6. Laboratorio 6

Esercizio 1

Argomento: array riempiti solo in parte, semplici algoritmi su array

Aggiungere alla classe ArrayUtil gli algoritmi su array visti a lezione utilizzando array riempiti solo in parte.

- generazione di un array di interi di numeri casuali (randomIntArray);

  public static int[] randomIntArray(int length, int n)

- stampare il contenuto di un array (printArray);

  public static String printArray(int[] v, int vSize)

- eliminare un elemento (ordine non importante);

  public static void remove(int[] v, int vSize, int index)

- eliminare un elemento (ordine importante)

  public static void removeSorted(int[] v, int vSize, int index)

- inserire un elemento

  public static void insert(int[] v, int vSize, int index, int value)

- ricerca del minimo

  public static int findMin(int[] v, int vSize)

- ricerca del massimo

  public static int findMin(int[] v, int vSize)

- ricerca di un valore in modo sequenziale

  public static int find(int[] v, int vSize, int target)

Testare gli algoritmi scrivendo una classe di collaudo **ArrayUtilTester** che

- riceva da standard input la dimensione e l'intervallo di variabilità di un array di numeri interi casuali
- Stampi il contenuto dell'array generato
- Collaudi gli algoritmi visualizzando dopo ogni operazione il contenuto dell'array attraverso la sequenza di richieste:

1. Chiedere all'utente un valore e una posizione in cui inserirlo
2. Chiedere all'utente di indicare la posizione di un elemento da eliminare (senza considerare rilevante l'ordine)
3. Chiedere all'utente di indicare la posizione di un elemento da eliminare (considerando rilevante l'ordine)
4. Chiedere all'utente un valore da cercare
5. Visualizzi minimo e massimo del contenuto dell'array

Ricordarsi di modificare la taglia di vSize se le operazioni che la modificano vanno a buon fine.

Esercizio 2

Argomento: gestione dinamica di un array unidimensionale, cicli, array bidimensionali

Scrivere il programma IsMagicSquare che verifichi se un quadrato di numeri è un "quadrato magico".

Una disposizione bidimensionale di numeri con dimensione n x n è un quadrato magico se contiene i numeri interi da 1 a n^2 e se la somma degli

elementi di ogni riga, di ogni colonna e delle due diagonali principali ha lo stesso valore. Esempi di quadrati magici, con n=4 e n=5:

```
16  3  2 13
 5 10 11  8
 9  6  7 12
 4 15 14  1
```

```
11 18 25  2  9
10 12 19 21  3
 4  6 13 20 22
23  5  7 14 16
17 24  1  8 15
```

L'utente introduce i numeri del (presunto) quadrato in sequenza, uno o più numeri per riga, senza alcuna relazione con il numero di righe del quadrato: quando l'utente ha introdotto tutti i numeri e ha chiuso lo standard input, il programma deve intraprendere le azioni seguenti:

- verifica che il numero di valori introdotti sia il quadrato di un numero intero n: in caso contrario, il programma termina segnalando un fallimento;
- verifica che la sequenza di valori introdotta contenga tutti (e soli) i numeri da 1 a n^2, senza ripetizioni: in caso contrario, il programma termina segnalando un fallimento (pensate bene a come implementare questo controllo, si puo' fare, anzi si deve fare, solo con quanto visto finora);
- dispone i valori all'interno di un array bidimensionale, riempito per righe seguendo l'ordine con cui sono stati forniti i valori: il primo valore prende posto nell'angolo in alto a sinistra, il secondo nella seconda posizione da sinistra della prima riga e così via, riempiendo righe successive del quadrato;
- verifica la validità delle regole del quadrato magico, interrompendo la verifica con la segnalazione di fallimento non appena una regola non sia rispettata;
- segnala il successo della verifica.

Esempio di input per il primo quadrato:

```
16 3 2 13 5 10
11 8 9 6 7 12 4
15
14 1
```

Programmazione di classi (ripasso per gli esercizi!)

Per programmare una classe è bene seguire le seguenti fasi:

1/ Progettare la classe

La progettazione va fatta prima di cominciare a scrivere codice!

1.1 Processo di astrazione

Stabilire quali sono le caratteristiche essenziali degli oggetti della classe, e fare un elenco delle operazioni che sara` possibile compiere su di essi.

1.1 Definire l'interfaccia pubblica
Definire i costruttori e i metodi da realizzare e per ciascuno di essi scrivere l'intestazione (specificatore di accesso, tipo di valore restituito, nome del metodo, eventuali parametri espliciti).
1.2 Definire le variabili di esemplare
E' necessario individuare le variabili di esemplare (***quante***? ***quali***? dipende dalla classe!) ed eventuali costanti. Per ciascuna variable si deve, poi, definire ***tipo*** e ***nome***.

