

7. SISTEMUL DE EVENIMENTE. COMPONENTE SWING COMPLEXE

7.1. Tratarea interacțiunii utilizatorului cu interfața grafică.....	<u>2</u>
7.1.1. Principii: sistemul de evenimente.....	<u>2</u>
7.1.2. Tipuri de evenimente și tratarea lor.....	<u>3</u>
7.1.3. Un exemplu.....	<u>5</u>
7.1.4. Clase de adaptare.....	<u>6</u>
7.1.5. Evenimentele și firele de execuție Swing.....	<u>6</u>
7.2. Clase interioare și utilitatea lor în tratarea evenimentelor.....	<u>7</u>
7.2.1. Concepte.....	<u>7</u>
7.2.2. Clase interioare anonime (anonymous inner classes).....	<u>8</u>
7.3. Lucrul cu componente complexe pentru afișare/editare date.....	<u>8</u>
7.3.1. Arhitectura componentelor Swing.....	<u>8</u>
7.3.2. Afișarea și editarea informației într-o componentă complexă.....	<u>10</u>
7.3.3. Modalități de a crea un model de date pentru o componentă complexă.....	<u>10</u>
7.3.4. Utilizarea componentelor de tip JList.....	<u>11</u>
7.3.4.1. Capabilități. Clase și metode aferente.....	<u>11</u>
7.3.4.2. Evenimente specifice și tratarea acestora.....	<u>12</u>
7.3.4.3. Definirea modelului pentru date.....	<u>13</u>
7.3.4.3.1. Principii de funcționare.....	<u>13</u>
7.3.4.3.2. Variante de obținere a unui model.....	<u>13</u>
7.3.4.3.3. Modelul ca array de Object.....	<u>14</u>
7.3.4.3.4. Modelul ca DefaultListModel.....	<u>14</u>
7.3.4.3.5. Modelul ca instanță a unei clase care extinde AbstractListModel.....	<u>14</u>
7.3.4.4. Definirea unui Renderer particularizat.....	<u>15</u>
7.3.5. Utilizarea componentelor de tip JComboBox.....	<u>16</u>
7.3.5.1. Capabilități și principii de funcționare.....	<u>16</u>
7.3.5.2. Evenimente specifice.....	<u>16</u>
7.3.5.3. Crearea modelului de date.....	<u>17</u>
7.3.6. Utilizarea componentelor de tip JSpinner.....	<u>18</u>
7.3.6.1. Evenimente specifice.....	<u>18</u>
7.3.6.2. Modele și editoarele atașate.....	<u>18</u>
7.3.7. Utilizarea componentelor de tip JTable.....	<u>19</u>
7.3.7.1. Caracteristici și principii de funcționare.....	<u>19</u>
7.3.7.2. Evenimente specifice.....	<u>20</u>
7.3.8. Crearea modelului de date.....	<u>21</u>
7.4. BIBLIOGRAFIE.....	<u>22</u>
7.5. ANEXA 1 - clase model pentru componentele Swing complexe.....	<u>22</u>
7.6. ANEXA 2 - Exemple de cod aditionale.....	<u>22</u>
7.6.1. Model de JComboBox personalizat.....	<u>22</u>
7.6.2. Model de JTable personalizat.....	<u>23</u>

7.1. Tratarea interactiunii utilizatorului cu interfata grafica

7.1.1. Principii: sistemul de evenimente

Analizand arhitectura Model-View-Controller constatam ca, folosind cunoștințele acumulate pana în acest punct, putem scrie modelul și view-ul. În continuare studiem modul în care le conectăm pe acestea două pentru a adăuga funcționalitate interfelei grafice create.

La interacțiunea cu utilizatorul, fiecare componentă grafică poate genera unul sau mai multe **evenimente**. Evenimentul este un obiect ce încapsulează informație despre acțiunea utilizatorului: spre exemplu, atunci când utilizatorul apasă o tastă, este memorat în obiectul eveniment codul tastei respective; în cazul click-urilor de mouse, se memorează numărul de click-uri (click simplu/dublu/triplu...) și poziția cursorului în cadrul componentei etc. Obiectul eveniment este creat de către componentă cu care utilizatorul a interacționat și pasat unuia sau mai multor obiecte externe care au capacitatea de a reacționa la acel eveniment prin executarea de cod (asa-numitele obiecte *handler* sau *listener*). Prin crearea obiectelor listener și atașarea lor la sursele de evenimente, programatorul definește reacția interfeței grafice la interacțiunea cu utilizatorul.

Distingem astăzi trei obiecte implicate în apariția și tratarea unui eveniment:

- **sursa evenimentului** - este componenta din GUI asupra careia a acționat utilizatorul; ca urmare a acțiunii acestuia, ea creează obiectul eveniment și îl păsează obiectelor listener
- **obiectul eveniment** - este cel care memorează natura și detaliile interacțiunii user-componentă ce a dus la apariția sa
- **obiectul listener** - este cel care tratează evenimentul; el recepționează obiectul eveniment și folosește informația continuta în acesta ca date de intrare ce parametrizează reacția la eveniment



Scopul programatorului este să poată executa cod cu ocazia apariției unui eveniment. Acest cod nu putea pre-exista în clasa componentei sursă, deoarece creatorii componentelor grafice nu puteau prevedea care vor fi necesitătile de moment ale programatorului; și chiar dacă ar fi putut, de la un programator la altul și de la o situație la alta nevoile sunt suficiente de diverse încât să facă imposibilă includerea codului reacție în insași componentă. Aceasta este motivul pentru care componenta grafică „colaborează” cu obiecte externe, create de programator, care au capacitatea de a reacționa la evenimentele sale.

Componenta ce constituie sursa evenimentului nu cunoaște a priori niciun fel de listener; ea menține însă o listă de referințe către obiecte listener (initial vida), disponând de metode pentru adăugarea sau eliminarea lor din lista. Este responsabilitatea programatorului să se asigure că a înregistrat listenerii potriviti în lista atașată componentei. La interacțiunea cu utilizatorul, componenta instantiază obiectul eveniment și parcurge lista de listeneri, apelandu-i fiecaruia o anumita metodă ce primește ca argument obiectul eveniment. Metoda conține codul-reacție la evenimentul în cauză, scris de către programator.

Obiectele listener sunt create de către programator, acesta având libertate maximă în conceperea lor. Pentru a avea însă capacitatea de a trata evenimente, ele trebuie să dispună de un set de metode - cele care le sunt apelate de către sursa evenimentului. Pentru a oferi componentei sursă garanția existenței acestor metode, clasa obiectelor listener este obligată să implementeze o anumita interfață ce le impune metodele necesare.

Pentru exemplificare, fie cazul concret al unui buton care cauzează ieșirea din aplicație. Pentru a trata apăsarea pe el sunt necesare următoarele masuri:

- la apăsare, butonul generează un eveniment de tip *ActionEvent*. Listenerii pentru acest tip de eveniment trebuie să implementeze interfața *ActionListener*, ce impune o singura metodă: *void actionPerformed(ActionEvent e)*. În consecință, cream o clasa (sa o numim *ExitHandler*) care implementează interfața și scriem metoda corespunzătoare

- pentru a executa codul metodei `actionPerformed()` la apasarea pe buton, este necesara inregistrarea unei instante a clasei `ExitHandler` ca listener pentru buton. Acest lucru se realizeaza prin adaugarea instantei in lista de listeneri a butonului, folosind metoda `addActionlistener()` a acestuia, ca in exemplul de mai jos



```

// clasa listener
class ExitHandler implements ActionListener{
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}
// ...iar in clasa ce creeaza interfata grafica se creeaza butonul si i se ataseaza listenerul:
JButton b = new JButton("Exit");
b.addActionListener(new ExitHandler());
  
```

Acest mod de organizare a sistemului de evenimente urmeaza de fapt un cunoscut design pattern numit *Observer*. Solutia ofera avantaje precum:

- sursa evenimentului nu este obligata sa cunoasca mai mult decat trebuie despre obiectele listener; in acest fel ea poate fi scrisa independent de codul care ii trateaza evenimentele
- obiectul de tip listener poate fi scris in orice mod doreste programatorul (asadar mentionand un grad mare de independenta fata de sursa) cat timp implementeaza interfata corespunzatoare

7.1.2. Tipuri de evenimente si tratarea lor

O componenta poate genera diverse tipuri de evenimente, in functie de natura interactiunii utilizatorului cu ea. Spre exemplu, un buton poate genera evenimente la apasare, dar si la intrarea sau iesirea cursorului din suprafata sa sau la apasarea unei combinatii de taste cand butonul are focus; in plus, o componenta `JList` genereaza evenimente atunci cand se modifica lista de elemente selectate, un `JTextField` genereaza evenimente atunci cand utilizatorul scrie sau selecteaza text etc. Fiecaruia dintre aceste evenimente ii corespunde un alt tip de obiect eveniment.

Rolul obiectului eveniment este acela de a cărăuș de informatie de la sursa evenimentului catre obiectul care il va trata; de aceea evenimentele dispun de metode pentru extragerea acestei informatii. Setul de metode disponibil depinde de natura evenimentului. Toate clasele eveniment sunt derive din `java.awt.EventObject`, care transmite subclaszelor urmatoarele doua metode:

- Object getSource()** - intoarce o referinta catre componenta ce a generat evenimentul
- String toString()** - intoarce un text ce descrie evenimentul

Vom imparti evenimentele posibile in doua categorii:

- evenimente generale** - cele care pot fi generate de orice componenta. Sunt de obicei evenimente elementare, de tip apasari de taste, actiuni ale mouse-ului etc
- evenimente specific** - sunt cele care tin de specificul componentei; spre exemplu, un buton poate fi apasat, un `JTable` permite selectarea unuia sau mai multor randuri sau celule, un `JTextField` permite editarea si selectarea de text etc.

Iata principalele evenimente generale suportate de componentele grafice:

Clasa eveniment	Generat la	Metode de interes ale obiectului eveniment	Interfata ce trebuie implementata de catre listener	Metode impuse de interfata listener	Clasa de adaptare
KeyEvent	actionarea tastelor cand focusul este pe componenta	getKeyChar() getKeyCode()	KeyListener	keyPressed() keyReleased() keyTyped()	KeyAdapter
MouseEvent	apasare sau eliberare a	getButton()	MouseListener	mouseClicked()	MouseAdapter

	unui buton de mouse; click; intrarea sau iesirea cursorului din suprafata unei componente	getClickCount() getPoint()		mousePressed() mouseReleased() mouseEntered() mouseExited()	
MouseEvent	miscarea cursorului in suprafata componentei sau drag'n'drop	idem mai sus	MouseMotionListener	mouseDragged() mouseMoved()	-
MouseWheelEvent	actionarea rotitei mouse-ului	getScrollAmount() getWheelRotation()	MouseWheel Listener	mouseWheelMoved()	-
FocusEvent	primirea sau pierderea focusului	getOppositeComponent() isTemporary()	FocusListener	focusGained() focusLost()	-

Toate metodele listenerilor de mai sus primesc ca parametru un eveniment de tipul corespunzator (de exemplu, *mouseClicked()* primește un parametru de tip *MouseEvent*).

