Programmazione Object Oriented

Esercitazioni Aggiuntive

Esercizio – Scuola di Sci

Scrivere l'interfaccia `Course` inserendola nel pacchetto `skischool`. L'interfaccia contiene i metodi:

- `String getCourseName();`
- `int getMinimumNumberOfParticipants();`
- `int getMaximumNumberOfParticipants();`
- `int getNumberOfParticipants();`
- `boolean addParticipant();`
- `boolean isCourseActivated();`

Scrivere le classi `SkiCourse` e `SnowboardCourse` che implementano l'interfaccia `Course`.
Inserire le classi nel pacchetto `skischool`.

Le classi devono essere implementate in modo da memorizzare un identificativo, il numero di partecipanti, la data di inizio corso e la durata (il numero di giorni). Lanciare un'eccezione non controllata `BadArgumentException` se la durata è negativa al momento della creazione.

Il corso di sci viene attivato solo se si raggiungono almeno quattro partecipanti e può avere al massimo otto partecipanti.

Il corso di snowboard viene attivato solo se si raggiungono almeno sei partecipanti e può avere al massimo dieci partecipanti.

Il metodo `addParticipant` restituisce `true` se è stato possibile aggiungere il partecipante al corso (il numero totale di partecipanti è inferiore al numero massimo consentito) e `false` altrimenti.

Il metodo `isCourseActivated` restituisce `true` se è stato raggiunto il numero minimo di partecipanti e quindi il corso è attivato.

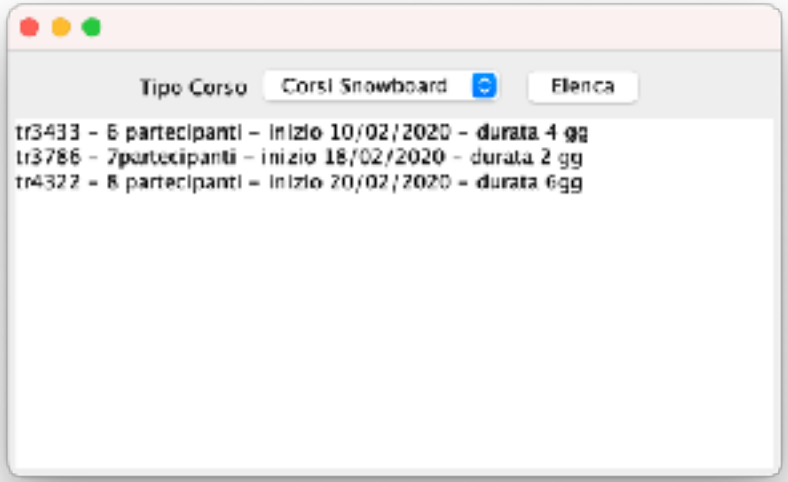
Esercizio – Scuola di Sci

Scrivere la classe `CourseList` (nel pacchetto `skischool`) che modelli i corsi offerti da una scuola di sci attraverso i seguenti metodi:

- `void addCourse(Course c)` che inserisca un corso nell'archivio in modo ordinato rispetto alla data.
- `Course getCourse(int i)` che restituisca l'i-esimo corso nell'archivio;
- `Course removeCourse(int i)` che rimuova l'i-esimo corso dall'archivio restituendolo.
- `ArrayList<Course> getCourses(String type)` che restituisca tutti i corsi in base al tipo (`SkiCourse` o `SnowboardCourse`).

Aggiungere poi il codice per

- creare una `CourseList` con almeno cinque oggetti `SkiCourse` e cinque oggetti `SnowboardCourse` con valori casuali (tutti i corsi devono essere attivi).
- Realizzare un'interfaccia grafica che consenta di visualizzare tutti gli `SkiCourse` o tutti gli `SnowboardCourse` utilizzando una `JCombobox`.