2/ Scrivere il codice

2.1 Verificate le impostazioni del vostro editor di testi, al fine rendere piu` efficiente il vostro lavoro di programmazione.
L'editor deve essere impostato nel seguente modo:
- tabulazione pari a 3 caratteri;
- indentazione automatica pari a 3 caratteri;
- conversione tabulazione in caratteri;
- numero di riga evidenziato.

2.2 Definire le costanti e le variabili di esemplare
Dopo aver definito le costanti e le variabili di esemplare, compilare e correggere eventuali errori.

2.3 Scrivere l'intestazione dei costruttori e dei metodi pubblici
Scrivere l'intestazione dei costruttori e dei metodi pubblici seguita dal corpo vuoto, come nell'esempio che segue:

```
public double getBalance()
{
}
```

Se un metodo restituisce un tipo di dati, scrivere nel corpo la sola istruzione ***return*** seguita da:
- false se il metodo restituisce un valore di tipo **boolean**
- 'a' se il metodo restituisce un valore di tipo **char**
- 0 se il metodo restituisce un valore numerico (**byte**, **short**, **int**, **long**, **float**, **double**)
- null se il metodo restituisce un riferimento a un oggetto

E' importante programmare l'istruzione ***return*** perche' questo permette di compilare senza errori i metodi prima di scriverne il codice.
Quando si scrive l'effettivo codice del metodo, si modifica l'istruzione ***return*** secondo quanto richiesto dal metodo.
Se il metodo non restituisce un tipo di dati non serve scrivere nulla.
Dopo aver scritto l'intestazione dei metodi, si compili e si correggano eventuali errori.

2.4 Realizzare i metodi
Non appena si è realizzato un metodo, si deve compilare e correggere gli errori di compilazione. Se il metodo e' complicato, compilare anche prima di terminare il metodo.
NON ASPETTARE DI AVER CODIFICATO TUTTA LA CLASSE PER COMPILARE!!!
Se fate questo, vi troverere, probabilmente, a dover affrontare un numero elevato di errori, col rischio di non riuscire a venirne a capo in un tempo ragionevole (come quello a disposizione per la prova d'esame).
SE LA COMPILAZIONE GENERA PIU' DI UN ERRORE, CORREGGERE SOLO IL PRIMO E POI RICOMPILARE.
Un unico errore puo' causare, infatti, piu' messaggi di errore diversi e, quindi la correzione del primo errore puo' far scomparire piu' messaggi di errore, non solo il primo.

3/ Collaudare la classe

Scrivere una *classe di collaudo* contenente un metodo main, all'interno del quale vengono definiti e manipolati oggetti appartenenti alla classe da collaudare.

- E` possibile scrivere ciascuna classe in un file diverso. In tal caso, ciascun file avra' il nome della rispettiva classe ed avra' l'estensione .java. Tutti i file vanno tenuti nella stessa cartella, tutti i file vanno compilati separatamente, solo la classe di collaudo (contenente il metodo main) va eseguita.
- E` possibile scrivere tutte le classi in un unico file. In tal caso, il file .java deve contenere una sola classe public. In particolare, la classe contenente il metodo main deve essere public mentre la classe (o le classi) da collaudare non deve essere public (non serve scrivere private, semplicemente non si indica l'attributo public). Il file .java deve avere il nome della classe public

Esercizio 3

Argomento: classi, tipo booleano, variabili statiche

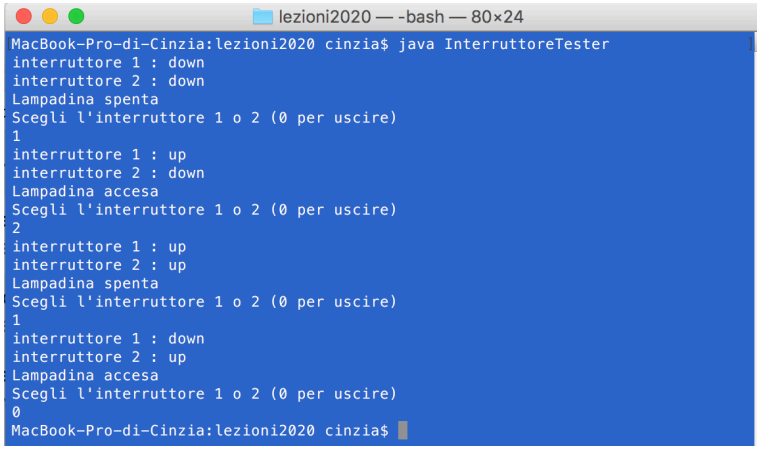
Scrivere la classe Interruttore i cui oggetti rappresentano degli interruttori. Ogni interruttore puo' assumere due stati, con il significato che l'interruttore sia su' o giu' (up/down nella descrizione testuale da riportare in output ma gestibile con una variabile booleana). Tutti gli interruttori sono collegati ad una stessa lampadina regolandone l'accensione e spegnimento. Ogni volta che un interruttore cambia stato, anche la lampadina cambia stato (accesa-> spenta, spenta->accesa).