Iată în continuare principalele evenimente specifice ale diverselor componente grafice discutate până acum, împreună cu interfetele ce trebuie implementate de către obiectele listener și metodele corespunzătoare:

Componentă	Clasa eveniment	Generat la	Metode utile ale obiectului eveniment (în plus față de <i>EventObject</i>)	Interfața ce trebuie implementată de către listener	Metode impuse de interfața listener	Clase de adaptare
Window și derivatele (JFrame, JDialog etc)	WindowEvent	acțiuni aplicate unei ferestre (inchidere, minimizare, maximizare, focus etc)	getOldState() getNewState()	WindowListener	windowClosed() windowClosing() windowIconified() windowDeiconified() windowActivated() windowDeactivated() windowOpened()	Window Adapter
AbstractButton și derivatele; JComboBox; JTextField	ActionEvent	efectuarea acțiunii proprii componentei în cauză (apasare pe buton; schimbarea selectiei în combo; Enter în text field etc)	getActionCommand() getModifiers() getWhen()	ActionListener	actionPerformed()	-
JTextComponent și derivatele sale	CaretEvent	modificarea pozitiei cursorului de editare	getDot() getMark()	CaretListener	caretUpdate()	-
AbstractButton și derivatele sale; JSlider; JSpinner	ChangeEvent	schimbarea valorii curente	-	ChangeListener	stateChanged()	-
AbstractButton și derivatele; JComboBox	ItemEvent	schimbarea stării de selecție a unui element (selectare sau deselectare)	getItem() getStateChange()	ItemListener	itemStateChanged()	-
Document-ul atașat unui JTextComponent	DocumentEvent	modificarea textului componentei	getLength() getOffset() getType()	DocumentListener	insertUpdate() removeUpdate() changedUpdate()	-
JList, JTable (mai exact modelul lor de date)	ListSelectionEvent	modificarea setului de valori selectate	getFirstIndex() getLastIndex()	ListSelectionListener	valueChanged()	-

Nota: evenimentele pentru componente complexe sunt detaliate în secțiunea dedicată fiecărei componente din cadrul acestui material.

O componentă poate genera mai multe evenimente ca urmare a unei singure interacțiuni cu userul! Spre exemplu, la click pe un buton se generează evenimente (și se apelează metode) după cum urmează:

- pentru lista sa de listeneri de tip *MouseListener*:
 - *mousePressed()* - la apasarea butonului de mouse
 - *mouseReleased()* - la eliberarea butonului de mouse
 - *mouseClicked()* - deoarece precedentele două combinate formează un click
- pentru lista sa de listeneri de tip *ActionListener*
 - *actionPerformed()* - corespunzător acțiunii specifice a componentei (apasarea pe buton)

Distingem in cadrul enumerarilor de mai sus doua tipuri de evenimente:

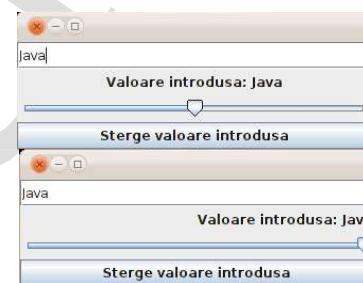
- **evenimente elementare (low-level)** - sunt cele mai “de jos” evenimente posibile, corespunzatoare interactiunii elementare a utilizatorului cu interfata grafica. Se refera in general la actiuni de mouse, apasari de taste sau evenimente legate de sistemul de fereste
- **evenimente de nivel inalt (semantice)** - sunt generate ca urmare a uneia sau mai multor succesiuni de evenimente elementare care duc la o anumita actiune de nivel inalt; spre exemplu, pentru a da click pe un buton putem proceda in mai multe moduri: folosind mnemonic-ul butonului (asadar tratand un *KeyEvent*), apasand SPACE cand butonul are focus (tot *KeyEvent*), dand click pe buton (*MouseEvent*) etc. Daca nu ar exista evenimente semantice, programatorul ar fi nevoie sa trateze toate aceste cazuri pentru a “prinde” apasarea butonului; din fericire, butonul genereaza un *ActionEvent* la apasarea sa (eveniment semantic), indiferent de calea pe care s-a ajuns la efectul de apasare

Programatorul este incurajat sa foloseasca cat mai mult posibil evenimente semantice. Aceasta deoarece, pe de o parte, combinatia de evenimente simple care duce la evenimentul semantic depinde de look and feel, si in plus ea poate diferi de la o folosire a componentei la alta (vezi exemplul butonului de mai sus, care poate fi apasat in mai multe moduri de catre acelasi utilizator in cadrul aceliasi rulari a aplicatiei).

7.1.3. Un exemplu

Iata un exemplu de tratare de evenimente pentru componente de diverse tipuri. In fereastra din figura alaturata, componentele reagioneaza astfel:

- cand utilizatorul apasa Enter in text field, valoarea este preluata si afisata in label-ul de dedesupt
- la apasarea pe buton va fi sters continutul text field-ului si al label-ului
- slider-ul are 3 pozitii; fiecare dintre ele corespunde vizual cate unei alinieri posibile a label-ului, astfel incat textul label-ului se muta odata cu cursorul sliderului (comparati figurile alaturate)
- cand cursorul mouse-ului trece pe deasupra label-ului, textul acestuia se coloreaza in rosu, revenind la culoarea initiala cand cursorul paraseste suprafata componentei. La click pe label, cat timp butonul de mouse este apasat textul se afiseaza alb pe negru, revenind la culorile initiale la eliberarea butonului



Vom crea o clasa care extinde *JFrame*, definind toate componentele pe post de campuri ale acesteia. Insusi obiectul derivat din *JFrame* va juca rolul de listener pentru evenimente de tip *ActionEvent* (generate de text field si de buton) si de tip *ChangeEvent* (generate de slider). Pentru evenimentele de mouse folosim o clasa separata,

```
public class ExempluEventuri extends JFrame implements ActionListener, ChangeListener{
    JButton b = new JButton("Sterge valoare introdusa");
    JTextField tf = new JTextField(30); // 30 de caractere
    JLabel l = new JLabel("", JLabel.CENTER);
    JSlider s = new JSlider(JSlider.HORIZONTAL, 0, 2, 1); // minim 0, maxim 2, implicit 1 (central)

    public ExempluEventuri(){
        b.addActionListener(this); tf.addActionListener(this); s.addChangeListener(this);
        l.addMouseListener(new HandlerMouseLabel(l)); l.setOpaque(true);
        getContentPane().setLayout(new GridLayout(4,1));
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(tf); add(l); add(s); add(b); pack();
    }
    public void actionPerformed(ActionEvent evt) {
        // in functie de componenta ce constituie sursa evenimentului executam cod diferit
        if(evt.getSource() == b){ tf.setText(""); l.setText(""); }
        else if(evt.getSource() == tf){ l.setText("Valoare introdusa: "+tf.getText()); }
    }
    public void stateChanged(ChangeEvent arg0) {
        switch(s.getValue()){
            case 0: l.setHorizontalAlignment(JLabel.LEFT);break;
            case 1: l.setHorizontalAlignment(JLabel.CENTER);break;
            case 2: l.setHorizontalAlignment(JLabel.RIGHT);break;
        }
    }
}
```

```

public static void main(String[] args) { new ExempluEventuri().setVisible(true); }

}

class HandlerMouseLabel implements MouseListener{
    JLabel l; // label-ul cu care "colaboreaza" listenerul, setat prin constructor
    Color baseFg, baseBg; // culorile initiale ale label-ului; de refacut la iesirea cursorului
    Color oldFg, oldBg; // culorile anterioare, folosite de click pe label
    public HandlerMouseLabel(JLabel x){l = x;baseFg = l.getForeground();baseBg=l.getBackground();}
    public void mouseEntered(MouseEvent e) {
        l.setForeground(Color.red);
    }
    public void mouseExited(MouseEvent e) { l.setForeground(baseFg); l.setBackground(baseBg); }
    public void mousePressed(MouseEvent e) {
        oldFg = l.getForeground(); oldBg = l.getBackground(); // memorare culori vechi
        l.setBackground(Color.black);l.setForeground(Color.white); // setare culori temporare
    }
    public void mouseReleased(MouseEvent e) { l.setBackground(oldBg);l.setForeground(oldFg); }
    public void mouseClicked(MouseEvent e) {}
}

```

7.1.4. Clase de adaptare

Dupa cum s-a observat anterior, unele componente genereaza acelasi tip de eveniment in urma mai multor actiuni posibile asupra componentei (de exemplu *JFrame*, care genereaza un *WindowEvent* pentru inchidere, minimizare, maximizare etc, sau *MouseEvent* pentru apasare/eliberare/click de buton etc). In astfel de cazuri interfata listener corespunzatoare va avea mai multe metode (cate una pentru fiecare tip de interactivitate). In consecinta, un obiect handler care ar implementa interfata ar fi obligat sa rescrie toate metodele prezente in ea, corespunzatoare tuturor tipurilor de evenimente posibile. Acest lucru este incomod atunci cand dorim tratarea unui singur caz, deoarece suntem in continuare obligati sa rescriem si celelalte metode ale interfetei; spre exemplu, daca intentionam sa executam cod ca urmare a inchiderii unei ferestre, singura metoda de interes a obiectului listener va fi *windowClosing()*, insa vom fi obligati sa le scriem si pe celelalte 6, cu corp vid!

Pentru situatii de acest fel, Java ne pune la dispozitie clasele de adaptare, cu numele XXXXAdapter (vezi tabelele anterioare), care sunt clase abstracte si pot fi folosite prin derivare (extends), eliminand necesitatea implementarii metodelor nedorite. Fiecare clasa de adaptare implementeaza interfata corespunzatoare si rescrie metodele mostenite, cu implementare vida. In acest fel, subclasele unei clase de adaptare nu mai au niciun fel de obligatie, putand rescrie ce metode doresc.

Lasand la o parte avantajul evident, clasele de adaptare au dezavantajul ca trebuie extinse, si deci nu sunt aplicabile in cazul claselor listener care deja extind o alta clasa.

```

public class InchidereFereastra extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
}

```

7.1.5. Evenimentele si firele de executie Swing

Aplicatiile AWT si Swing pornesc cel putin doua fire de executie: cel al metodei *main()* si cel denumit "event-dispatching thread". In acesta din urma sunt tratate toate evenimentele - inclusiv cele care (re)deseneaza componentele si efectueaza layout-ul containerelor; toate aceste operatiuni sunt introduse intr-o coada de asteptare (FIFO) si tratate pe rand. De aici doua consecinte:

- este vital ca tratarea evenimentelor sa se realizeze cat mai repede cu putinta, pentru a nu tine pe loc plasarea si redesenarea componentelor. Cat timp ruleaza un eveniment de durata, interfata grafica nu va mai raspunde la comenzi utilizatorului. Atunci cand tratarea unui eveniment ia mult timp, se recomanda ca handlerul evenimentului sa creeze un thread (fir de executie) separat care sa execute codul dorit

- orice schimbare a starii componentei trebuie efectuata din event-dispatching thread. In caz contrar, riscam ca starea sa se schimbe in mijlocul redesenarii/relocarii componentei, deoarece firul de executie principal ruleaza concurrent cu cel de evenimente

Există însă o potențială problemă: să spunem că utilizatorul apasă pe un buton, rezultatul fiind demararea unei operațiuni de durată; programatorul porneste un thread separat în acest scop, însă din acest thread are nevoie să apeleze metode care forțează redesenarea aceleiași componentă sau a alteia. Aceste metode trebuie rulate și ele în event-dispatching thread, nu în thread-ul curent. În acest scop, clasa **SwingUtilities** pune la dispozitivă programatorului două metode:

- **invokeLater(Runnable)** - metoda adaugă un task în event-dispatching thread (sub forma unei instante de Runnable) și se întoarce imediat
- **invokeAndWait(Runnable)** – același lucru, numai că așteaptă execuția task-ului și abia apoi se întoarce

7.2. Clase interioare și utilitatea lor în tratarea evenimentelor

7.2.1. Concepție

Să ne imaginăm că dorim ca, la apasarea pe un buton din interfața grafică, să fie afișat în status bar un mesaj. Dacă cream o clasa listener separată pentru buton, ne vom lovi de faptul că din acea clasa nu mai avem acces la status bar ca să-i putem modifica continutul, și ar trebui să construim un getter pentru status bar (ne-elegant) sau să realizăm într-un fel sau altul o cuplare mai strânsă între clasa GUI și clasa handler astfel încât cea din urmă să aibă acces la componentele interfeței grafice. O modalitate de a rezolva aceasta problema este folosirea de clase interioare ("inner classes").

