Progetto Risolutore Sudoku, studente Marco Greco Matricola 200901

Il mio secondo progetto (risolutore di Sudoku) si basa sull'implementazione di due classi:

- ✓ Sudoku
- ✓ SudokuGui.

La prima è concentrata sulla logica del gioco con un main al fine di test e la seconda sulla grafica.

Classe Sudoku:

//Per semplicità chiamo celle le 81 celle e macrocelle quelle 3x3.

Nella classe Sudoku ho dichiarato una matrice di interi (griglia) la quale sarà la nostra griglia 9x9 nelle cui celle verranno inseriti tutti i numeri da 1 a 9 nel rispetto dei vincoli del gioco. Una matrice di booleani (grigliaInput) utile per tenere conto di quali celle sono quelle preimpostate dall'utente all'inizio e una variabile per il numero della soluzione corrente (numSol) utile per la stampa.

```
ArrayList<int[][]> listaSoluzioni= new ArrayList<>();
int[][] griglia;
boolean[][] grigliaInput;
int numSol=1;
```

Costruttori

La classe presenta due Costruttori: quello di default che inizializza a 0 tutte le celle della griglia 9x9 e a false tutte quelle della matrice di booleani; e quello di impostazioni che riceve come parametro una matrice Nx3 (dove N è il numero delle celle preimpostate) e si avvale del metodo imposta che riceve come parametri la riga i (imp[i][0]) la colonna j (imp[i][1]) e il relativo valore v (imp[i][2]) della cella preimpostata.

```
public Sudoku() {
    griglia=new int[9][9];
    grigliaInput=new boolean[9][9];
}

public Sudoku(int[][] imp) {
    griglia=new int[9][9];
    grigliaInput=new boolean[9][9];
    for(int i=0;i<imp.length;i++) {
        imposta(imp[i][0],imp[i][1],imp[i][2]);
    }
}
```

Metodo Imposta

Il metodo imposta crea una nuova cella con riga e colonna <i,j> passati come parametri, e verifica se rispetta i vincoli del gioco (in tal caso solleva un'eccezione).

Il metodo così assegna (se i parametri sono corretti) alla posizione [i,j] il valore alla griglia e true alla grigliaInput per ricordare che si tratti di una cella preimpostata.

```
public void imposta(int i, int j,int v) {
    Cella cel = new Cella(i,j);
    if(!assegnabile(cel,v)||v<=0||v>9) throw new IllegalArgumentException("Parametri iniziali non validi");
    grigliaInput[cel.rig][cel.col]=true;
    griglia[cel.rig][cel.col]=v;
}
```

Classe Cella

Per eleganza (anche se ci perdo un po' in termini di efficienza) mi sono avvalso di una classe Cella identificata dalla riga i e colonna j. (Durante la stesura della relazione ho pensato che sarebbe stato ancora più elegante creare direttamente anziché una matrice (griglia) di interi una matrice di Celle in cui ogni Cella possedeva il valore v oltre che riga e colonna...)

```
private class Cella{
    int rig,col;

    @SuppressWarnings("unused")
    public Cella() {
        this.rig=0; this.col=0;
    }
    public Cella(int riga,int colonna) {
        if(riga<0||riga>=griglia.length||colonna<0||colonna>=griglia.length) throw new IllegalArgumentException();
        this.rig=riga;this.col=colonna;
    }
}
```

Metodo Colloca

Uno dei metodi più importanti della classe è Colloca che si avvale del compito di collocare per tentativi le possibili scelte (da 1 a 9) in ogni cella del Sudoku; quindi utilizzando la tecnica del backtracking.

```
private void colloca(Cella c) {
    if(numSol==201) return;
    for(int num=1;num<=9;++num) {
        if(!grigliaInput[c.rig][c.col] ) {
            if(assegnabile(c,num)) {
                assegna(c,num);
                if(c.rig==griglia.length-1 && c.col==griglia.length-1) scriviSoluzione();
                else {
                    if(c.col>=griglia.length-1) colloca(new Cella(c.rig+1,0));
                    else colloca(new Cella(c.rig,c.col+1));
                }
                deassegna(c);
            }
        }
    }
    //Non ho messo l'else perchè deve stare fuori dal ciclo
    if(grigliaInput[c.rig][c.col]) {
        if(!(c.rig==griglia.length-1 && c.col==griglia.length-1)) {
            if(c.col==griglia.length-1) colloca(new Cella(c.rig+1,0));
            else colloca(new Cella(c.rig,c.col+1));
        }
        else scriviSoluzione();
    }
}
```

