PROGETTO SUDOKU

Gaia Assunta Bertolino
Mat. 209507 A.A. 2020/2021
Corso di Programmazione orientata agli oggetti

Organizzazione delle classi:
Il progetto è stato organizzato secondo quattro classi (di cui la prima sviluppata a lezione):

1) classe Backtracking<P,S> nei tipi generici P e S
2) classe PuntoDiScelta che definisce la cella individuata da due coordinate
3) classe Sudoku che estende la classe Backtracking nei tipi generici PuntoDiScelta e Integer
4) classe SudokuGUI

Nella seguente relazione, viene tralasciato il funzionamento della classe di Backtracking<P,S> vista a lezione.
Il programma è strutturato attraverso la definizione del tipo PuntoDiScelta, la scrittura dell'algoritmo di funzionamento di gioco nella classe Sudoku e l'implementazione di una GUI d'interazione in SudokuGUI che rende disponibili anche degli strumenti di salvataggio e rispristino di una partita.

class PuntoDiScelta

La classe definisce il tipo PuntoDiScelta, rendendo la scrittura del codice del sudoku più agevole, in quanto racchiude, sottoforma di variabili d'istanza, le due coordinate che individuano una cella (e dunque un punto di scelta secondo l'algoritmo di risoluzione per backtracking). I metodi da cui è composta sono:

- **int getI() / int getJ() ->** metodi getters che restituiscono rispettivamente i valori delle variabili d'istanza che individuano un punto di scelta
- **String toString() ->** metodo sovrascritto che restituisce una stringa rappresentativa di un PuntoDiScelta
- **boolean equals(Object o) ->** metodo sovrascritto che verifica l'uguaglianza fra due oggetti di tipo PuntoDiScelta confrontando la coincidenza dei valori delle due variabili d'istanza

class Sudoku extends Backtracking<PuntoDiScelta, Integer>

La classe struttura l'algoritmo di risoluzione del gioco del sudoku. Lo fa innanzitutto definendo una base di gioco, una lista di punti di scelta, una lista di soluzioni e una variabile *soluzione* che sarà utile per gestire il limite del numero di soluzioni da calcolare. I metodi da cui è composta la classe sono:

- **Sudoku() ->** costruttore che inizializza la base ad una matrice di interi di dimensione 9x9 impostando tutte le sue celle al valore 0. Inoltre, inizializza i punti di scelta che possono essere selezionati invocando il metodo *puntiDiScelta()*.
- **int[][] getBase() ->** metodo che restituisce una copia profonda della base di gioco
-

- **boolean verificaGriglia(int i, int j, int v) ->** metodo che, assegnati due indici i e j che individuano univocamente una cella, verifica che il valore v che si vuole assegnare a questa cella non violi la regola per cui non debba essere già presente nella sottomatrice contenente la cella.
- **void imposta(int i, int j, int v) ->** metodo che assegna ad una cella della base un determinato valore v passato come argomento. Viene richiamato dal metodo *assegna(PuntoDiScelta ps, Integer s)*
- **boolean assegnabile(PuntoDiScelta ps, Integer s) ->** metodo che verifica se è possibile assegnare un valore s ad una cella (individuata dal punto di scelta passato come argomento) senza violare le regole stesse del gioco. Viene dunque verificato che il valore sia accettabile altrimenti si solleva una eccezione; si verifica poi che la cella non sia già occupata e per fare ciò implementa un meccanismo che riesce a rendere valido l'algoritmo di backtracking (senza modificarlo direttamente) anche nel caso in cui si voglia calcolare la soluzione per tentativi di una partita in cui delle celle sono già state occupate. Infatti, se non ci fosse l'accorgimento creato, l'algoritmo procederebbe a sostituire anche le celle fissate con dei valori validi per la base mentre, attraverso l'implementazione realizzata, vengono calcolate solo delle soluzioni in cui il valore delle celle inserite dall'utente non vengono modificate, rendendo non accettabili valori diversi da quello già contenuto in ciascuna di esse. Successivamente viene invece verificato che il valore da inserire non sia già presente nella riga, colonna o sottomatrice che contengono la cella.
- **void assegna(PuntoDiScelta ps, Integer) ->** metodo sovrascritto della classe Backtracking<P,S> che richiama a sua volta il metodo *imposta(int i, int j, int v)* per assegnare un valore in una cella della base
- **List<PuntoDiScelta> puntiDiScelta() ->** metodo che restituisce una lista di tutti i possibili punti di scelta
- **void deassegna(PuntoDiScelta ps, Integer s) ->** metodo sovrascritto della classe Backtracking<P,S> che imposta a 0 la cella individuata dal punto di scelta
- **void scriviSoluzione(PuntoDiScelta p) ->** metodo sovrascritto della classe Backtracking<P,S> che viene richiamato quando è stata calcolata una soluzione e dunque viene ricopiata la base ottenuta in una nuova matrice, a sua volta salvata all'interno della lista delle soluzioni. Viene poi incrementata la variabile che tiene conto di quante soluzioni sono state trovate.
- **Collection<Integer> scelte(PuntoDiScelta p) ->** metodo sovrascritto della classe Backtracking<P,S> che costruisce una lista di quei possibili valori che una cella può avere. Nel caso del sudoku, i valori vanno da 1 a 9 compresi
- **boolean ultimaSoluzione(PuntoDiScelta p) ->** metodo sovrascritto della classe Backtracking<P,S> che verifica se si sono calcolate il numero di soluzioni voluto.
- **boolean esisteSoluzione(PuntoDiScelta p) ->** metodo sovrascritto della classe Backtracking<P,S> che verifica di aver appena aggiunto un valore assegnabile all'ultima cella della base. In caso affermativo, vuol dire che si è calcolata una soluzione per il gioco.
- **void risolvi() ->** metodo sovrascritto della classe Backtracking<P,S> che richiama il metodo *tentativo(List<PuntoDiScelta> ps, PuntoDiScelta p)* della superclasse sulla prima cella della base, così da innescare il meccanismo di risoluzione del backtracking