În Java există posibilitatea definirii unei clase în interiorul alteia, pe post de membru al celei exterioare. Efectul este că din metodele celei interioare "se vad" membrii clasei exterioare, inclusiv cei declarati *private*. Iată un exemplu:

```
class GUI extends JFrame{
    private JButton b;
    private JLabel l;

    class HandlerulButonului implements ActionListener {
        public void actionPerformed(ActionEvent ae){ l.setText("Ati dat click!"); }
    }
    public GUI(){
        b=new JButton("Exit"); l = new JLabel();
        b.addActionListener(new HandlerulButonului());
        add(b);add(l);pack();setVisible(true);
    }
    public static void main(String[] args){ new GUI(); }
}
```

In urma compilării exemplului de mai sus vor rezulta două fisiere: *GUI.class* și *GUI\$HandlerulButonului.class*.

O clasa interioara se comportă ca un membru cu drepturi depline al celei exterioare - are acces la toate campurile clasei ce o cuprinde (inclusiv cele declarate private!), însă nu este vizibilă în afara acesteia.

O instantă a clasei interioare depinde strict de una a clasei exterioare, de aceea dacă dorim ca dintr-o a treia clasa să o instantiem pe cea interioară vom scrie:

```
GUI.HandlerulButonului b=(new GUI()).new HandlerulButonului();
```

O clasa interioara poate fi declarată și în cadrul unei metode a clasei exterioare (o clasa locală). În acest caz ea are acces la campurile clasei exterioare și la variabilele locale ale metodei, însă cu condiția că acestea din urmă să fie declarate ca final.

7.2.2. Clase interioare anonime (anonymous inner classes)

Relatia clasa interioara – clasa exterioara poate fi si mai stransa in cazul crearii unei clase interioare anonime – o „clasa de unica folosinta”, declarata si instantiata pe loc in momentul in care avem nevoie de ea. Sintaxa este mai deosebita:

```
class AnonimaInterioara {
    public static void main(String[] args) {
        JButton b=new Button();

        b.addActionListener(new ActionListener() { // incepe definitia clasei anonime
            public void actionPerformed(ActionEvent ae) {
                System.exit(0);
            }
        };// inchidem accolada clasei si paranteza lui ActionListener, apoi
           // incheiem instructiunea cu ";"
    }
}
```

Sintaxa de mai sus se traduce astfel: „adauga ca listener al butonului un nou obiect ce deriva din *ActionListener* si care are implementarea urmatoare” – aici urmand definitia clasei anonime, ce rescrie metodele necesare.

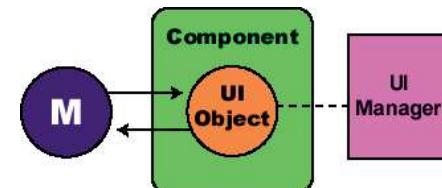
In urma compilarii clasei de mai sus observam aparitia unui fisier suplimentar *AnonimalInterioara\$1.class*. Pentru fiecare clasa interioara anonyma compilatorul creeaza un fisier al carui nume este format din cel al clasei exterioare urmat de \$ si de numarul clasei interioare, ales in functie de ordinea in care apar clasele interioare in codul sursa.

7.3. Lucrul cu componente complexe pentru afisare/editare date

7.3.1. Arhitectura componentelor Swing

Arhitectura componentelor Swing a pornit initial de la conceptul separarii interfetei in Model, View si Controller. Informatiile de stare ale componentei sunt depozitate in model. Din motive practice, in Swing ultimele doua elemente sunt contopite intr-o singura entitate, denumita generic "UI (User Interface) object" in care sunt cuplate mult mai strans view-ul si controllerul.

O deosebire notabila fata de AWT este ca componentele grafice Swing nu mai inglobeaza acum date direct (asa cum se intampla in cazul unui *java.awt.List* sau *java.awt.Choice*, in care adaugam direct elementele de afisat), ci isi iau datele din modelul componentei, care este un obiect separat. De aceea adaugarea de elemente in cazul unui *JComboBox*, *JList* sau *JTable* nu se mai realizeaza cu metode ale componentei in cauza, ci cu metode ale modelului cu care lucreaza componenta; chiar si in cazul in care componenta insasi dispune de metode pentru managementul elementelor afisate, acestea sunt de fapt delegate catre metodele omonime din model.



In functie de tipul de componenta Swing, datele continute in model pot apartine urmatoarelor categorii (care nu se exclud reciproc!):

- date despre starea vizuala a componentei - astfel de modele sunt folosite in cazul componentelor care nu inglobeaza date de la utilizator (ex: *JButton*, *JMenuItem* etc). Modelul depoziteaza toate informatiile despre starea curenta a componentei (selectat sau nu, apasat sau nu, activ sau nu etc). In acelasi timp, componenta insasi ("view"-ul) pune la dispozitia utilizatorului metode care de fapt sunt delegate catre modelul ei (ex: clasa *JButton* are metode precum *setMnemonic()*, *setAction()* etc., care sunt delegate catre *ButtonModel*-ul sau). In acest fel, programatorul poate lucra cu componenta fara sa trebuiasca sa stie ca in spatele ei se afla un model
- date pe care utilizatorul doreste sa le afiseze in cadrul componentei. Componente de tipul *JComboBox*, *JList* sau *JTable* sunt simple front-end-uri (façade) pentru datele aflate in obiectul model ce le corespunde. Datele din model sunt introduse/sterse de catre programator prinapelarea unor metode ale modelului. Modelul instiintea componenta in momentul modificarii datelor continute si, ca urmare, aceasta se actualizeaza

Unele componente folosesc mai multe modele. Spre exemplu, *JList* si *JTable* folosesc un model de date (cel in care stau informatiile afisate) si un model de selectie (cel in care este memorata lista de elemente/celule selectate la un moment dat). In plus, *JTable* foloseste un model pentru coloane, in care memoreaza informatiile despre coloanele afisate.

La modul general, lucrul cu modele inseamna externalizarea informatiilor de stare ale componentei, cu avantaje precum:

- flexibilitate. Programatorul poate decide (daca doreste) unde si cum sunt stocate datele; algoritmul de memorare a datelor direct intr-un *JList* sau *JTable* n-ar putea fi decat unul singur, general, care s-ar dovedi in dese cazuri departe de optim
- ne-duplicarea datelor. Daca informatia este deja depozitata intr-un obiect al aplicatiei sau intr-o baza de date externa, ar fi neconvenabil si inefficient sa fim nevoiti sa cream o copie a datelor in componenta grafica. Pe langa spatiul suplimentar ocupat, ar trebui in plus sa avem grija permanent ca eventualele modificari aduse datelor (prin intermediul componentelor ce permit editarea lor - ex: *JComboBox*, *JTable*) sa fie propagate inapoi catre sursa lor
- propagarea automata a schimbarilor din model. Exista in acest scop infrastructura de evenimente descrisa anterior
- mai multe componente isi pot prelua datele din acelasi model. Spre exemplu, un *JSlider* si un *JProgressBar* pot afisa o aceeasi valoare preluata dintr-un unic obiect model; un *JTable* poate afisa detaliat aceeasi lista de elemente prezenta intr-un *JList* etc

Interactiunea intre model si componenta care il foloseste este bidirectionala, dupa cum urmeaza:

- componenta initiaza interactiunea cu modelul in urmatoarele cazuri:
 - pentru extragerea datelor de care are nevoie pentru a-si putea desena continutul. In acest scop modelul dispune de un ansamblu de metode de tip getter
 - pentru modificarea datelor cuprinse in model, atunci cand componenta permite utilizatorului modificarea datelor (ex: editare in *JComboBox* sau *JTable*). In astfel de cazuri modelul este dotat si cu seteri
- modelul initiaza interactiunea cu componenta la schimbarea datelor pe care el le contine. Interactiunea ia forma unei notificari de modificare trimisa componentei; reactia tipica a componentei este folosirea metodelor getter ale modelului pentru obtinerea noii stari si re-afisarea actualizata

Relatia intre model si componenta ce il foloseste este bazata tot pe pattern-ul *Observer* si pe infrastructura de evenimente. Modelul nu cunoaste tipul exact de componenta cu care interactioneaza; el mentine o lista de obiecte *listener* - cele care vor fi anuntate de schimbarile produse in model. In momentul aparitiei unei modificari, modelul va parcurge lista de listeneri si il va instiinta pe fiecare de modificarile aparute. In functie de complexitatea datelor, unele modele ofera simpla notificare ("datele s-au schimbat") sau o notificare optimizata ("aceste date s-au schimbat"). Componenta (sau componente!) cu care "colaboreaza" modelul se inregistreaza ca listener al acestuia; la primirea notificarii ea se va folosi de metodele getter ale modelului pentru a solicita din nou datele care s-au schimbat.

Nota: se pot inregistra ca listeneri si alte obiecte decat componenta grafica cu care colaboreaza modelul. Deseori dorim sa reacionam la schimbarea datelor din model ca urmare a interactiunii cu view-ul atasat lui! (ex: schimbarea informatiei din modelul de date al unui *JTable* in urma editarii unei celule).

Notificarea initiată de model presupune apelarea unei anumite metode a fiecarui listener; pentru ca un obiect sa poata juca rolul de listener pentru modelul unei componente, el trebuie sa ofere garantia existentei metodelor necesare, lucru realizat prin implementarea unei interfete. Fiecare tip de model are o interfata corespunzatoare; listenerii lui vor avea ca tip de date acea interfata, astfel ca singurele metode pe care modelul le poate apela listenerilor sunt cele din interfata. Spre exemplu, modelul de buton mentine o lista de obiecte *ActionListener* (pe care le refera prin intermediul unor referinte de tip *ActionListener*, indiferent de tipul lor de date!); interfata *ActionListener* contine metoda *actionPerformed()*, ce va fi apelata fiecarui listener la aparitia unei modificari.

In scopul stabilirii modelului, componente complexe dispun de metoda *setModel()*. Un efect secundar al acesteia este ca componenta (sau un "subansamblu" al ei) se auto-inregistreaza in lista de listeneri a modelului, astfel incat sa fie notificata si sa se actualizeze automat in momentul modificarii datelor din model.