Ho limitato le soluzioni a 200 poiché altrimenti si potrebbe aspettare molto tempo se si passa una matrice di impostazioni molto povera di celle preimpostate. Come punto di scelta ho utilizzato la cella e le possibili scelte non sono altro che i valori da 1 a 9.

```
if(!grigliaInput[c.rig][c.col] ) {
    if(assegnabile(c,num)) {
```

La tecnica del backtracking deve lavorare solo con le celle non preimpostate, pertanto ho inserito la condizione if(!grigliaInput[c.rig][c.col]) che verifica se siamo in presenza di una cella non preimpostata e verificando in tal caso l'assegnabilità del valore in essa.

Se si trattava di una cella preimpostata richiamo il colloca alla cella successiva se non siamo sull'ultima cella della griglia dove in tal caso devo stampare la soluzione. (Non ho messo l'else poiché ciò non può stare nel ciclo!!)

```
    if(grigliaInput[c.rig][c.col]) {
        if(!(c.rig==griglia.length-1 && c.col==griglia.length-1)) {
            if(c.col==griglia.length-1) colloca(new Cella(c.rig+1,0));
            else colloca(new Cella(c.rig,c.col+1));
        }
        else scriviSoluzione();
    }
```

Se il numero è assegnabile nella cella c (rispetta i vincoli) lo assegna chiamando il metodo assegna che imposta alla riga e colonna della cella il numero (valore) e se non siamo arrivati all'ultima cella della griglia richiamo il metodo colloca alla cella successiva, altrimenti scriviamo la soluzione.

```
if(assegnabile(c,num)) {
    assegna(c,num);
    if(c.rig==griglia.length-1 && c.col==griglia.length-1) scriviSoluzione();
    else {
        if(c.col>=griglia.length-1) colloca(new Cella(c.rig+1,0));
        else colloca(new Cella(c.rig,c.col+1));
    }
    deassegna(c);
}
```

Se si sono provati tutti i numeri e si ha sempre ottenuto assegnabile==false allora il metodo colloca ritorna e verrà eseguito deassegna che imposta la cella a 0.

Metodo Assegnabile

Il metodo assegnabile riceve come parametri la cella e il valore da attribuire.

```
private boolean assegnabile(Cella cella,int numero) {
        for(int i=0;i<griglia.length;++i) {
            if(griglia[cella.rig][i]==numero) return false;
        }
        for(int j=0;j<griglia.length;++j) {
            if(griglia[j][cella.col]==numero) return false;
        }

        int x=cella.rig/3*3;
        int y=cella.col/3*3;
        for(int i=x; i<x+3; i++)
            for(int j=y; j<y+3; j++)
                if(griglia[i][j]==numero) return false;

    return true;
}
```

Verifico con il primo for la presenza di uno stesso numero all'interno della stessa riga, con il secondo sulla stessa colonna ed infine con un semplice calcolo di indici se è presente nella sottomatrice 3x3 (macrocella).

```
for(int i=0;i<griglia.length;++i) {
    if(griglia[cella.rig][i]==numero) return false;
}
```

Controllo Riga

```
for(int j=0;j<griglia.length;++j) {
    if(griglia[j][cella.col]==numero) return false;
}
```

Controllo Colonna

```
int x=cella.rig/3*3;
int y=cella.col/3*3;
for(int i=x; i<x+3; i++)
    for(int j=y; j<y+3; j++)
        if(griglia[i][j]==numero) return false;
```

Controllo macrocella 3x3

Il calcolo consiste nel cercare di ottenere sempre lo 0 se ci troviamo in un indice riga/colonna tra 0 e 2, sempre 3 se riga/colonna è tra 3 e 5, ed invece 6 se riga/colonna è tra 6 e 8. I valori 0,3,6 sono gli indici di partenza di ogni riga/colonna della macrocella3x3: <0,0>,<0,3>,<0,6>,<3,0>,<3,3>,<3,6>,<6,0>,<6,3>,<6,6>.

Per esempio se devo verificare la cella <4,5> mi troverei a controllare la macrocella centrale ed in fatti otteniamo x=3 e y=3. I due cicli for scorrono in modo intuitivo fino a x/y +3 (escluso).