La classe deve contenere: tutte le variabili d'istanza e le variabili statiche ritenute necessarie a descriverne lo stato e i seguenti costruttori e metodi:

```
// Costruttore: inizializza l'interruttore in stato "down"
// Assumiamo "down" corrisponda a false e "up" a true
public Interruttore();

// Metodo di accesso della variabile di esemplare interruttore
public boolean getStatusInterruttore();

//Metodo di accesso alla variabile statica lampadina
public boolean isBulbOn();

//Modificatore: cambia lo stato dell'interruttore
//(e della lampadina!)
public void changeStatus();

//Stampa lo stato dell'interruttore: up/down a seconda
// che status sia true o false
public String printStatus();
```

Per testare la classe, scrivere un programma TestInterruttore che crea due interruttori (oggetti della classe Interruttore) e poi, in maniera iterativa, offre all'utente la possibilita' di agire su uno dei due interruttori (visualizzando l'esito dell'operazione) a seconda che venga inserito il numero 1 o 2, o di terminare l'esecuzione se l'input e' 0.

Esempio:



Esercizio 4

Argomento: progettazione e collaudo di classi

Si scriva la classe **MyComplex**, che rappresenta un numero complesso, la cui interfaccia pubblica è specificata nel file MyComplex.java disponibile tra i file messi a disposizione per il laboratorio (e contiene come commenti le uniche nozioni sui numeri complessi che e` necessario conoscere ai fini di questo esercizio).
Collaudare la classe **MyComplex** con la classe di prova **MyComplexTester**, disponibile tra i file messi a disposizione per il laboratorio. Il programma riceve due numeri complessi da standard input, uno per riga, nel formato "x y ", dove x (parte reale) e y (parte immaginaria) sono separate dal carattere spazio ' '. Fornisce a standard output somma, sottrazione, prodotto, divisione, elementi inversi e coniugati dei numeri. Provare con numeri semplici, in modo da verificare i risultati ottenuti.
Effettuare il collaudo in due modi: ciascuna delle due classi in un file .java ad essa omonimo (con entrambe le classi public), oppure entrambe le classi in un solo file **MyComplexTester.java** (con **MyComplexTester** public e **MyComplex** no)

Osservazione:

Molti dei metodi da implementare ricevono come parametro esplicito un (riferimento a un) oggetto di tipo MyComplex, z, da elaborare insieme all'oggetto a cui fa riferimento il parametro implicito (this)

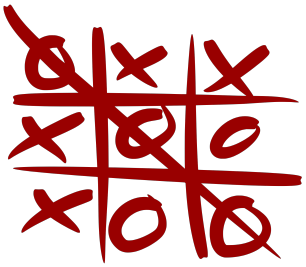
In tutti i casi in cui occorra ispezionare la parte reale e la parte immaginaria di z si potrebbe scrivere z.getRe() e z.getIm(), passando attraverso l'interfaccia pubblica degli oggetti, ma questo non e' necessario: si puo' scrivere direttamente z.realPart e z.imagPart, perche' stiamo scrivendo codice di metodi della classe MyComplex, che puo' accedere anche alle caratteristiche private della classe, comprese le variabili di esemplare DI QUALSIASI OGGETTO che sia esemplare della classe, non solo dell'oggetto che ha ricevuto come parametro implicito.

Non bisogna pensare alle variabili di esemplare private di un oggetto come a una sua caratteristica a cui non si possa accedere "dall'esterno": semplicemente, non vi si puo' accedere da codice che sia esterno alla classe di cui l'oggetto e' esemplare! Questo vuol dire che un metodo che venga invocato con un esemplare di MyComplex come parametro implicito potra' accedere alle caratteristiche private di altri oggetti di tipo MyComplex di cui conosca il riferimento (ad esempio perche' l'ha ricevuto come parametro esplicito)

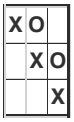
Questa strategia e' molto potente, perche' consente a metodi di esemplare di ispezionare o anche modificare variabili di esemplare di tutti gli oggetti creati come esemplari della classe stessa, anche se l'interfaccia pubblica della classe non lo consente!

Esercizio 5

Argomento: progettazione di classi, uso di array bidimensionali



Scrivere un programma per giocare a "tris" (tic-tac-toe in inglese), il classico gioco in cui due giocatori dispongono alternativamente un proprio contrassegno in una casella di una scacchiera 3 x 3 finché uno dei due non pone tre contrassegni in una fila orizzontale, verticale o diagonale.