Trebuie remarcat faptul ca, in cadrul arhitecturii unei componente Swing, pattern-ul *Observer* este folosit bidirectional:

- componenta doreste sa poata colabora cu un model fara a impune tipul exact al acestuia, ci doar o lista de metode strict necesare, motiv pentru care modelul trebuie sa implementeze o anumita interfata
- in sens opus, modelul doreste sa poata notifica listenerii sai fara a le impune tipul de date ci doar anumite seturi de metode, de unde si necesitatea ca listenerii sa implementeze la randul lor alt tip de interfata

Un exemplu concret: in cazul unui *JTable*, UI-ul trebuie sa poata afla de la model ce dimensiune au datele (cate linii, cate coloane), titlurile coloanelor etc, iar pentru aceasta modelul are o suita de metode getter; pe de alta parte, utilizatorul poate modifica datele din tabel prin editare directa, ceea ce impune prezenta unor setteri in lista de metode ale modelului, care sa fie apelati de catre UI la nevoie. Toate acestea sunt cuprinse in interfata *TableModel*. De cealalta parte, pentru ca *JTable-ul* sa poata reaciona la modificarile din model, o anumita clasa interna a sa implementeaza interfata *TableModelListener*.

Toate componentele grafice Swing respecta arhitectura prezentata anterior; atunci cum se face ca pana acum am facut abstractie de ea, si de ce acum ne punem problema ei? Explicatie: impartim componentele Swing in doua categorii:

- componente simple - sunt componente care fie isi stocheaza in model propria stare si nu afiseaza date externe componentei, fie acestea sunt nesemnificative si este improbabil ca programatorul sa aiba nevoie sa aleaga propria modalitate de stocare. In aceste cazuri, componenta are metode de management al informatiei din model care sunt delegate catre metodele modelului (ex: metoda *setMnemonic()* a butoanelor)
- componente complexe - este cazul componentelor care afiseaza informatii complexe sau in cantitati mari (ex: *JList*, *JTable*, *JTree* etc). In cazul acestora, datele se gasesc in afara componentei, iar programatorul doreste flexibilitate in stocarea si managementul lor. Materialul de fata detaliaza utilizarea componentelor complexe

7.3.2. Afisarea si editarea informatiei intr-o componenta complexa

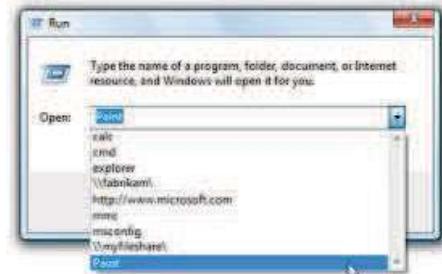
Componentele Swing sunt containere ce pot afisa diferite tipuri de date, nu numai text (de remarcat ca *JComponent* deriva din *java.awt.Container*). De aceea, clasele de tip model prezente in Swing lucreaza cu date de tip *Object* – acestea putand fi String-uri, imagini sau orice altceva, iar programatorul putand decide modul in care GUI-ul va prezinta pe ecran aceste date (de exemplu, putem impune ca datele de tip boolean sa fie afisate intr-un tabel sub forma unui checkbox, datele care nu pot lua decat un set de valori discrete – sub forma unui drop-down list etc). Se poate de asemenea alege in ce fel sa fie editate aceste date, pentru componente care suporta si schimbarea valorilor prezентate (ex: *JSpinner*, *JComboBox*, *JTable* etc)

Pentru aceasta, Swing defineste doua concepte – cel de *renderer* si cel de *editor*. O componenta Swing care prezinta, prin intermediul UI-ului sau, date stocate in model, se foloseste pentru aceasta de serviciile unui obiect de tip *renderer*.

Renderer-ul este o componenta grafica; fiecare element este afisat intr-o instanta de renderer. Programatorul poate decide ce fel de renderer sa foloseasca in functie de tipul de date afisat (numar, Boolean, String, Icon etc) sau in functie de pozitia elementului in UI: de exemplu, in cazul unui *JTable*, putem alege ca renderer-ul pentru IQ-ul unei persoane sa fie o reprezentare vizuala, sub forma unui *JProgressBar*; pentru aceasta impunem ca coloana corespunzatoare sa foloseasca un *JProgressBar* ca renderer. Spre exemplu, *JList*, *JComboBox* si *JTable* folosesc ca renderer implicit o componenta de tip *JLabel*; componenta complexa decide, in functie de natura datelor de afisat, daca sa le reprezinte in cadrul label-ului sub forma de poza sau de text.

Pentru componente care suporta si editarea valorii, in mod analog exista conceptul de *editor* – componenta cu ajutorul careia se realizeaza editarea.

Editor-ul este si el la alegerea programatorului: de exemplu, intr-un *JTable*, intregii poti fi editati cu un simplu *JTextField*, sau cu un *JFormattedTextField* (ca sa impunem un anumit format), sau cu un *JSpinner* etc).



Fie exemplul concret al combo box-ului. In figura alaturata, elementele din lista derulabila (cele read-only) sunt afisate fiecare in cate o instanta de renderer; in schimb suprafata de baza a combo box-ului contine o instanta de editor, care in acest caz este un text field.

7.3.3. Modalitati de a crea un model de date pentru o componenta complexa

Pentru definirea modelului unei componente complexe programatorul are la dispozitie mai multe variante:

- cea mai generala – crearea unei clase model ce implementeaza interfata corespunzatoare. Este cea mai laborioasa, deoarece presupune scrierea tuturor metodelor interfetei (inclusiv managementul listei de listeneri!), insa ofera programatorului flexibilitate maxima, in masura in care acesta si-o doreste

- cea facila, dar inflexibila: constructori ai componentei care primesc ca argument array-uri de *Object*. In acest fel, programatorul poate crea doar array-ul cu date, facand abstractie de existenta modelului. Flexibilitatea este insa redusa, deoarece constructorul genereaza automat un model read-only
- cea facila, care permite inclusiv modificarea datelor din model: exista implementari complete ale interfetelor model (clase precum *DefaultListModel*, *DefaultTableModel* etc), care dispun de metode suplimentare convenabile pentru adaugare/stergere de date din model
- cea intermediara, compromisul intre flexibilitate si efort de implementare: in cazul unora dintre componentele complexe (ex: *JList*, *JTable*), exista clase abstracte care implementeaza interfetele necesare si care au deja definita o parte a functionalitatii (in principal sistemul de management si notificare a listener-ilor – vezi metodele *fire**()). Programatorul va fi nevoie sa rescrie numai setul de metode absolut necesar, avand insa flexibilitatea de a modifica si restul metodelor in caz de nevoie

Sectiunile urmatoare vor detalia modalitatile de implementare ale acestor variante pentru principalele componente Swing complexe.

7.3.4. Utilizarea componentelor de tip *JList*

7.3.4.1. Capabilitati. Clase si metode aferente

Componentele de tip *JList* afiseaza o succesiune de elemente, permitand selectarea de catre utilizator a unuia sau mai multora dintre ele. Pentru afisarea fiecarui element este folosita o instanta de renderer; cel implicit este un *JLabel*, ceea ce face ca din oficiu *JList* sa poata afisa text sau imagini (icon). Programatorul poate decide in totalitate cum sa fie afisat fiecare element prin definirea renderer-ului dorit.

Elementele afisate pot fi dispuse pe una sau mai multe coloane, in functie de valoarea atributului *layoutOrientation* (vezi tabelul de metode). Metoda *setLayoutOrientation()* este cea care stabileste modul de aranjare a elementelor, folosind constante din clasa *JList*:

- VERTICAL - elementele vor fi dispuse pe o singura coloana
- HORIZONTAL_WRAP - elementele vor fi dispuse pe orizontala, unul dupa altul; cand nu mai ramane spatiu pe randul curent, se trece pe randul urmator de la inceput
- VERTICAL_WRAP - elementele vor fi dispuse unul dupa altul pe verticala pana la epuizarea spatiului pe coloana, dupa care se va incepe de sus o coloana noua



Nota: componentele *JList* sunt in general plasate in containere de tip *JScrollPane* pentru a afisa automat scrollbar-uri atunci cand numarul de elemente este mare.

Obiectul *JList* foloseste doua modele externe:

- **modelul pentru date**, ce contine informatia afisata in *JList*. Modelul ia forma unui obiect a carui clasa implementeaza interfața *ListModel*. Setarea modelului de date se realizeaza cu metoda *setModel()* a clasei *JList*
- **modelul pentru selectie**, ce memoreaza stilul de selectie al listei si setul de elemente selectate in ea. Obiectul ce joaca rolul de model de selectie trebuie sa implementeze interfața *ListSelectionModel*. Modelul de selectie poate fi setat cu metoda *setSelectionModel()* a clasei *JList*

JList permite selectie simpla sau multipla. Stabilirea modului de selectie se realizeaza cu metoda *setSelectionMode()* ce primește ca argument constanta din clasa *ListSelectionModel*:

- SINGLE_SELECTION - un singur element poate fi selectat la un moment dat. Pozitia sa va putea fi extrasa cu metoda *getSelectedIndex()*, iar valoarea cu *getSelectedValue()*
- SINGLE_INTERVAL_SELECTION - utilizatorul poate selecta o singura insiruire de elemente consecutive
- MULTIPLE_INTERVAL_SELECTION - utilizatorul poate selecta liber orice combinatie de elemente. In aceste ultime doua cazuri, pozitiile elementelor selectate pot fi extrase cu *getSelectedIndices()* iar valorile lor cu *getSelectedValues()*

Metode <i>JList</i>	Descriere
void setSelectionMode(int) int getSelectionMode()	stabileste daca lista permite selectie multipla sau exclusiv simpla. Argumentele posibile sunt constante din clasa <i>ListSelectionModel</i> : SINGLE_SELECTION (utilizatorul poate selecta un singur rand),

Studentul poate utiliza prezentul material si informatiile continute in el exclusiv in scopul asimilarii cunostintelor pe care le include, fara a afecta dreptul de proprietate intelectuala detinut de autor.

	SINGLE_INTERVAL_SELECTION (se poate selecta un singur interval de elemente consecutive) si MULTIPLE_INTERVAL_SELECTION (se pot selecta mai multe intervale de elemente simultan)
void setSelectedIndex(int) int getSelectedIndex()	extragerea sau setarea pozitiei elementului selectat. Atunci cand lista permite selectie multipla, getterul intoarce pozitia cea mai mica dintre cele selectate
void setSelectedIndices(int[]) int[] getSelectedIndices()	idem ca mai sus, pentru cazul in care lista a fost configurata sa permita selectie multipla
Object getSelectedValue() void setSelectedValue(Object,boolean) List getSelectedValuesList()	getterii permit obtinerea elementului selectat (pentru selectie simpla) sau a unei colectii ce contine toate elementele selectate (pentru selectie multipla). Setterul stabileste elementul selectat, al doilea argument specificand daca lista ar trebui sa faca scroll pentru face vizibil noul element selectat
void setLayoutOrientation(int) int getLayoutOrientation()	stabileste cum sunt organizate elementele pe linii si coloane. Valurile posibile sunt constante din clasa JList: HORIZONTAL_WRAP/VERTICAL_WRAP (mai multe coloane) si VERTICAL (o singura coloana)
void setVisibleRowCount(int) int getVisibleRowCount()	Setarea si extragerea numarului de linii afisate. In orientarea verticala, setterul stabileste numarul de linii preferat; in celelalte orientari determina modul de spargere in linii (JList-ul va incerca sa efectueze wrapping-ul de asa natura incat sa respecte numarul de linii solicitat)
void setModel(ListModel) ListModel getModel()	setarea sau extragerea modelului de date (vezi si sectiunea corespunzatoare de mai jos)
ListSelectionModel getSelectionModel() void setSelectionModel(ListSelectionModel)	setarea sau extragerea modelului de selectie
void setCellRenderer(ListCellRenderer) ListCellRenderer getCellRenderer()	stabilirea sau extragerea componentei folosite pe post de renderer (vezi sectiunea corespunzatoare de mai jos)
boolean getValueIsAdjusting()	metoda delegata catre cea omonima din model (vezi sectiunea dedicata evenimentelor)

7.3.4.2. Evenimente specifice si tratarea acestora

Principalul eveniment specific generat de o componenta *JList* este *ListSelectionEvent*, generat de schimbarea selectiei in cadrul listei. Evenimentul este unul semantic, deoarece modificarea setului listei de elemente selectate se poate realiza in mai multe moduri (folosind mouse sau tastatura). In plus, modelul de date genereaza evenimente de tip *ListDataEvent* ori de cate ori datele continute in el se modifica, notificand prin intermediul lor *JList*-ul.