Se non si verifica nessuno di questi tre casi il valore è assegnabile e il metodo ritorna true.

Metodo scriviSoluzione

Il metodo scriviSoluzione() oltre che stampare le soluzioni faccio una copia della griglia attuale(la soluzione) e la aggiungo alla listaSoluzioni. Se ci troviamo sugli indici pari a 2 o a 5, sulle colonne procedo a mettere il trattino verticale, mentre nelle righe quelli orizzontali tanti quanto è la lunghezza della griglia*2+3 (i trattini orizzontali sono molto piccoli).

```
void scriviSoluzione() {
    int[][] soluz = new int[9][9];
    System.out.println("    Soluzione "+numSol);
    numSol++;
    for( int i=0; i<griglia.length; ++i ){
        for( int j=0; j<griglia[0].length; ++j ) {
            soluz[i][j]=griglia[i][j];
            System.out.print(griglia[i][j]+" ");
            if(j==2 || j==5) System.out.print("| ");
        }
        System.out.println();
        if(i==2 || i==5) {
            for(int k=0;k<griglia.length*2+3;++k) System.out.print("-");
            System.out.println();
        }
    }
    listaSoluzioni.add(soluz);
    System.out.println();

}
```

Esempio di output di un Sudoku con due soluzioni:

```
    Soluzione 1
9 4 1 | 5 7 2 | 8 6 3
6 3 8 | 1 4 9 | 5 2 7
2 5 7 | 8 3 6 | 1 9 4
---------------------
1 7 6 | 4 9 3 | 2 8 5
4 9 3 | 2 8 5 | 6 7 1
8 2 5 | 6 1 7 | 4 3 9
---------------------
7 1 2 | 9 5 8 | 3 4 6
3 8 4 | 7 6 1 | 9 5 2
5 6 9 | 3 2 4 | 7 1 8
```

```
    Soluzione 2
9 4 1 | 5 7 2 | 8 6 3
6 3 8 | 1 4 9 | 5 2 7
2 7 5 | 8 3 6 | 1 9 4
---------------------
1 6 7 | 4 9 3 | 2 8 5
4 9 3 | 2 8 5 | 6 7 1
8 5 2 | 6 1 7 | 4 3 9
---------------------
7 1 4 | 9 2 8 | 3 5 6
3 8 6 | 7 5 1 | 9 4 2
5 2 9 | 3 6 4 | 7 1 8
```

Classe SudokuGui:

La classe SudokuGui possiede un main che si limita a creare una nuova finestra dell'applicazione e renderla visibile.

```
public static void main(String[] args) {
    JFrame fin = new FinestraApp();
    fin.setVisible(true);
}
```

Classe FinestraApp

La classe relativa alla finestra estende JFrame, nella quale dichiaro un JTextField sol che corrisponde al riquadro nero dove segnala all'utente quale soluzione sta visionando; i bottoni risolvi,previous,reset,next,salva,carica, due label s e c per il testo salva e carica, il numero della soluzione attuale (soluzione) e un oggetto di tipo Sudoku.