Esercizio – Mensa

Implementare la classe `Card` che modelli i vari tipi di tesserino utilizzati per una mensa universitaria. Ogni tesserino è caratterizzato da codice, nome, cognome, e dalla proprietà `active` (quest'ultima è una variabile booleana indicante se il tesserino è utilizzabile). Fornire i metodi:

- `activate()` che setti a `true` lo stato della variabile attivo (se `active` è già `true` lancia l'eccezione `RunTimeException`).
- `deactivate()` che setti a `false` lo stato della variabile attivo (se `active` è già `false` lancia l'eccezione `RunTimeException`).

Implementare poi due sottoclassi:

- `StudentCard` caratterizzata da matricola, scadenza, saldo, fascia, bonus con i metodi
  - `double calcolaPrezzo()` che calcoli il prezzo del pasto di uno studente in base alla fascia: coloro che appartengono alla fascia A pagano 2.50€, mentre coloro che appartengono alla fascia B pagano 1.50€. Inoltre gli studenti vincitori di borse di studio (i cui tesserini hanno la variabile `bonus` settata a `true`) hanno lo sconto di 1€.
  - `boolean isBonus()` che restituisca `true` se lo studente ha vinto una borsa di studio.
  - `double simulatePayment()` che simuli il pagamento di un pasto. Se il tesserino è scaduto lancia l'eccezione controllata `ExpiredCardException`, altrimenti sottrae al saldo il costo del pasto. Nel caso in cui il saldo è insufficiente per pagare il pasto viene lanciata l'eccezione non controllata `InsufficientBalanceException`. Il valore restituito corrisponde all'importo pagato dallo studente.
  - `void pay(double x)` che aggiorni il saldo con la somma x passata come argomento. Se la somma da versare è negativa lancia l'eccezione `RunTimeException`.
- `StaffCard` caratterizzata dalle variabili `department`, `spentAmount`, `category` (che può essere `professor` or `administrative`). Corredare la classe con i metodi
  - `double pay()` che aggiunga a `spentAmount` il costo del pasto che dipende dal valore di categoria. Nel caso di `professor` l'importo del pasto è 1.60€, nel caso di `administrative` l'importo è 4.00€. Il metodo restituisce l'importo pagato dal personale.
  - `void changeCategory()` che modifichi la categoria del tesserino.

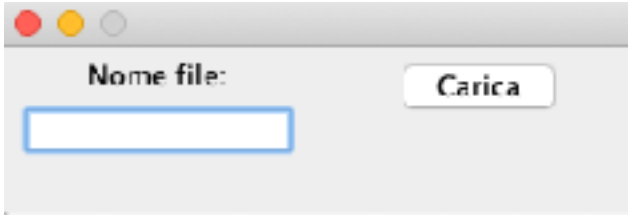
Esercizio – Mensa

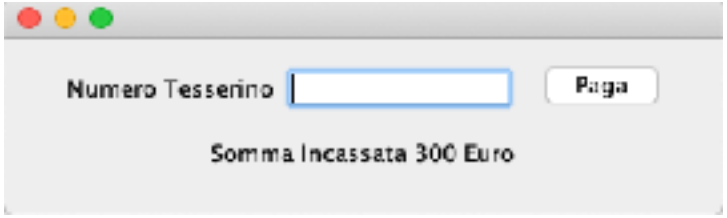
Scrivere la classe `MealPayment` che modelli una collezione di `Card` e fornisca i seguenti metodi:

- `void addCard(Card t)` che inserisca un tesserino nell'archivio.
- `boolean useCard(int code)` che simuli il pagamento di un pasto per la persona in possesso del tesserino con codice `code`. Il metodo non gestisce le eventuali eccezioni lanciate. Restituisce `true` se il codice è presente nell'archivio, `false` altrimenti.
- `double computeTotal()` che restituisca la somma incassata fino a quello istante.
  N.B. *Utilizzare una variabile che tenga traccia di tale somma durante i pagamenti.*
- `MealPayment getCardByType(int x)` che restituisca l'elenco dei tesserini di una certa tipologia (`x=0` indica la tipologia `StudentCard` e `x=1` una `StaffCard`). Per valori di x diversi da `0` e `1` viene la lanciata l'eccezione controllata `InvalidParameterException`.
- `double getSpentAmount()` che restituisca la somma spesa da tutti i possessori di `Card`.