Interfata	Utilitate interfata	Metode	Descriere
ListDataListener	implementata de obiecte care trateaza modificarile de informatie din modelul de date	void intervalAdded(ListDataEvent) void intervalRemoved(ListDataEvent) void contentsChanged(ListDataEvent)	- notificare de adaugare a unui interval de elemente - notificare de stergere a unui interval de elemente - notificare de modificare complexa (care nu poate fi descrisa de celelalte doua metode)
ListDataEvent	obiect eveniment creat de catre ListModel atunci cand datele continute se modifica	int getIndex0() int getIndex1() getType()	- prima pozitie a zonei ce a suferit modificarile - ultima pozitie a zonei ce a suferit modificarile - tip modificare (adaugare/stergere interval etc)
ListSelectionListener	implementata de obiectele care trateaza modificarile de selectie intr-un JList	valueChanged(ListSelectionEvent)	- metoda apelata listenerului ca urmare a schimbarii setului de elemente selectate
ListSelectionEvent	obiect eveniment creat de modelul de selectie atunci cand setul de elemente selectate se modifica	int getFirstIndex() int getLastIndex() boolean getValueIsAdjusting()	- prima pozitie a zonei ce a suferit modificarile de selectie - ultima pozitie a zonei ce a suferit modificarile de selectie - notificarea este una intermedia? (vezi mai jos explicatia)

Felul in care se genereaza evenimentele cere o explicatie separata. Sa consideram cazul unei liste ce permite selectie unica (SINGLE_SELECTION) in care exista deja un element selectat; la click pe un altul se vor genera doua evenimente: unul cauzat de deselectarea elementului curent, si inca unul aferent selectarii celui nou. Programatorului i se ofera astfel flexibilitatea de a reactiona separat la cele doua etape ale schimbarii de stare a listei; pe de alta parte, sunt destule cazuri in care dorim o singura reacție, la sfarsitul intregului proces, care sa tina cont de selectia finala. In ambele cazuri insa se executa aceeasi metoda a listenerului; pentru a determina in ce etapa ne aflam exista metoda *getValueIsAdjusting()*: la primul eveniment selectia este inca in curs de ajustare (si deci metoda va produce true), pe cand la al doilea metoda *getValueIsAdjusting()* va returna false.

```
// o lista care reacționează la dublu click pe un element și la schimbarea selectiei
JFrame f = new JFrame();
final JList lista = new JList(new String[]{"Java", "PHP", "C#"});
f.getContentPane().add(lista); f.pack();
lista.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
```

```

        if (e.getClickCount() == 2) {
            int index = lista.locationToIndex(e.getPoint());
            System.out.println("Dublu click pe elementul " + index);
        }
    });
lista.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent evt) {
        if(!evt.getValueIsAdjusting()){
            System.out.println("Ati selectat elementul numarul"+lista.getSelectedIndex());
        }
    }
});
f.setVisible(true);

```

7.3.4.3. Definirea modelului pentru date

7.3.4.3.1. Principii de functionare

Dupa cum cunoastem, *JList* este o simpla fatada pentru datele din modelul sau. Intre *JList* si model are loc o comunicatie bidirectionala, dupa cum urmeaza:

- *JList* trebuie sa ii poata solicita modelului numarul de elemente si elementul de pe fiecare pozitie
- modelul trebuie sa poate notifica *JList*-ul ori de cate ori datele sale se schimba

Aceasta comunicatie presupune cate un set de metode pe care fiecare dintre obiecte i le apeleaza celuilalt. Este necesara deci o garantie reciproca: *JList* trebuie sa se asigure de prezenta metodelor trebuincioase in model si invers. Acest lucru se realizeaza prin implementarea de interfete: modelul trebuie sa implementeze interfata *ListModel*, iar *JList* (mai exact, un anumit "subansamblu" al sau) va implementa *ListDataListener*. In acest fel, modelul nu este nici macar constient de faptul ca lucreaza cu un *JList* - orice obiect ce implementeaza *ListDataListener* va putea fi anuntat de schimbarile din model. Reciproc, *JList* isi poate citi datele din orice obiect, atata timp cat respectivul implementeaza interfata *ListModel*.

Interfata *ListModel* impune claselor ce o implementeaza urmatorul set de metode:

- **void addListDataListener(ListDataListener)** si **void removeListDataListener(ListDataListener)** - sunt metode care asigura managementul listei de listeneri atasati modelului de date (cei care vor fi instiintati ori de cate ori apar schimbari in informatia din model)
- **public int getSize()** - intoarce numarul de elemente din model
- **public Object getElementAt(int pozitie)** - intoarce elementul de pe pozitia solicitata

Ultimele doua metode de mai sus ii ofera posibilitatea *JList*-ului sa-i solicite modelului numarul de elemente si apoi, in cadrul unei bucle for, sa extraga fiecare element in parte. Primele doua metode ii ofera *JList*-ului garantia de a se putea adauga pe sine ca listener al modelului, astfel incat sa se poata actualiza cand este nevoie.

Nota: la apelarea metodei setModel() a clasei JList, obiectul JList se auto-adauga implicit ca listener al modelului.

Observam ca metoda *getElementAt()* intoarce *Object*, asadar putem returna din model orice fel de obiect Java. Cu renderer-ul implicit, *JList* are din start capacitatea de a afisa text sau imagini (insa nu ambele simultan). Pentru fiecare element obtinut de la modelul sau, *JList* analizeaza tipul obiectului si actioneaza astfel:

- daca elementul este *instanceof Icon* il va afisa sub forma de poza, setand icon-ul label-ului ce constituie renderer-ul implicit
- daca elementul este de orice alta natura, *JList* va afisa in renderer rezultatul metodei *toString()* a obiectului

7.3.4.3.2. Variante de obtinere a unui model

Exista cel putin cateva variante de a oferi un model unei componente *JList*:

- **un tablou de obiecte.** *JList* dispune de cel putin doua modalitati de a "se servi" dintr-un tablou de elemente de orice natura (vezi detalii in sectiunile urmatoare). Este solutia ideală atunci cand dorim sa afisam rapid o lista de elemente ce nu se va modifica mai apoi

- **un obiect de tip DefaultListModel.** *DefaultListModel* este o clasa concreta “la cheie” - ea implementeaza *ListModel* si contine metode pentru adaugare/stergere/cautare/modificare de elemente. Este solutia ideală atunci cand lista se va modifica in timp, insa nu este convenabila cand datele sunt deja memorate intr-un alt obiect sau intr-o baza de date, deoarece ar cere duplicarea acestora (crearea unei copii a lor in obiectul de tip *DefaultListModel*)
- **un obiect a carui clasa extinde AbstractListModel.** Clasa *AbstractListModel* implementeaza interfata *ListModel* si ofera “de-a gata” toata infrastructura de listeneri si de notificare a acestora, lasand programatorul sa se concentreze strict pe metodelor care tin de manipularea datelor (*getSize()* si *getElementAt()*)
- **un obiect a carui clasa implementeaza ListModel.** Este solutia cea mai flexibila, dar si cea mai laborioasa. Utilizarea ei se justifica numai atunci cand niciuna dintre celelalte solutii nu este convenabila

Vom detalia in continuare toate cazurile cu exceptia ultimului.

7.3.4.3.3. Modelul ca array de Object

Cea mai simpla modalitate de a afisa informatie in *JList* este de a-i oferi acestuia un array sau o colectie¹ de obiecte. Pe baza tabloului sau colectiei *JList* isi va defini intern un model care va fi insa immutable - nu vor putea fi adaugate sau sterse elemente din el. Solutiile posibile sunt urmatoarele:

- folosirea unuia dintre constructorii cu argument ai lui *JList*: **JList(Object[])** sau **JList(Vector)**
- folosirea metodelor **setListData(Object[])** sau **setListData(Vector)**

```
String[] date = { "Ion", "Gheorghe", "Maria" };
JList lista = new JList(date);
// sau, pentru a seta lista de elemente afisate intr-un moment ulterior instantierii:
lista.setListData(date);
```

7.3.4.3.4. Modelul ca DefaultListModel

Clasa *DefaultListModel* este o clasa concreta care implementeaza *ListModel* si care mentine o lista modificabila de elemente, disponand in acest sens de toate metodelor necesare pentru operatiile uzuale (adaugare/stergere/cautare/ modificare de element). Cateva dintre ele sunt prezентate in tabelul alaturat.

Nota: pentru un *JList* care foloseste *DefaultListModel* pe post de model de date, modificarea listei de elemente cere o conversie de referinta, deoarece metoda *getModel()* produce o referinta mai generala, de tip *ListModel*:

Metode de interes DefaultListModel	Descriere
<i>Object get(int pozitie)</i> <i>Object getElementAt(int pozitie)</i>	Extragerea elementului de pe o anumita pozitie
<i>void set(int pozitie, Object element)</i> <i>void setElementAt(Object element, int pozitie)</i>	Inlocuirea elementului de pe o anumita pozitie
<i>void add(int pozitie, Object element)</i> <i>addElement(Object element)</i>	Adaugare element pe pozitia specificata Adaugare element la sfarsitul listei
<i>removeElement(Object element)</i> <i>removeElementAt(int pozitie)</i> <i>remove(int pozitie)</i>	Stergerea unui element prin specificarea fie a valorii, fie a pozitiei sale in cadrul listei
<i>getSize()</i> <i>size()</i>	Returneaza numarul de elemente din lista

```
JList lista = new JList(new DefaultListModel());
// ... iar ulterior, cand dorim sa accesam modelul:
DefaultListModel m = (DefaultListModel)lista.getModel();
m.add(5,"The fifth element");
```

7.3.4.3.5. Modelul ca instanta a unei clase care extinde AbstractListModel

Clasa *AbstractListModel* implementeaza interfata *ListModel* si ofera “de-a gata” toata infrastructura de management de listeneri. Pe langa cele doua metode obligatorii provenite din clasa *ListModel* (*addListDataListener()* si *removeListDataListener()*) gasim in clasa *AbstractListModel* cateva metode cu nivel de acces *protected* si al caror nume

1 O colectie Java reprezinta un obiect care are capabilitatea de a contine alte obiecte. Unul dintre capitolele urmatoare va trata detaliat subiectul

incepe cu fire (ex: `fireContentsChanged()`). Acestea sunt modalitatatile de a notifica toti listenerii modelului (si implicit `JList-ul`) ori de cate ori datele din model se modifica.