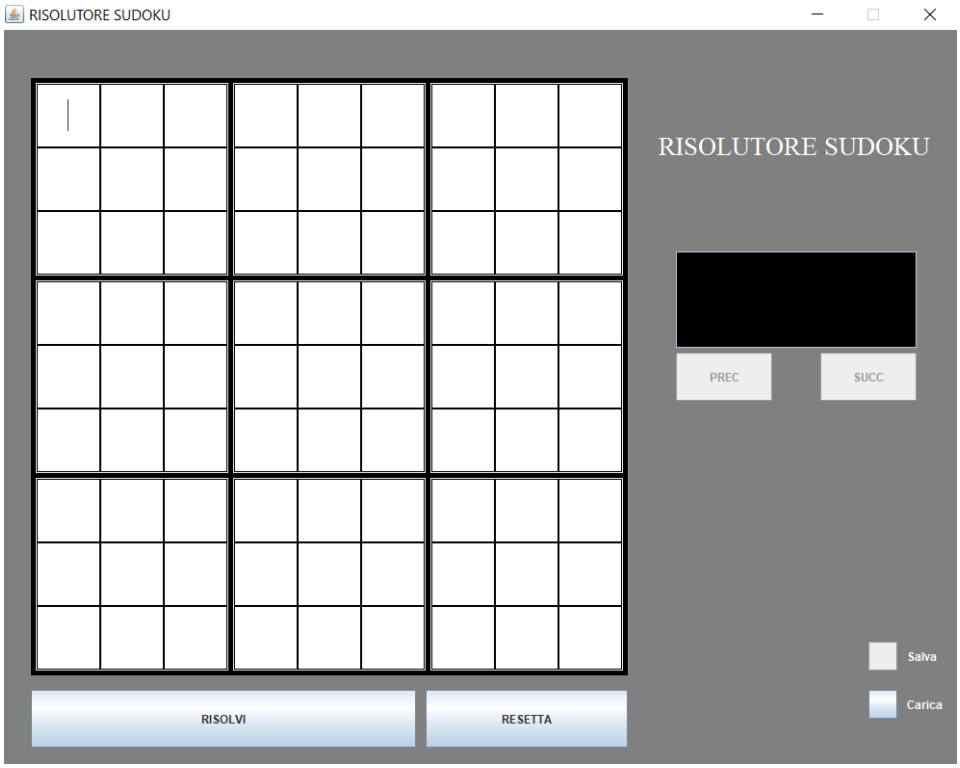
```
class FinestraApp extends JFrame {
    JTextField[][][][] matriceSu = new JTextField[3][3][][];
    JTextField sol;
    JLabel s,c;
    JButton risolvi,previous,reset,next,salva,carica;
    ActionListener listener = new AscoltatoreEventi();
    int soluzione=0;
    private Sudoku sud;
```

I JTextFields delle 81 celle li raggruppo in 9 blocchi di matrici 3x3 (macrocelle) i quali li salvo nella matrice complessiva che raffigurerà la nostra griglia Sudoku (una matrice di matrici). Quindi in poche parole creo una matrice multidimensionale (matriceSu) 3x3 che conterrà le 9 macrocelle 3x3. Ciò mi è stato molto utile nella rappresentazione grafica del Sudoku.

Il tutto è accompagnato dalla creazione di un ascoltatore (listener).

Costruttori

La classe FinestraApp contiene solo il costruttore di default. Al suo interno setto il titolo della finestra, la sua dimensione, la posizione e una volta creato gli aggiungo un pannello principale (pannello) con lo scopo di contenere i vari componenti: risolvi,previous,reset,next,salva,carica, title,sol e la griglia del sudoku. Si preoccupa quindi di creare tutto ciò che l'utente visiona all'apertura dell'applicazione.




```
carica = new JButton("");
carica.setBounds(900, 685, 30, 30);
carica.addActionListener(listener);
pannello.add(carica);

risolvi = new JButton("RISOLVI");
risolvi.setBounds(30, 685, 400, 60);
risolvi.addActionListener(listener);
pannello.add(risolvi);

previous = new JButton("PREC");
previous.setEnabled(false);
previous.setBounds(700, 335, 100, 50);
previous.addActionListener(listener);
pannello.add(previous);

reset = new JButton("RESETTA");
reset.setBounds(440, 685, 210, 60);
reset.addActionListener(listener);
pannello.add(reset);

next = new JButton("SUCC");
next.setEnabled(false);
next.setBounds(850, 335, 100, 50);
next.addActionListener(listener);
pannello.add(next);
```

Il layout del pannello pannello lo setto a null, garantendomi di settare le dimensioni e posizioni (bounds) a mio piacimento dei vari componenti ai quali aggiungo successivamente l'ascoltatore listener.

```
JPanel pannello = new JPanel();
add(pannello);
pannello.setBackground(Color.GRAY);
pannello.setLayout(null);
```

Al pannello pannello aggiungo il pannello griglia con background di colorazione nera che farà da base alle 9 macrocelle3x3 del Sudoku (layout null per lo stesso motivo).

```
JPanel griglia = new JPanel();
griglia.setLayout(null);
griglia.setBackground(Color.BLACK);
griglia.setBounds(30,50, 620, 620);
pannello.add(griglia);
```

All’interno del pannello griglia ho deciso di inserire 9 pannelli (per le macrocelle 3x3) con un layout GridLayout dove al suo interno ad ogni giro del for una volta creato aggiungo un jTextField che rappresenterà una cella. Ad ogni JTextField setto l’allineamento del testo Centrale, un font più grande e un bordo nero che evidenzia gli spazi sottili tra ogni cella.

