GESTIONE LOCALE *SOLELUNA*.

Il programma, pensato per essere utilizzato dal gestore del locale, o da un suo dipendente, che riceve i clienti di persona o viene contattato telefonicamente, è diviso in due package. Uno rappresenta l'accesso al programma, vi troviamo il *main* nella classe *gestisci_locale*, separato dalle classi che effettivamente creano e gestiscono le prenotazioni proprio al fine di utilizzarlo solo come interfaccia con l'utente. Nell'altro package troviamo la classe *affitto* estesa da *catering*, a sua volta estesa da *animazione*. Inoltre troviamo la classe *eccezioni* che estende la classe Exception e *gestisci_prenotazioni* che è l'effettivo motore del programma.

- ## Le prenotazioni (affitto, catering, animazione)

La classe *affitto* rappresenta il servizio basico del locale. Lo si può prenotare con un nome e una data (in altre parole gli oggetti della classe *affitto* sono prenotazioni di questo servizio), ha un prezzo fisso, metodi per visualizzare e modificare il nome e un metodo per visualizzare l'intero oggetto. Il servizio di catering prevede comunque l'affitto del locale, per questo ho deciso di estendere la classe *affitto*. Di conseguenza la classe *catering* eredita variabili e metodi da *affitto*, alcuni vengono aggiunti ed altri sovrascritti. Lo stesso discorso vale per il servizio di animazione, che prevede sia l'affitto che il catering. Tutti i metodi e le variabili hanno il modificatore di visibilità di tipo *private* oppure *protected*, al fine di impedire al *main*, che si trova in un altro package, di creare prenotazioni "non controllate". Tutte e tre le classi implementano l'interfaccia *Serializable*, al fine di poter salvare la lista di prenotazioni.

- ## Gestisci prenotazioni

Questa classe è il fulcro del programma, può chiamare i metodi di *affitto* e delle sue sottoclassi. Gestisce la lista delle prenotazioni, creata come *LinkedList* (la necessità principale di una lista di prenotazioni ordinate per data è quella di poter inserire gli elementi nel mezzo della lista velocemente). Tra i metodi di questa classe quelli che possono essere chiamati dal *main* sono principalmente *void* e, soprattutto, non prendono nessun parametro in entrata. In questo modo si può modificare uno dei due package (essendo le due classi in questione quelle che comunicano nei due diversi package) senza modificare l'altro.

Inoltre la classe gestisce le date, attraverso *Calendar*(utilizzato per creare la data odierna o per ottenere la data di domani e così via) e *DateFormat* (per *traformare* le stringe in date e viceversa). Infatti la data di una prenotazione viene sia inserita (dal gestore/dipendete) che salvata (nell'oggetto affitto/catering/animazione) come stringa. Per poter confrontare due date tra loro o con la data corrente, però, c'è bisogno di trasformare queste stringhe in oggetti Date, al fine di poter utilizzare determinati metodi della classe *Date* del pacchetto *util*.

Per salvare la lista delle prenotazioni la classe ha un metodo per serializzare gli oggetti della LinkedList e creare un file chiamato "prenotazioni_soleluna.dat". Al fine di caricare la lista salvata all'avvio del programma, il costruttore di questa classe (chiamato dal main) invoca il metodo creato per deserializzare il file, il quale assegna l'oggetto appena deserializzato alla variabile d'istanza della classe (la LinkedList *pren*). Al primo avvio il file non esiste ancora quindi s'incorre in un errore, che viene gestito, quindi non crea disagi all'utente.

- **Gestione degli errori, degli input e la classe *eccezioni***

La gestione degli errori avviene sempre nella classe *gestisci_ prenotazioni*, attraverso l'utilizzo del costrutto *try-catch*.

L'eccezione *NoSuchElementException* viene lanciata dal metodo next() di un iteratore quando non sono presenti nella lista ulteriori elementi. Utilizzando un costrutto do-while per iterare con un iteratore e inserendo il metodo nel try si può evitare di interrompere il programma e semplicemente fare una comunicazione. Ho utilizzato questa procedura nei metodi void, per comunicare al gestore che la lista delle prenotazioni è vuota e allo stesso tempo gestire il primo avvio (la lista è per forza vuota) o il caso in cui, per qualche ragione, fossero state disdette tutte le prenotazioni. Per quanto riguarda invece i metodi in cui, attraverso l'iteratore, si confrontano delle date, ho usato un approccio differente. Nel metodo *verifica_data*, il quale viene anche chiamato dagli altri metodi in questione ed è il primo che necessita di un iteratore, inserisco come primo elemento della lista una prenotazione "fasulla", con data antecedente a quella odierna (sono così sicura che sia sempre al primo posto della lista), che si usa per fare i vari confronti e che viene poi eliminata alla fine dei metodi che la utilizzano. L'eliminazione di questa prenotazione avviene tramite il metodo remove() della classe LinkedList (diverso da quello dell'iteratore), che rimuove il primo elemento della lista.

Le eccezioni del tipo *ParseException* lanciate durante la trasformazione di una stringa in una data sono state usate in combinazione ai costrutti try-catch e do-while per verificare l'input della data da parte del gestore, e continuare a chiedere una data del tipo corretto. Allo stesso modo è stata utilizzata l'eccezione *InputMismatchException* per validare l'input riguardante il numero di bambini.

Quando l'input richiesto è un carattere la strategia utilizzata prevede sempre un costrutto do-while, questa volta però, affiancato da un if-else. La creazione della classe *eccezioni* riguarda un proprio uno di questi casi. Il metodo privato *crea()* , che ritorna un oggetto di tipo *affitto*, quando chiamato per una prenotazione di tipo animazione, deve chiedere in input un carattere. La verifica della correttezza del carattere è costruita come sopra, quindi non si procede fino a che non si mette il carattere corretto, infine si può fare il *return*. Nonostante ciò il compilatore teme che in qualche modo possa non avvenire il *return*, per questo è stata creata la classe *eccezioni* che nel caso si uscisse dal ciclo senza un input corretto (cosa teoricamente impossibile) viene lanciata attraverso il comando *throw* lanciando il messaggio di errore e facendo terminare il programma.

La gestione degli input nel *main,* invece, è molto semplice. Essendo l'interfaccia testuale già in un ciclo do-while ed avendo utilizzato *switch-case* per le varia opzioni, è bastato inserire come *default* il messaggio di input errato.