Atentie! Ori de cate ori modificati continutul sau ordinea datelor din model, nu uitati sa apelati una dintre metodele fire...() ! In caz contrar, `JList-ul` nu se va actualiza automat.

Sigurele doua metode ramase abstracte in clasa `AbstractListModel` sunt `getSize()` si `getElementAt()`, pe care trebuie sa le implementeze programatorul ce extinde clasa. Iata un exemplu simplu de model ce isi citeste datele dintr-un tablou, si care la fiecare depasire a capacitatii tabloului creeaza unul nou, mai incapator:

```
class ListaNume extends AbstractListModel{
    String[] nume = {"Ion", "Gheorghe"};

    public int getSize(){ return nume.length; }
    public Object getElementAt(int pozitie){ return nume[pozitie]; }
    public void adaugaNume(String n){
        nume = Arrays.copyOf(nume, nume.length+1);
        nume[nume.length-1] = n;
        fireContentsChanged(this,-1,-1); // determina actualizarea integrala a JList-ului
    }
}
// ...iar pentru a seta un model de acest tip pentru JList-ul lst:
listaNume l = new ListaNume();
lst.setModel(l);
l.adaugaNume("Vasile");
lst.setModel(l); // JList-ul se va actualiza automat
```

7.3.4.4. Definirea unui Renderer particularizat

O componenta de tip `JList` poate afisa datele din model in diverse forme, nu neaparat text sau icon. Pentru ca programatorul sa aiba libertatea de a reprezenta datele oricum doreste, `JList` nu face el insusi afisarea, ci se foloseste de serviciile unui obiect ajutator – asa-numitul *renderer* – care este folosit pentru afisarea fiecarui element si a carui clasa trebuie sa implementeze `ListCellRenderer`. `JList` dispune de metodele `getCellRenderer()` si `setCellRenderer()` pentru lucrul cu acest obiect.

Interfata `ListCellRenderer` impune o singura metoda, `getListCellRendererComponent()`, care returneaza componenta ce va fi afisata pe post de “cofraj” al fiecarui element in `JList`. Aspectul renderer-ului trebuie sa tina cont de diferite informatii: pozitia elementului in lista, daca acesta este selectat, daca are focus etc – toate acestea reflectandu-se in argumentele metodei `getListCellRendererComponent()`. Presupunand ca dorim sa afisam intregii vizual, sub forma unui progress bar, iata cum am putea proceda:

```
class RendererListaProgressBar implements ListCellRenderer{
    public Component getListCellRendererComponent(JList list, Object value, int index, boolean
        isSelected, boolean cellHasFocus) {
        JProgressBar p = new JProgressBar();
        // vrem sa fie afisata si valoarea numerica
        p.setStringPainted(true); p.setString(value.toString());
        // limitele si valoarea curenta, pentru reprezentarea vizuala
        p.setMinimum(0); p.setMaximum(50); p.setValue(Integer.parseInt(value.toString()));
        // daca elementul e chiar cel selectat, semnalizam vizual printr-un border negru de 2pixeli
        if(isSelected) p.setBorder(new LineBorder(Color.RED,2));
        return p;
    }
}
// ...iar pentru JList, dupa crearea lui:
Lista.setCellRenderer(new RendererListaProgressBar());
```

7.3.5. Utilizarea componentelor de tip JComboBox

7.3.5.1. Capabilitati si principii de functionare

Un *JComboBox* este o componenta ce afiseaza un set de elemente din care utilizatorul poate selecta unul singur. ComboBox-ul poate fi sau nu editabil. In caz afirmativ, utilizatorul poate introduce/modifica manual valoarea curenta; in caz contrar, valoarea curenta este stabilita exclusiv prin selectarea uneia dintre cele cuprinse in lista derulabila de elemente.

Functionalitatea oferita de combo box este realizata prin colaborarea a cateva obiecte:

- lista derulabila de elemente - este realizata cu ajutorul unui popup menu
- renderer-ul - folosit pentru a afisa fiecare element din popup menu, si de asemenea pentru valoarea curenta atunci cand combo box-ul este needitabil. Ca si in cazul *JList*-ului, renderer-ul implicit este un *DefaultListCellRenderer* derivat din *JLabel*, ceea ce confera combo box-ului capabilitatea de a afisa pentru fiecare element fie un text, fie un icon (dar nu ambele simultan, deoarece combo box-ul, ca si *JList*-ul, se orienteaza in functie de tipul de date al valorii primite din model)
- editor-ul - folosit pentru afisarea valorii curente atunci cand combo box-ul este editabil. Editorul implicit este un text field

Componenta *JComboBox* se foloseste de un model de date care trebuie sa implementeze una din doua interfete:

- **ComboBoxModel** - pentru un model nemodificabil
 - **MutableComboBoxModel** - pentru un model ale carui date se pot modifica.
- Atentie! O parte dintre metodele clasei *JComboBox* (cele care modifica datele continue) sunt delegate catre modelul de date (ex: *insertItemAt()*); pentru ca ele sa functioneze trebuie ca modelul sa implementeze *MutableComboBoxModel*!**

Metode JComboBox	Descriere
<code>JComboBox(Object[])</code> <code>JComboBox(Vector)</code>	crearea unui JComboBox care afiseaza datele dintr-un tablou sau colectie
<code>void addItemAt(Object)</code> <code>void removeItem(Object)</code> <code>void insertItemAt(Object,int pozitie)</code> <code>void removeItemAt(int pozitie)</code> <code>void removeAllItems()</code>	modificarea listei de elemente actionand direct asupra combo box-ului. Metodele sunt delegate catre cele din model
<code>int getItemCount()</code>	numarul de elemente din model
<code>int getSelectedIndex()</code> <code>void setSelectedIndex()</code>	stabilirea sau extragerea pozitiei elementului selectat, numerotata intre 0 si N-1, unde N este numarul de elemente afisate in lista derulabila
<code>Object getSelectedItem()</code> <code>void setSelectedItem(Object)</code>	stabilirea sau extragerea valorii curente. Atentie! Pentru un combo box editabil, aceasta valoare poate fi cea din editor, care nu este prezenta in lista derulabila!
<code>void setEditable(boolean)</code> <code>boolean isEditable()</code>	stabileste sau determina daca combo box-ul este editabil
<code>void setRenderer(ListCellRenderer)</code> <code>ListCellRenderer getRenderer()</code> <code>void setEditor(ComboBoxEditor)</code> <code>ComboBoxEditor getEditor()</code>	stabilirea si extragerea renderer-ului si editor-ului utilizat in cadrul combo box-ului
<code>void showPopup()</code> <code>void hidePopup()</code>	deschiderea sau inchiderea listei derulabile de catre programator (nu ca urmare a actiunii utilizatorului)

Nota: valoarea editata manual este separata de lista derulabila si nu este introdusa automat in aceasta din urma cand userul apasa Enter!

7.3.5.2. Evenimente specifice

Componentele de tip *JComboBox* pot genera doua tipuri de evenimente specifice:

- *ActionEvent* - generat la stabilirea valorii curente prin selectarea unui element din lista derulabila sau prin apasarea tastei Enter in editor (pentru combo editabil), **chiar si atunci cand noua valoare este identica cu cea veche!** Trebuie tinut de asemenea cont de faptul ca, atunci cand combo box-ul este editabil si utilizatorul apasa Enter in cadrul editor-ului, se genereaza doua *ActionEvent*-uri! (unul pentru incheierea editarii si unul pentru schimbarea valorii selectate)
- *ItemEvent* - generat la schimbarea starii de selectie a fiecarui element in parte. Distingem doua cazuri:
 - daca utilizatorul selecteaza din lista sau introduce manual (apasand apoi Enter) o alta valoare decat cea curenta, vor fi generate doua evenimente de tip *ItemEvent*: unul aferent deselectarii valorii curente si inca unul corespunzator selectarii/introducerii noii valori. Acest lucru se intampla si in timpul in care lista derulabila este vizibila si utilizatorul parurge din taste lista de valori, deoarece aceasta cauzeaza schimbarea valorii curente!

- daca valoarea curenta este identica cu cea veche (ex: userul da click pe acelasi element din lista derulabila), nu se genereaza **ItemEvent**-uri, in schimb se genereaza **ActionEvent**!

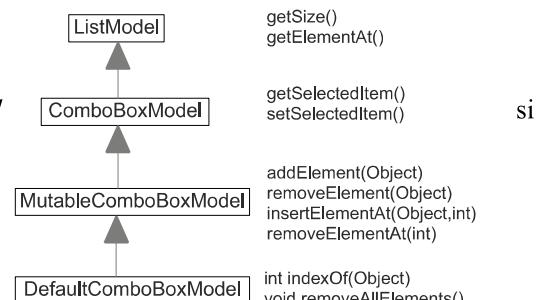
```
// un combo box cu nume de oameni care afiseaza poza fiecaruia pe masura parcurgerii listei
public class PozeOameni {
    public static void main(String[] args) {
        String[] oameni = {"elena", "sorin", "liana"};
        JFrame f = new JFrame();
        final JLabel labelPoza = new JLabel(); labelPoza.setPreferredSize(new Dimension(100,100));
        JComboBox combo = new JComboBox(oameni);
        combo.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent evt) {
                if (evt.getStateChange() == ItemEvent.SELECTED) { // la selectarea unei persoane
                    labelPoza.setIcon(new ImageIcon(
                        // pozele stau intr-un subpachet poze al pachetului curent si fiecare
                        // poza are nume identic cu al persoanei si extensia .jpg
                        getClass().getResource("poze/" + evt.getItem().toString() + ".jpg")));
                }
            }
        });
        Container contentPane = f.getContentPane(); contentPane.setLayout(new BorderLayout());
        contentPane.add(labelPoza); contentPane.add(combo, "South");
        f.pack(); f.setVisible(true);
    }
}
```

7.3.5.3. Crearea modelului de date

Iata cum se aplica variantele de creare a unui model discutate anterior pentru componenta de tip *JComboBox*:

- crearea integrala a unei clase model presupune implementarea uneia din doua interfeite:
 - **ComboBoxModel** - pentru un model nemodificabil
 - **MutableComboBoxModel** - pentru model cu date modificabile. Indicat in cazul in care dorim sa functioneze metodele de modificare a datelor prezente in API-ul clasei *JComboBox* (ex: *insertItemAt()*).
- constructori cu argumente: *JComboBox(Object[])* si *JComboBox(Vector)*. Constructorii preiau datele din tabloul, respectiv colectia pasata ca argument si creeaza un model intern read-only
- clasa *DefaultComboBoxModel*, care beneficiaza de constructori ce primesc ca argument array de obiecte sau colectie de tip *Vector*, si care dispune de toate metodele necesare pentru managementul listei de elemente
- o clasa care extinde *AbstractListModel* si implementeaza una dintre interfetele prezentate anterior. Interfetele *ComboBoxModel* *MutableComboBoxModel* deriva amandoua din *ListModel*

Figura alaturata evidentaiza metodele suplimentare adaugate de fiecare interfata sau clasa din ierarhie fata de parintele sau. Grati implementarii interfetei *MutableComboBoxModel*, clasa *DefaultComboBoxModel* contine toate metodele listate!