1								
				9			4	
	2							
				5		7		
							8	
	3			4	6			
9			3	6		8		
				1				
	1							7

Il primo for scorre le righe i della griglia (matriceSu), il secondo le colonne j della griglia (matriceSu), il terzo le righe m della macrocella e il quarto le colonne n della macrocella.

La matrice 3x3 rappresenta la nostra macrocella alla quale verranno assegnati tutti i nove JTextField ed essa verrà assegnata a sua volta alla posizione i,j della griglia matriceSu.

La x e la y sono rispettivamente le coordinate della posizione del pannello della macrocella. La x aumenta di 205 ogni scorrimento di colonna ritornando a zero ad ogni scorrimento di riga e la y non può solo che aumentare ad ogni scorrimento di riga.

```
int y=5;
for(int i=0;i<3;++i) {
    int x = 5;
    if(i!=0) y+=205;
    for(int j=0;j<3;++j) {
        JPanel Panel3x3 = new JPanel(new GridLayout(3,3));
        JTextField[][] matrice3x3 = new JTextField[3][3];
        for(int m=0;m<3;++m)
            for(int n=0;n<3;++n) {
                JTextField jt = new JTextField();
                jt.setHorizontalAlignment(SwingConstants.CENTER);
                jt.setFont(font);
                jt.setBorder(BorderFactory.createLineBorder(Color.black));
                matrice3x3[m][n]=jt;
                matriceSu[i][j]=matrice3x3;
                Panel3x3.add(jt);
            }
        Panel3x3.setBounds(x, y, 200, 200);
        x+=205;
        griglia.add(Panel3x3);
    }
}
```


Metodi stampaSoluzione e contaCellePreimpostate

In seguito ho effettuato l'implementazione di due metodi privati:

stampaSoluzione che passatogli un oggetto di tipo Sudoku e un intero che corrisponde al numero della soluzione setta il testo di ogni cella con il valore corrispondente prestando attenzione a quale soluzione si tratta. Per fare ciò utilizzo il metodo setText sulla cella di JTextField passandogli la stringa (Integer.toString) dell'intero presente nella cella in posizione [c+3*i][m+3*j] della matrice in posizione numeroSoluzione all'interno della listaSoluzioni.

```
private void stampaSoluzione(Sudoku sudoku,int numeroSoluzione) {
    for(int i=0;i<matriceSu.length;i++)
        for(int c=0;c<matriceSu[0].length;++c)
            for(int j=0;j<matriceSu[0][0].length;++j)
                for(int m=0;m<matriceSu[0][0][0].length;++m)
                    matriceSu[i][j][c][m].setText(Integer.toString(sudoku.listaSoluzioni.get(numeroSoluzione)[c+3*i][m+3*j]));
}
```

contaCellePreimpostate() con lo scopo di contare il numero di celle preimpostate in modo da poter fornire la grandezza delle righe della matrice imposta successivamente creata. Il metodo è banale poiché vi è solo un contatore che aumenta una volta incontrata una cella in cui il contenuto del JTextField non è vuoto (c'è un valore).

```
private int contaCellePreimpostate() {
    int N=0;
    for(int i=0;i<matriceSu.length;i++)
        for(int c=0;c<matriceSu[0].length;++c)
            for(int j=0;j<matriceSu[0][0].length;++j)
                for(int m=0;m<matriceSu[0][0][0].length;++m)
                    if(!(matriceSu[i][j][c][m].getText().isEmpty()))
                        N++;
    return N;
}
```

Classe AscoltatoreEventi

Ho creato la inner class AscoltatoreEventi ed ho implementato l'unico metodo dell'interfaccia ActionListener implementata dalla classe.

```
private class AscoltatoreEventi implements ActionListener{
    public void actionPerformed(ActionEvent e) {
```

- **Pressione Risolvi**

Se abbiamo premuto il bottone **risolvi**, si occupa di creare la matrice di impostazioni e di verificare la validità di ogni contenuto di un JTextField pronto a lanciare un'eccezione. Azzera soluzione (in caso di una nuova pressione futura), abilita i bottoni next e previous e inizializza la matrice imposta con tante righe quanto ritorna il metodo contaCellePreimpostate().

