione della loro superclasse **Stanza**?
- Le modifiche che subiscono le due versioni di **Stanza** (*con* e *senza* campi `protected`) ipotizzate nel precedente homework, sono le stesse?

Esercizio 2 (cont.)

- Ampliare i test di tutte le classi nella gerarchia che ha radice in **Stanza**:
 - **Stanza**, **StanzaMagica**, **StanzaBuia** e **StanzaBloccata**
- Eliminare dal codice delle classi **Borsa**, **Stanza**, **StanzaMagica**, **StanzaBuia** e **StanzaBloccata** (estensioni di **Stanza**) **ogni** ciclo di ricerca da un collezione (ad es. di un attrezzo per nome, o di una stanza per direzione)
- ✓ affidarsi invece sempre e solo alle funzionalità offerte dai metodi già offerti nelle classi del JCF
 - ad es. quelli di ricerca in una collezione

TDD (Facoltativo)

- ✓ N.B. È perfettamente lecito e consigliabile fare l'esercizio 5 anche prima degli esercizi 3&4

Esercizio 3

- Aggiungere alla classe **Borsa** dei metodi di interrogazione del suo contenuto:
 - `List<Attrezzo> getContenutoOrdinatoPerPeso()` ;
restituisce la lista degli attrezzi nella borsa ordinati per peso e quindi, a parità di peso, per nome
 - `SortedSet<Attrezzo> getContenutoOrdinatoPerNome()` ;
restituisce l'insieme degli attrezzi nella borsa ordinati per nome
 - `Map<Integer, Set<Attrezzo>> getContenutoRaggruppatoPerPeso()`
restituisce una mappa che associa un intero (rappresentante un peso) con l'insieme (comunque *non* vuoto) degli attrezzi di tale peso: tutti gli attrezzi dell'insieme che figura come valore hanno lo stesso peso pari all'intero che figura come chiave
- Utilizzare questi metodi per migliorare la stampa del contenuto della **Borsa** (ad es. aggiungere e/o modificare un comando *guarda* per la stampa del suo contenuto>>)

Esercizio 3 (Notazione Es.)

- Si utilizzi *piombo:10* per indicare un riferimento ad un oggetto **Attrezzo** di nome “piombo” e di peso 10
- Per brevità scriviamo *piombo* al posto di *piombo:10* quando non è utile ripetere il dettaglio sul peso
- Si utilizzi quindi:
 - { *piombo, piuma, libro, ps* } per indicare un **Set** di attrezzi
 - [*piuma, libro, ps, piombo*] per indicare una **List** di attrezzi
 - (5, { *libro, ps* }) per indicare una coppia chiave/valore di una **Map<Integer, Set<Attrezzi>>**

Esercizio 3 (Esempio)

- Si consideri una **Borsa** contenente questo insieme di riferimenti ad oggetti **Attrezzo**:
{ piombo:10, ps:5, piuma:1, libro:5 }
- Allora i metodi di cui prima, invocati sullo questa **Borsa**:
 - `List<Attrezzo> getContenutoOrdinatoPerPeso () ;`
deve restituire: *[piuma, libro, ps, piombo]*
 - `SortedSet<Attrezzo> getContenutoOrdinatoPerNome () ;`
deve restituire: *{ libro, piombo, piuma, ps }*
 - `Map<Integer, Set<Attrezzo>> getContenutoRaggruppatoPerPeso ()`
deve restituire una **Map** contenente tutte e sole le seguenti coppie chiave/valore: *(1, { piuma }) ; (5, { libro, ps }) ; (10, { piombo })*

Esercizio 4

- Aggiungere alla classe **Borsa** un nuovo metodo
 - `SortedSet<Attrezzo> getSortedSetOrdinatoPerPeso () ;`
restituisce l'insieme gli attrezzi nella borsa ordinati per peso e quindi, a parità di peso, per nome
- ✓ Scrivere un test per verificare che due attrezzi di stesso peso ma nome diverso rimangano distinti nel risultato

Esercizio 5

- Utilizzando JUnit, scrivere una batteria di test-case *minimali* per verificare la correttezza delle soluzioni prodotte negli esercizi 3&4 precedente
 - *minimali*: ovvero facenti utilizzo delle collezioni più semplici possibile utili alla verifica (piccole e con **Attrezzi** di nomi/pesi in configurazioni *a loro volta minimali*)
 - ✓ N.B. È perfettamente lecito e consigliabile fare questo esercizio anche prima degli esercizi 3&4
- Solo dopo aver completato il precedente punto, valutare se ampliare i test-case di sopra con altri test-case non minimali, per migliorarne la copertura