Atentie! Nu uitati sa trimiteți notificările din model către listenerii săi (și implicit combo box-ului) cand se modifica datele modelului!

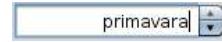
```
JComboBox combo;
// creare model pe baza un array de obiecte
String[] tari = {"Romania", "Germania", "Austria"}; combo = new JComboBox(tari);
// setare model ulterior, folosind DefaultListModel
combo.setModel(new DefaultComboBoxModel(new String[]{"Uganda", "Peru"}));
// adaugare de element in modelul setat anterior
((DefaultComboBoxModel) combo.getModel()).addElement("China");
// ...sau alternativ, cu metode din JComboBox:
combo.addElement("Venezuela");
// determinarea pozitiei unui element din lista:
System.out.println("Pozitia Chinei: "+((DefaultComboBoxModel) combo.getModel()).indexOf("China"));
```

Nota: pentru un exemplu mai larg de model ce implementeaza *MutableComboBoxModel* consultati anexa de exemple a materialului.

7.3.6. Utilizarea componentelor de tip JSpinner

Un *JSpinner* este o componenta care permite selectia sau editarea unei valori dintr-un domeniu de valori posibile. Spre deosebire de combo box, nu mai exista posibilitatea afisarii unei liste derulabile; componenta afiseaza numai valoarea curenta, utilizatorul putandu-se deplasa inainte/inapoi prin lista de valori. Din acest motiv este recomandabil ca *JSpinner* sa fie folosit atunci cand succesiunea valorilor este una evidenta (numere, date calendaristice etc).

Din punct de vedere tehnic, un spinner se compune din:



- doua butoane, folosite pentru trecerea la valoarea urmatoare/anteroioara
- un *editor* ce afiseaza valoarea curenta; componenta implicita este un *JFormattedTextField*, pentru a evita pe cat posibil valorile invalide la introducerea manuala. Nu mai exista renderer - valoarea curenta este afisata de catre componenta de tip editor

Datele sunt memorate in modelul atasat componentei, a carui clasa trebuie sa implementeze interfata *SpinnerModel*.

Metode de interes JSpinner	Descriere
Object getValue() void setValue(Object)	obtinerea sau stabilirea valorii curente a JSpinner-ului. Incercarea de a seta o valoare in afara domeniului valid va genera o eroare (mai exact un <i>IllegalArgumentException</i>)
Object getNextValue() Object getPreviousValue()	obtinerea valorii urmatoare/precedente
JComponent getEditor() void setEditor(JComponent)	extragerea sau stabilirea componentei ce joaca rolul de editor pentru spinner
void commitEdit()	introduce valoarea din editor in model (echivalentul apasarii tastei Enter de catre user)

7.3.6.1. Evenimente specifice

Componentele *JSpinner* genereaza un singur tip de eveniment propriu lor: *ChangeEvent*, produs ori de cate ori valoarea curenta se schimba; acest lucru se intampla fie la actionarea butoanelor de trecere la valoarea precedenta/urmatoare, fie la incheierea editarii manuale a valorii (de obicei, apasand tasta Enter). Listenerii trebuie sa implementeze interfata *ChangeListener*, care contine unica metoda *void valueChanged(ChangeEvent)*.

```
final JSpinner sp = new JSpinner();
sp.addChangeListener(new ChangeListener() {
    public void valueChanged(ChangeEvent e) {
        System.out.println("Valoarea curenta este acum "+sp.getValue());
    }
});
```

7.3.6.2. Modele si editoarele atasate

JSpinner foloseste un model pentru date ce trebuie sa implementeze interfata *SpinnerModel*. Aceasta impune urmatoarele metode:

- **void addChangeListener(ChangeListener)** si **void removeChangeListener(ChangeListener)** - folosite pentru managementul listei de listeneri
- **void setValue(Object)** si **Object getValue()** - pentru setarea, respectiv extragerea valorii curente
- **Object getNextValue()** si **Object getPreviousValue(Object)** - returneaza valoarea urmatoare, respectiv precedenta din succesiune

In cazul lui *JSpinner*, *editor*-ul este mult mai dependent de model decat la alte componente, deoarece eventuala valoare introdusa manual trebuie neaparat sa faca parte din domeniul afisat de spinner. De aceea, la schimbarea modelului (si deci a naturii/regulilor interne ale datelor afisate) este necesara si adaptarea regulilor editorului la noul tip de date. Spre

exemplu, daca spinner-ul afiseaza valori numerice, atunci editor-ul va permite numai introducerea de numere; daca in schimb sunt afisate date calendaristice, formatul permis de editor trebuie si el schimbat.

Variantele programatorului de a stabili modelul de date al unui *JSpinner* sunt urmatoarele:

- crearea unei clase care implementeaza interfata *SpinnerModel* - varianta cea mai flexibila dar si cea mai laborioasa
- crearea unei clase care extinde *AbstractSpinnerModel* - ca si alte clase abstracte cu acelasi rol, aceasta rezolva managementul listenerilor, lasand in grija programatorului numai rescrierea metodelor ce tin de obtinerea si setarea valorii curente
- utilizarea uneia dintre clasele model deja disponibile. Swing ofera urmatoarele trei clase predefinite, pentru fiecare dintre ele existand si un editor aferent:
 - **SpinnerNumberModel** - model pentru cazul in care spinner-ul afiseaza o succesiune de numere. Modelul dispune de metode ce permit alegerea valorii minime si maxime si a pasului intre doua valori consecutive (ex: daca dorim ca spinner-ul sa "numere" din 2 in 2). Editor-ul aferent este *JSpinner.NumberEditor* (clasa interioara a lui *JSpinner*)
 - **SpinnerDateModel** - model pentru afisarea de informatii calendaristice. Clasa permite stabilirea limitelor (data minima/maxima), a pasului intre doua valori consecutive, si a campului de calendar ce face subiectul modificarii (ex: la actionarea butoanelor spinner-ului poate fi modificat campul zi, luna, an etc). Editor-ul aferent este *JSpinner.DateEditor*
 - **SpinnerListModel** - model care afiseaza o succesiune de valori preluata dintr-o colectie de tip *List*. Modelul ofera metode pentru setarea listei dorite. Editor-ul aferent este *JSpinner.ListEditor*

Editorul folosit este decis in functie de tipul modelului; pentru fiecare dintre cele trei enumerate anterior se foloseste editorul aferent, iar pentru alte modele se utilizeaza implicit *JSpinner.DefaultEditor*.

Clasa	Metode specifice de interes (suplimentare fata de cele din <i>SpinnerModel</i>)	Descriere
SpinnerNumberModel	SpinnerNumberModel(valoare, minim, maxim, pas)	creaza un model cu parametrii specificati. Constructorul este supraincarcat; tipurile de date ale celor 4 argumente pot fi int, double sau Comparable/Number
	void setMinimum(Comparable) / Comparable getMinimum() void setMaximum(Comparable) / Comparable getMaximum() void setStepSize(Number) / Number getStepSize()	obtinere/setare valoare minima, maxima si pas intre doua valori consecutive
SpinnerDateModel	SpinnerDateModel(Date valoare, Comparable dataInceput, Comparable dataFinal, int campCalendar)	constructor ce stabileste toti parametrii necesari: datele de inceput si de final, campul de calendar modificat si valoarea curenta
	void setStart(Comparable dataStart) / Comparable getStart() void setEnd(Comparable dataFinal) / Comparable getEnd() void setCalendarField(int)/int getCalendarField()	extragere/ stabilire parametri. Campul de calendar este specificat sub forma unei constante din clasa Calendar: YEAR, MONTH, DAY_OF_MONTH etc
SpinnerListModel	SpinnerListModel(List) SpinnerListModel(Object[])	constructori ce initializeaza datele din model cu cele preluate dintr-o colectie de tip List sau dintr-un tablou de obiecte
	void setList(List) / List getList()	extragerea sau setarea listei de elemente parcuse

7.3.7. Utilizarea componentelor de tip *JTable*

7.3.7.1. Caracteristici si principii de functionare

JTable este una dintre cele mai complexe componente Swing. Ea permite afisarea tabulara a informatiilor si editarea lor la nivel de celula, putandu-se personaliza aproape orice aspect al functionarii componentei. Cateva exemple:

- se poate stabili la nivel de fiecare celula daca celula este editabila sau nu
- tabelul poate fi configurat sa permita selectia de celule individuale sau numai de linii/coloane integrale
- tabelul poate permite redimensionarea si schimbarea ordinii coloanelor prin drag'n'drop

Componenta *JTable* se foloseste de nu mai putin de 3 modele:

- modelul de date, care trebuie sa implementeze interfata *TableModel*. Programatorul poate alege sa foloseasca constructorul *JTable(Object[][] date, Object[] numeColoane)*, poate crea o clasa care extinde *AbstractTableModel* sau poate folosi direct o instanta a clasei *DefaultTableModel*

- modelul de selectie, care trebuie sa implementeze interfata *ListSelectionModel*. Modelul implicit este un *DefaultListSelectionModel*
- modelul pentru coloane, care implementeaza interfata *TableColumnModel*. Fiecare coloana este reprezentata sub forma unui obiect *TableColumn*, ce dispune de propriul header (antet al coloanei), renderer de celule, editor de celule, renderer pentru header etc. Modelul implicit este un *DefaultTableColumnModel*

Renderer-ul trebuie sa implementeze *TableCellRenderer*, iar renderer-ul implicit este un *DefaultTableCellRenderer*.

Renderer-ul se poate stabili in doua moduri:

- in functie de coloana. Fiecare *TableColumn* are atasate un renderer si un editor de celula
- in functie de natura datelor obtinute din model. Aceasta este solutia implicita pentru cazul in care nu a fost definit renderer per coloana. Exista in acest scop metode precum *setDefaultRenderer()*

Editor-ul trebuie sa implementeze *TableCellEditor* sau sa extinda *AbstractCellEditor* , iar editor-ul implicit este un *DefaultCellEditor* .