Ho creato un'etichetta "ciclo" in modo che se durante l'assegnamento del valore nella matrice di impostazione dovesse andare storto qualcosa (Se mi passa un formato sbagliato, come ad esempio una stringa=>NumberFormatException) fermo tutto il ciclo e al try e catch successivo mostrerà il messaggio di dialogo "Parametri Iniziali Errati !". Non ho fatto uscire al primo catch il messaggio di dialogo poiché altrimenti essendo catturata nuovamente l'eccezione, il messaggio comparirebbe due volte. Quindi se tutto fila liscio crea un nuovo Sudoku invocando il costruttore di impostazioni e lo assegna a sud e invoca sullo stesso il metodo risolvi() della classe Sudoku. Stampa la prima soluzione, mostra la soluzione corrente e abilita il bottone next se vi è più di una soluzione. Il controllo dell'eccezione basato sul non rispetto dei vincoli è gestito nell'ultimo try-catch.

```
if(e.getSource()==risolvi) {
    soluzione=0;
    next.setEnabled(false);
    previous.setEnabled(false);
    int[][] imposta= new int[contaCellePreimpostate()][3];
    int count=0;
    ciclo: for(int i=0;i<matriceSu.length;i++) {
        for(int c=0;c<matriceSu[0].length;++c) {
            for(int j=0;j<matriceSu[0][0].length;++j) {
                for(int m=0;m<matriceSu[0][0][0].length;++m) {
                    if(!(matriceSu[i][j][c][m].getText().isEmpty())) {
                        imposta[count][0]=c+3*i;
                        imposta[count][1]=m+3*j;
                        try {
                            imposta[count][2]=Integer.parseInt(matriceSu[i][j][c][m].getText());
                        }catch(NumberFormatException exc) {break ciclo;} //il messaggio di avviso lo gestisco alla riga 190
                        count++;
                    }
                }
            }
        }
    }
    try {
        sud = new Sudoku(imposta);
        sud.risolvi();
        stampaSoluzione(sud,0);
        sol.setText("SOLUZIONE "+(soluzione+1)+" DI "+(sud.numSol-1));
        if(sud.numSol-1!=1)next.setEnabled(true);
        salva.setEnabled(true);
    }catch(IllegalArgumentException ex){JOptionPane.showMessageDialog(null, "Parametri Iniziali Errati !");}
}   //Cattura anche l'eccezione in cui gli passo una stringa.
```

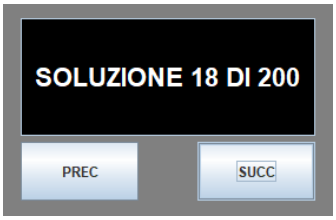
- **Pressione Previous e Next**

Se si è premuto il bottone previous se la soluzione precedente è la prima allora disabilito il bottone previous in modo da non permettere all'utente di accedere ad una soluzione negativa causando il sollevamento di un'eccezione. Successivamente in qualunque caso ci troviamo diminuisco soluzione, stampo la soluzione corrente, aggiorno il testo della soluzione corrente e abilito il bottone next nel caso l'utente avesse visionato prima tutte le soluzioni.

Una volta capito cosa succede se si preme il bottone previous è intuitivo capire la versione riguardo la pressione del bottone next: disabilitando il bottone next se la soluzione successiva è l'ultima (numSol-2 perché numSol parte da 1 ed è il numero delle soluzioni totali).

```
if(e.getSource()==previous) {
    if(soluzione-1<1) previous.setEnabled(false);
    soluzione--;
    stampaSoluzione(sud,soluzione);
    sol.setText("SOLUZIONE "+(soluzione+1)+" DI "+(sud.numSol-1));
    next.setEnabled(true);
}

if(e.getSource()==next) {
    if(soluzione+1>=sud.numSol-2) next.setEnabled(false);
    soluzione++;
    stampaSoluzione(sud,soluzione);
    sol.setText("SOLUZIONE "+(soluzione+1)+" DI "+(sud.numSol-1));
    previous.setEnabled(true);
}
```



- **Pressione Resetta**

Alla pressione del bottone reset disabilita i bottoni next e previous, pulisce il testo della soluzione corrente, disabilita il bottone salva e imposta soluzione corrente (soluzione) a 0.

```
if(e.getSource()==reset) {
    for(int i=0;i<matriceSu.length;i++)
        for(int c=0;c<matriceSu[0].length;++c)
            for(int j=0;j<matriceSu[0][0].length;++j)
                for(int m=0;m<matriceSu[0][0][0].length;++m)
                    matriceSu[i][j][c][m].setText("");
    next.setEnabled(false);
    previous.setEnabled(false);
    soluzione=0;
    sol.setText("");
    salva.setEnabled(false);
}
```

- **Pressione Salva e Carica**

Come ultimo controllo gestisco il salvataggio e il caricamento su/da file cercando di mantenere consistenza con la gestione salvataggio/caricamento del progetto sui polinomi.