class SudokuGUI

La classe contiene al suo interno altre classi che definiscono la reale implementazione della GUI. Dunque, SudokuGUI ha il solo compito di invocare la costruzione di una finestra inziale di tipo FinestraIniziale con un sommario delle regole del gioco e da cui è possibile accedere al gioco vero e proprio.

class FinestraIniziale extends JFrame implements ActionListener

La classe ha lo scopo di dare una implementazione alla finestra di benvenuto all'utente. E' composta dai seguenti metodi:

- **FinestraIniziale() ->** costruttore che definisce le caratteristiche della pagina, compresa una JTextArea non editabile che contiene le regole e un bottone che permette di iniziare il gioco
- **boolean uscita() ->** metodo che viene richiamato quando si prova a chiudere una finestra. Esso visualizza una ulteriore finestra di conferma dell'uscita e, eventualmente, permette di non chiudere realmente la finestra.
- **void actionPerformed(ActionEvent e) ->** metodo che deriva dall'implementazione di ActionListener. Il metodo viene richiamato quando si interagisce con il bottone della finestra a cui è stato aggiunto un ActionListener attraverso il metodo component.addActionListener(this) che fa iniziare la partita rendendo invisibile la finestra iniziale e generando la finestra di gioco

class GiocoGUI extends JFrame implements ActionListener

La classe ha lo scopo di implementare una GUI rappresentativa del gioco del sudoku. Infatti, viene visualizzata una griglia in cui è possibile inserire dei valori (valutati secondo le regole del gioco), salvare una partita, aprire una partita salvata e anche visualizzare delle soluzioni che tengano conto delle celle già inserite dall'utente. Essa è composta dai seguenti metodi:

- **GiocoGUI() ->** costruttore che definisce le caratteristiche della base di gioco, tra cui un pannello di 81 JTextField, ciascuno dei quali rappresenta una cella del sudoku. Inoltre, disattiva i pulsanti di salvataggio e di nuova partita in quanto ancora la base è vuota
- **boolean uscita() ->** metodo che viene richiamato quando si prova a chiudere una finestra. Esso visualizza una ulteriore finestra di conferma dell'uscita e, eventualmente, permette di non chiudere realmente la finestra.
- **void valuta(JTextField t, int i) ->** metodo che viene richiamato quando si prova ad inserire un valore in una cella e, per farlo, richiama a sua volta il metodo *assegnabile(PuntoDiScelta ps, Integer s)* per verificare che sia accettabile e, in caso contrario, genera una finestra di tipo FinestraErrore con un messaggio di errore. Inoltre, gestisce le eccezioni con un opportuno blocco try-catch generate nel caso in cui si provi ad inserire un numero minore di 1 o maggiore di 9 o un carattere non numerico.