Implementare un'interfaccia grafica per

- caricare da un file una lista di tesserini;
- effettuare il pagamento di un pasto (chiedere di inserire solo il codice del tesserino) e visualizzare la somma incassata.





Esercizio – Gestione Albergo

Realizzare una piccola applicazione con GUI per la gestione delle prenotazioni di un albergo.

Il sistema deve tenere traccia

- dei clienti (cognome, nome, telefono, numero di carta di credito),
- delle stanze (numero, piano, tipologia, servizi, telefono),
- delle prenotazioni (check-in, check-out, cliente, stanza).

All'inizio il sistema chiede il nome del file dal quale prelevare i dati, permettendo poi di selezionare tra *Richiesta Prenotazione* e *Gestione Prenotazione*.

Con *Richiesta Prenotazione* è possibile inserire le date di check-in/check-out, la tipologia (i.e., singola, doppia, matrimoniale) della stanza e chiedere la disponibilità.

A questo punto è possibile (i) visualizzare i *Dettagli* dell'eventuale stanza disponibile, e (ii) *Prenotare* la stanza

Nel primo caso vengono evidenziate le caratteristiche della stanza in formato testo.

Nel secondo caso si richiedono i dati del cliente e viene mostrato un messaggio di conferma della prenotazione oppure un messaggio d'errore appropriato (e.g., carta di credito non valida, stanza già prenotata).

Con *Gestione Prenotazione* si richiedono le informazioni del cliente e la richiesta (se cancellazione o modifica di caratteristiche).

N.B. Quando una prenotazione è passata, cioè la data di check-out è antecedente la data odierna, questa deve essere cancellata dall'archivio.

Esercizio - Aeroporto

Un volo contiene le informazioni sulla tratta (aeroporto di partenza e di destinazione), codice volo, compagnia aerea, tipologie di biglietto acquistabili (i.e., economy, business, first class) e la lista di posti con lo stato (i.e., disponibile/occupato).

Implementare il metodo `List<Seat> filterSeats(Filter f)` che restituisca la lista di posti che soddisfino il filtro `f`. Sovrascrivere i metodi `toString`, `equals` e `clone` di `Object` in maniera da massimizzare il riutilizzo del codice.

Un posto ha un numero di fila, una lettera indicante la posizione nella fila e uno stato (i.e., libero/prenotato).

Un filtro permette di filtrare i posti per numero di fila, posizione nella fila e stato.

Una prenotazione tiene traccia delle informazioni anagrafiche del cliente e delle informazioni sul volo prenotato (codice volo, tipo biglietto, posto).

Scrivere la classe `FlightList` che modelli una collezione di voli e fornisca i seguenti metodi:

- `void addFlight(Flight v)` che inserisca un volo nell'archivio. Se l'oggetto `v` ha un codice volo uguale ad uno già presente nell'archivio viene lanciata l'eccezione controllata `BadCodeException`.
- `FlightList search(String o, String d)` che restituisca un oggetto `FlightList` contenente tutti i voli che partono da `o` ed arrivano a `d`.

Esercizio - Aeroporto

Considerando le classi ai punti precedenti, scrivere una classe Java che realizzi un'interfaccia grafica per la prenotazione di posti su un volo.

Nell'interfaccia grafica

- deve essere possibile specificare una tratta e alla pressione di un pulsante visualizzare in un area di testo i voli abilitati su quella tratta.
- i voli visualizzati devono poter essere selezionati con un pulsante a menu (casella combinata).
- una volta selezionato un volo, permette di scegliere il posto (tra quelli liberi). Usare il metodo `filterSeats` creando un opportuno filtro.
- una volta selezionato il posto si apre una finestra che consente di immettere i dati anagrafici del passeggero.
- una volta terminata la prenotazione, i dati del volo devono essere registrati su file.

Implementare un'eccezione non controllata per segnalare che un volo non ha posti liberi. L'eccezione deve essere lanciata quando si seleziona un volo senza posti liberi.