Indipendentemente da quale dei due bottoni premo creo un nuovo jFileChooser jfc al quale gli assegno un FileFilter consentendo di salvare/caricare solo file in formato txt (per semplicità, in generale file testo).

```
String nomefile=null;
int valore;
FileNameExtensionFilter filtro = new FileNameExtensionFilter("File TXT", "txt");
JFileChooser jfc = new JFileChooser();
jfc.setFileFilter(filtro);
if(e.getSource()==salva) valore = jfc.showSaveDialog(null);
else valore = jfc.showOpenDialog(null);
if(valore==JFileChooser.APPROVE_OPTION) {
    nomefile=jfc.getSelectedFile().getAbsolutePath();
```

Se stiamo salvando, racchiuso in un try-catch (può sorgere un'eccezione del tipo IOException se ad esempio un file è corrotto) creo un PrintWriter (Bufferizzato) al quale gli passo il Path del nomefile selezionato con il JFileChooser. Considerando solo le celle preimpostate, controllando quindi che in quella posizione nella grigliaInput risulti esserci true, scrivo sul file ordinatamente prima la priga, poi la colonna e poi il contenuto (valore).

```
if(e.getSource()==salva) {
    PrintWriter pw;
    try {
        pw = new PrintWriter(new BufferedWriter(new FileWriter(nomefile)));
        for(int i=0;i<sud.griglia.length;++i) {
            for(int j=0;j<sud.griglia[0].length;++j) {
                if(sud.grigliaInput[i][j]==true) {
                    pw.println(i);
                    pw.println(j);
                    pw.println(Integer.toString(sud.griglia[i][j]));
                }
            }
        }
        pw.close();
        JOptionPane.showMessageDialog(null,"Hai salvato il tuo operato sul file: " + nomefile);
    } catch (IOException e1) {
        JOptionPane.showMessageDialog(null,"Qualcosa è andato storto..");
        e1.printStackTrace();
    }
}
```

Se stiamo in fase di caricamento è opportuno dapprima resettare il tutto invocando il metodo doClick() sul bottone reset per poi creare un BufferedReader e leggere quindi da file. Ho utilizzato un for infinito e leggo la linee di testo a tre a tre in cui la prima ci da il numero di riga, la seconda il numero di colonna e la terza il valore. La fine di un file su un BufferedReader lo si individua quando la linea letta è null, procedendo in tal caso al break del ciclo infinito.

```
else {
    reset.doClick();
    try {
        BufferedReader br = new BufferedReader(new FileReader(nomefile));
        String linea=null;
        cicloinf:for(;;) {
            int riga=0;
            int colonna=0;
            for(int i=0;i<3;i++) {
                linea=br.readLine();
                if(linea==null) break cicloinf;
                switch(i) {
                case 0: riga=Integer.parseInt(linea);break;
                case 1: colonna=Integer.parseInt(linea);break;
                case 2: matriceSu[riga/3][colonna/3][riga-riga/3*3][colonna-colonna/3*3].setText(linea);break;
                }
            }
        }
        br.close();
    } catch (IOException e1) {
        JOptionPane.showMessageDialog(null,"Qualcosa è andato storto.. Il file potrebbe essere corrotto");
        e1.printStackTrace();
    }
}
```

In base a quale delle tre righe ci troviamo ho realizzato uno switch dove nei casi 0 e 1 mi salvo riga e colonna e nella terza linea (indice 2) procedo direttamente a settare il valore sulla cella corrispondente presente nella matrice multidimensionale matriceSu di JTextField. Ricordo che riga e colonna vanno da 0 a 8 e in matriceSu abbiamo le macrocelle3x3 con indici che vanno quindi sempre da 0 a 2; pertanto per identificare la posizione della cella nella macrocella corrispondente ho dovuto sistemare gli indici con qualche calcolo intuitivo. Sempre chiudere i relativi PrintWriter e BufferedReader!