- **void ActionPerformed(ActionEvent e) ->** metodo che deriva dall'implementazione di ActionListener. Il metodo viene richiamato ogni volta che si interagisce con un componente della finestra a cui è stato aggiunto un ActionListener attraverso il metodo component.addActionListener(this). In base al componente cliccato, invoca un metodo che svolge le istruzioni richieste. Inoltre, gestisce le eccezioni di tipo IOException che possono essere sollevate da quei metodi che lavorano coi file per fare il salvataggio o il ripristino di una partita e verifica anche che, ogni volta che si modifica una cella del sudoku, siano opportunamente valutate quali funzioni siano richiamabili attraverso i bottoni della GUI così che, ad esempio, inserito anche un solo valore sia già possibile operare un salvataggio della partita.
- **void creaFile() throws IOException ->** metodo che viene richiamato prima di *salvataggio()* ma dopo il metodo *nuovo()* se l'utente ha scelto di creare un nuovo file prima di effettuare il salvataggio. Il file viene creato se non ne esiste uno con lo stesso nome e percorso
- **void caricaPartita() throws IOException ->** metodo che ha il compito di analizzare il contenuto del file da cui si vuole ripristinare la partita e lo fa mentre lo legge con un BufferedReader. Poiché il salvataggio avviene seguendo la forma matriciale, è facile operare il ripristino leggendo una riga alla volta e separandola opportunamente attraverso uno StringTokenizer che tagli lungo gli spazi. Successivamente, assegna ogni valore sia alla base del gioco che alla cella della GUI corrispondente, impostando che le celle con valore 0 siano vuote (e quindi modificabili dall'utente) mentre le altre bloccate. Le eccezioni sollevabili sono gestite con un blocco try-catch e comprendono eventuali problemi riconducibili ad un file non esistente o alla presenza di corruzioni del file per cui i dati presenti non sono utilizzabili per ricostruire la matrice. In entrambi i casi, vengono generate delle finestre di tipo FinestraErrore per segnalare il problema
- **void nuovapartita() ->** metodo che viene invocato quando si clicca sul bottone relativo ad una nuova partita e imposta sia le celle della base al valore 0 che quelle della GUI vuote ed editabili. Inoltre, disattiva i bottoni di salvataggio e nuova partita.
- **void nuovo() ->** metodo che viene chiamato quando si clicca sul bottone per operare il salvataggio della partita. Chiede dunque all'utente se vuole prima creare un nuovo file e, in caso affermativo, viene creata una finestra di tipo FinestraInput che rende possibile l'inserimento del nome e del percorso del file da creare, altrimenti richiama il metodo *salvataggio()*
- **void salvataggio() throws IOException ->** metodo che viene richiamato per operare il salvataggio della partita chiedendo prima all'utente di selezionare il file. Successivamente, viene salvata la base del gioco grazie ad un BufferedWriter che ricopia i valori seguendo la distribuzione matriciale separandoli con uno spazio. Inoltre, il metodo gestisce l'eccezione di inesistenza del file con un blocco try-catch e generando una finestra di tipo FinestraErrore
- **final class TxtFileFilter extends FileFilter->** classe privata e final che rende visibili solo cartelle e file di text (con estensione .txt) quando si prova a selezionare un file su cui salvare il gioco
- **private class FinestraErrore extends JFrame ->** classe che ha lo scopo di definire un finestra che possa mostrare dei messaggi di errore personalizzati all'utente. Ciò viene fatto passando un messaggio sottoforma di stringa al costruttore *FinestraErrore(String errore)*

- **private class FinestraInput extends JFrame ->** classe che ha lo scopo di definire una finestra che renda possibile all'utente inserire percorso e nome di un file da creare quando si vuole salvare una partita

class FinestraSoluzioni extends JFrame implements ActionListener

La classe ha lo scopo di definire una finestra che mostri le soluzioni (11 in tutto) relative al sudoku iniziato dall'utente. I metodi da cui è composta sono:

- **FinestraSoluzioni() ->** costruttore che definisce le caratteristiche della finestra e invoca il metodo *risolvi()* del sudoku, il metodo *visibili()* che abilita opportunamente i bottoni next e previous della schermata in base al numero della soluzione visualizzata sulla finestra e il metodo *soluzione(soluzione)* con *soluzione* ancora pari a 0 in modo che venga mostra la soluzione di indice 0 (reso possibile dal meccanismo di salvataggio delle soluzioni in una lista di cui si è parlato sopra).
- **void visibili() ->** metodo che abilita opportunamente i bottoni next e previous della schermata in base al numero indicativo della soluzione in cui si trova
- **void soluzione(int i) ->** metodo che mostra sulla finestra la soluzione di indice i nella lista delle soluzioni calcolate
- **boolean uscita() ->** metodo che viene richiamato quando si prova a chiudere una finestra. Esso visualizza una ulteriore finestra di conferma dell'uscita e, eventualmente, permette di non chiudere realmente la finestra.
- **void ActionPerformed(ActionEvent e) ->** metodo che deriva dall'implementazione di ActionListener. Il metodo viene richiamato ogni volta che si interagisce con un componente della finestra a cui è stato aggiunto un ActionListener attraverso il metodo component.addActionListener(this) e richiama i metodi che svolgono rispettivamente le operazioni richieste tra cui la visualizzazione della soluzione precedente o successiva attraverso l'invocazione del metodo *soluzione(int i)*