Esercizio 6

- Supportare partite svolte in labirinti diversi
 - deve essere possibile aggiungere e conservare il riferimento all'oggetto **Labirinto** in cui si svolge una partita direttamente dentro un oggetto **Partita** con il costruttore **Partita(Labirinto)** *ed anche* con il metodo **Partita.setLabirinto(Labirinto)**
- Per facilitare la costruzione di questi oggetti **Labirinto**, si aggiunga la classe **LabirintoBuilder**
- Classe dedicata esclusivamente alla creazione di oggetti **Labirinto** utilizzando una tecnica (*method-chaining*) che faciliti la costruzione incrementale di un oggetto il cui stato è complesso
 - formato da molte informazioni, ovvero: stanze, adiacenze, stanza iniziale, stanza vincente, attrezzi, ecc. ecc.

Esercizio 6 (continua)

- Per chiarire cosa deve fare questa classe `LabirintoBuilder`, l'insieme dei metodi che offre, e la loro semantica, si consideri il seguente codice esemplificativo del suo utilizzo mostrato di seguito:

- `Labirinto monolocale = new LabirintoBuilder()`

```
.addStanzaIniziale("salotto") // aggiunge una stanza, che sarà anche iniziale  
.addStanzaVincente("salotto") // specifica quale stanza sarà vincente  
.getLabirinto();
```

`Labirinto bilocale = new LabirintoBuilder()`

```
.addStanzaIniziale("salotto")  
.addStanzaVincente("camera")  
.addAttrezzo("letto",10) // dove? fa riferimento all'ultima stanza aggiunta  
.addAdiacenza("salotto", "camera", "nord") // camera si trova a nord di salotto  
.getLabirinto();
```

`Labirinto trilocale = new LabirintoBuilder()`

```
.addStanzaIniziale("salotto")  
.addStanza("cucina")  
.addAttrezzo("pentola",1) // dove? fa riferimento all'ultima stanza aggiunta  
.addStanzaVincente("camera")  
.addAdiacenza("salotto", "cucina", "nord")  
.addAdiacenza("cucina", "camera", "est")  
.getLabirinto(); // restituisce il Labirinto così creato
```

Esercizio 6 (continua)

- Implementare `LabirintoBuilder`
- Scrivere dei test sulla classe per individuare e correggere gli errori di `LabirintoBuilder`
- Completare le funzionalità rispetto alle sole esemplificate per gestire
 - i diversi tipi di stanza
 - ecc. ecc.

Esercizio 7

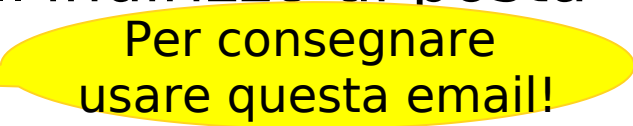
- Una volta appurato il corretto funzionamento di **LabyrinthBuilder** (esercizio 6) rivedere ed ampliare i test già scritti su altre classi
 - ad esempio **ComandoVai**
- Sfruttare la nuova classe per creare test-case con fixture di complessità crescenti: labirinto «monolocale», «bilocale», ecc.

Controlli Prima della Consegna

- Cambiare il codice di `DiaDia` affinché supporti la creazione di una `Partita` da svolgersi in una certo `Labirinto` fornito tramite un suo costruttore che permetta di specificarlo
 - aggiungere il costruttore: `DiaDia(Labirinto,io)`
- Ad esempio:

```
public class DiaDia {  
    ...  
    public static void main(String[] argc) {  
        /* N.B. unica istanza di IOConsole  
           di cui sia ammessa la creazione */  
        IO io = new IOConsole();  
        Labirinto labirinto = new LabirintoBuilder()  
            .addStanzaIniziale("LabCampusOne")  
            .addStanzaVincente("Biblioteca")  
            .addAdiacenza("LabCampusOne", "Biblioteca", "ovest")  
            .getLabirinto();  
  
        DiaDia gioco = new DiaDia(labirinto, io);  
        gioco.gioca();  
    }  
    ...}
```


TERMINI E MODALITA' DI CONSEGNA

- La soluzione deve essere inviata al docente entro le 21:00 del 24 maggio 2020 come segue:
 - Svolgere in gruppi di max 2 persone
 - Esportare (con la funzione File->Export di Eclipse) il progetto realizzato nel file **homework3.zip**
 - Inviare il file **homework3.zip** all'indirizzo di posta elettronica poo.roma3@gmail.com  Per consegnare usare questa email!
 - Nel corpo del messaggio riportare eventuali malfunzionamenti noti, ma non risolti
 - L'oggetto (subject) *DEVE* iniziare con la stringa **[2020-HOMEWORK3]** seguita dalle matricole
 - Ad es.: **[2020-HOMEWORK3] 412345 454321**