Il programma inizia visualizzando la scacchiera vuota, come segue (ogni puntino rappresenta una casella vuota)
I...I
I...I
I...I
e chiedendo al primo giocatore di inserire le coordinate della casella in cui vuole porre il suo contrassegno (che sara` un carattere X).
Le coordinate si indicano con due numeri interi (valori ammessi: 0, 1 o 2), il primo numero essendo l'indice di riga (a partire dall'alto) ed il secondo l'indice di colonna (a partire da sinistra).
Il programma deve verificare se la casella richiesta è libera oppure no. Nel primo caso visualizza la scacchiera aggiornata e chiede all'altro giocatore di inserire la propria mossa (verrà usato il contrassegno O). Nel secondo caso, invece, il programma fornisce una segnalazione d'errore, visualizza nuovamente la stessa scacchiera visualizzata in precedenza e chiede al giocatore di inserire una nuova mossa; analogo comportamento si verifica se il giocatore introduce delle coordinate non valide, ma il messaggio d'errore deve essere diverso.
Il programma deve essere in grado di segnalare la vittoria di uno dei due giocatori qualora questa avvenga, oppure di porre fine alla partita quando la scacchiera è piena senza che uno dei due giocatori abbia raggiunto la vittoria.
Al termine di una partita, il programma chiede al giocatore se intende giocare un'altra partita oppure no: nel secondo caso il programma termina.

Risolvere il problema in due passi

1. Scrivere una classe **MyTris** che rappresenti la scacchiera e la cui interfaccia pubblica è definita a questo link (css)
2. Scrivere poi una classe eseguibile che usi **MyTris**, e che gestisca una partita a tris tra due giocatori, seguendo il comportamento descritto sopra.

Esercizio 6

Argomento: progettazione di classi

Implementare la classe Player che rappresenta un giocatore e la classe eseguibile Risiko che simula un turno al noto gioco.

In particolare, la classe Player dovra' tener conto del nome del giocatore,
dei tre lanci di dado (facciamo che siano sempre tre) e del punteggio (numero di tiri vincenti) . Nella classe dovranno essere inoltre definiti:

```
// costruttore: il punteggio iniziale e' 0 cosi' come
// i valori dei tiri dei tre dadi
public Player(String aName)

// metodi di accesso
public String getName();
public String getScore();

// simula un turno di lancio di dadi attribuendo a ciascun
// lancio un valore casuale tra 1 e 6
public void turno()

// restituisce un riferimento a un nuovo array di interi in
// cui i valori ottenuti nei tre lanci sono ordinati
public int[] sortDice()

//aggiorna il punteggio incrementandolo di una unita'
public void addPoint()

//resetta il punteggio
public void resetScore()

//restituisce una stringa con il nome del giocatore e
//i valori dei lanci dei dadi
public String toString()
```

Il metodo turno() della classe Player fa uso della classe java.util.Random che rende disponibili metodi per generare numeri pseudo-casuali, ovvero quasi casuali. Consultare l'interfaccia pubblica della classe Random, in particolare il metodo int nextInt(int n).

Scrivere una classe eseguibile Risiko.java che effettua le seguenti operazioni:
- acquisisce da standard input il nome del primo giocatore
- acquisisce da standard input il nome del secondo giocatore
- definisce un oggetto di classe Player per ciascuno dei due giocatori
- lancia tre volte il dado per ciascun giocatore (si usa il metodo turno())
- stampare i risultati di ciascun giocatore (stampare implicitamente con toString())
- dichiari un riferimento ad un array di interi e gli assegni il risultato dell'invocazione del metodo sortDice() per il primo giocatore
- stampare i valori ordinati dei lanci seguite dal nome del giocatore
- ripetere le due operazioni precedenti per il secondo giocatore
- confrontore i risultati ottenuti nei lanci dai due giocatori assegnando un punto (chiamata al metodo addScore()) al giocatore che vince ciascun confronto
-verificare chi sia il vincitore e stampare in uscita il nome e i punteggi
- invia a standard output il risultato dei lanci

- calcola il vincitore e invia il risultato a standard output (metodo String vincitore(...)).

In esecuzione la classe Risiko dovra' produrre un risultato simile al seguente:

```
$ java Risiko
Giocatore n. 1? Pippo
Giocatore n. 2? Pluto
lanci di Pippo: 2 3 1
lanci di Pluto: 6 6 2
lanci ordinati
1 2 3  Pippo
2 6 6  Pluto
Pluto vince 3 a 0
```

NB: in caso di pareggio vince il secondo giocatore (che si difende).