Metode de interes JTable	Descriere
JTable(Object[][] date, Object[] numeColoane) JTable(Vector date, Vector numeColoane)	crearea unui JTable ce afiseaza informatia dintr-un tablou bidimensional de Object sau dintr-o colectie. Constructorul creeaza automat un model intern, read-only
void changeSelection(int linie, int coloana, boolean comutare, boolean extindere) void clearSelection() void selectAll() boolean isCellSelected(int linie, int coloana) boolean isColumnSelected(int coloana) boolean isRowSelected(int rand)	managementul selectiei
boolean isCellEditable(int linie, int coloana) boolean editCellAt(int linie, int coloana)	managementul editarii la nivel de celula
TableCellEditor getCellEditor(int linie, int coloana) void setCellEditor(TableCellEditor) TableCellRenderer getCellRenderer(int linie, int coloana) void setDefaultRenderer(TableCellRenderer)	stabilire/extragere editor si renderer
Object getValueAt(int linie, int coloana) void setValueAt(Object, int linie, int coloana)	obtinere/setare a valorii unei celule
void setModel(TableModel) / TableModel getModel() void setColumnModel(TableColumnModel) / TableColumnModel getColumnModel() void setSelectionModel(ListSelectionModel) / ListSelectionModel getSelectionModel()	<ul style="list-style-type: none"> - stabilire/extragere model de date - stabilire/extragere model de coloane - stabilire/extragere model de selectie

7.3.7.2. Evenimente specifice

O prima particularitate a *JTable*-ului este ca principalele evenimente specifice nu sunt generate de catre *JTable*-ul insusi, ci de catre modelele sale. In aceste conditii, **listenerii nu trebuie atasati JTable-ului, ci modelului care genereaza evenimentul!**

Evenimentele specifice generate sunt urmatoarele:

- modelul de date genereaza evenimente de tip *TableModelEvent* la modificarea informatiei. Listenerii aferenti trebuie sa implementeze interfata *TableModelListener*
- modelul de selectie genereaza evenimente de tip *ListSelectionEvent* la modificarea setului de celule selectate. Listenerii trebuie sa implementeze *ListSelectionListener*
- modelul de coloane genereaza evenimente de tip *ColumnModelEvent* la adaugarea, stergerea sau schimbarea ordinii coloanelor. Listenerii trebuie sa implementeze *TableColumnModelListener*

Clasa	Metode de interes
TableModelListener	tableChanged(TableModelEvent)
ListSelectionListener	valueChanged(ListSelectionEvent)
TableColumnModelListener	<ul style="list-style-type: none"> columnAdded() columnMarginChanged() columnMoved() columnRemoved() columnSelectionChanged()

```
JTable tabel = new JTable(2,3); // tabel gol cu 2 linii si 3 coloane
ListSelectionModel sm = tabel.getSelectionModel();
sm.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent evt) {
        if(!evt.getValueIsAdjusting())
            System.out.println("Coordonate celula selectata: "+
                tabel.getSelectedRow() + "-" + tabel.getSelectedColumn());
    }
});
tm.addTableModelListener(new TableModelListener() {
    public void tableChanged(TableModelEvent evt) {
        System.out.println("S-a modificat celula: "+
            tabel.getSelectedRow() + "-" + tabel.getSelectedColumn());
    }
});
```

7.3.8. Crearea modelului de date

Modelul de date al unui *JTable* trebuie sa implementeze interfata *TableModel*, ce impune urmatoarele metode:

- **void addTableModelListener(TableModelListener)** si **void removeTableModelListener(TableModelListener)** - necesare pentru managementul listenerilor
- **int getRowCount()** si **int getColumnCount()** - stabilesc numarul de linii, respectiv de coloane ale tabelului
- **Object getValueAt(int linie, int coloana)** si **void setValueAt(Object, int linie, int coloana)** - citirea si setarea valorii unei celule
- **boolean isCellEditable(int linie, int coloana)** - stabileste daca este permisa editarea celulei desemnate de acele coordonate
- **String getColumnName(int coloana)** - determina numele (titlul) coloanei pe baza pozitiei acestia in tabel
- **Class getColumnClass(int pozitie)** - determina natura datelor coloanei, in functie de care se alege renderer-ul si editor-ul implicit

Variantele de a obtine un model de *JTable* sunt aceleasi ca pana acum:

- initializarea continutului afisat de tabel cu ajutorul unui tablou bidimensional de *Object* sau al unei colectii; datele trebuie insotite de inca un tablou care contine numele coloanelor. Constructor de interes: *JTable(Object[][] date, Object[] numeColoane)*
- clasa *DefaultTableModel*, ce ofera un model modificabil, la cheie, cu API-ul prezentat mai jos
- clasa ce extinde *AbstractTableModel*
- clasa care implementeaza direct *TableModel*, daca niciuna dintre variantele de mai sus nu satisface

Metode de interes <i>DefaultTableModel</i> (suplimentare fata de cele ale interfetei <i>TableModel</i>)	Descriere
<i>DefaultTableModel(Object[][] date, Object[] numeColoane)</i> <i>void setDataVector(Object[][] date, Object[] numeColoane)</i>	stabilirea la instantiere sau ulterior a datelor afisate prin preluarea lor dintr-un tablou bidimensional de <i>Object</i>
<i>void addRow(Object[] date)</i> <i>void insertRow(int pozitie, Object[] date)</i> <i>void addColumn(Object numeColoana, Object[] date)</i> <i>moveRow(int randInceput, int randFinal, int pozitieTinta)</i>	<ul style="list-style-type: none"> - adaugare de rand la sfarsit - introducerea unui rand pe pozitia dorita - adaugare de coloana noua - mutare rand sau succesiune de randuri la pozitia dorita

```
// afisare rapida date in JTable, read-only
Object[][] produse = {{"mere",5}, {"struguri",10}};
Object[] coloane = {"Denumire", "Pret"};
JTable t = new JTable(produse, coloane);

// creare model modificabil
DefaultTableModel m = new DefaultTableModel(produse, coloane);
t.setModel(m);
m.addRow(new String[]{"nectarine", 15}); // tabelul se va actualiza automat
```

Pentru un exemplu mai elaborat, care creeaza un model de *JTable* prin extinderea clasei *AbstractTableModel*, consultati anexa de exemple a materialului.

7.4. BIBLIOGRAFIE

- Sistemul de evenimente:
 - Evenimente suportate de componentele Swing:
<http://docs.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html>
 - API-ul interfețelor listener (tabel): <http://docs.oracle.com/javase/tutorial/uiswing/events/api.html>
 - Tratarea evenimentelor: <http://docs.oracle.com/javase/tutorial/uiswing/events/index.html>
- Arhitectura componentelor Swing: <http://java.sun.com/products/jfc/tsc/articles/architecture/>

7.5. ANEXA 1 - clase model pentru componentele Swing complexe

Componentă	Tipuri de model utilizate	Interfete model	Clase abstracte ajutătoare	Clase concrete "la cheie"
JList	date	ListModel	AbstractListModel	DefaultListModel
	selectie	ListSelectionModel	-	DefaultListSelectionModel
JComboBox	date	ComboBoxModel MutableComboBoxModel	AbstractListModel (nu există una dedicată pentru combo box)	DefaultComboBoxModel
JSpinner	date	SpinnerModel	AbstractSpinnerModel	SpinnerNumberModel SpinnerDateModel SpinnerListModel
JTable	date	TableModel	AbstractTableModel	DefaultTableModel
	selectie	ListSelectionModel	-	DefaultListSelectionModel
	coloane	TableColumnModel	-	DefaultTableColumnModel
JTree	date	TreeModel	-	DefaultTreeModel
	selectie	TreeSelectionModel	-	DefaultTreeSelectionModel

7.6. ANEXA 2 - Exemple de cod aditionale

7.6.1. Model de JComboBox personalizat

Codul listat mai jos implementează un model de *JComboBox* care permite modificarea liste de elemente. Clasa extinde *AbstractListModel* (pentru a profita de infrastructura de listeneri deja scrisă acolo) și implementează *MutableComboBoxModel*.

```
class ModelCombo extends AbstractListModel implements MutableComboBoxModel {

    private String[] elemente = new String[0];
    private String elementSelectat = null;

    public int getSize() { return elemente.length; }
    public Object getElementAt(int pozitie) {
        System.out.println("getelementat(" + pozitie + ")");
        return elemente[pozitie];
    }

    public void addElement(Object element) {
        System.out.println("addelement");
        elemente = Arrays.copyOf(elemente, elemente.length + 1);
        elemente[elemente.length - 1] = element.toString();
        fireContentsChanged(this, -1, -1);
    }

    public int indexOf(Object element) {
        for (int i = 0; i < elemente.length; i++) {
            if (elemente[i].equals(element.toString())) {
                return i;
            }
        }
    }
}
```

```

        return -1;
    }

    public void removeElement(Object element) {
        int pozitie = indexOf(element);
        removeElementAt(pozitie);
    }

    public void insertElementAt(Object element, int pozitie) {
        String[] elemente_tmp = Arrays.copyOf(elemente, elemente.length + 1);
        System.arraycopy(elemente, pozitie, elemente_tmp, pozitie + 1, elemente.length - pozitie
                           + 1);
        elemente_tmp[pozitie] = element.toString();
        elemente = elemente_tmp;
    }

    public void removeElementAt(int pozitie) {
        String[] elemente_tmp = Arrays.copyOf(elemente, elemente.length - 1);
        System.arraycopy(elemente, pozitie + 1, elemente_tmp, pozitie, elemente.length - pozitie);
        elemente = elemente_tmp;
        fireContentsChanged(this, -1, -1);
    }

    public void setSelectedItem(Object element) {
        elementSelectat = element.toString();
    }

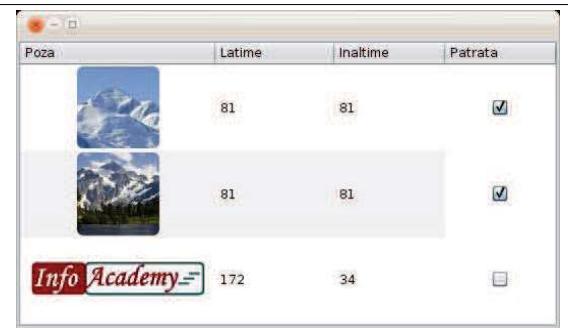
    public Object getSelectedItem() {
        return elementSelectat;
    }
}

// ...iar pentru a adauga in lista combo box-ului elementul din editor la apasarea tastei Enter:
final JComboBox combo = new JComboBox(new ComboModel());
combo.setEditable(true);
combo.addActionListener(new ActionListener(){
    private void comboActionPerformed(java.awt.event.ActionEvent evt) {
        if (evt.getActionCommand().equals("comboBoxEdited")) {
            String element = combo.getSelectedItem().toString();
            if (model.indexOf(element) < 0) {
                model.addElement(element);
            }
        }
    }
});
});
```

7.6.2. Model de JTable personalizat

Clasa de mai jos implementeaza un model de tabel care memoreaza o lista de icon-uri (obiecte de tip ImageIcon) si ii transmite tabelului caracteristicile acestora: poza insasi, dimensiunile ei si daca poza este sau nu patrata.

```
class TModel extends AbstractTableModel{  
    ImageIcon[] poze = new ImageIcon[0];  
  
    public int getRowCount() { return poze.length; }  
  
    public int getColumnCount() {  
        return 4;  
    }  
  
    public Object getValueAt(int linie, int coloana) {  
        ImageIcon poza = poze[linie];  
        switch(coloana){  
            case 0: return poza;  
            case 1: return poza.getIconWidth();  
            case 2: return poza.getIconHeight();  
        }  
    }  
}
```



rev. 1643

```
        case 3: return poza.getIconHeight()==poza.getIconWidth();
        default: return "ERROR";
    }
}

public void adaugaPoza(ImageIcon i){
    poze = Arrays.copyOf(poze,poze.length+1);
    poze[poze.length-1] = i;
    fireTableRowsInserted(poze.length-1, poze.length-1);
}

public String getColumnName(int coloana) {
    return new String[]{"Poza","Latime","Inaltime","Patrata"}[coloana];
}

public Class getColumnClass(int coloana) {
    switch(coloana){
        case 0: return Icon.class;
        case 3: return Boolean.class;
        default: return String.class;
    }
}

// ...iar pentru a seta acest model pentru un JTable:
JTable t = new JTable();
TModel m = new TModel();
t.setModel(m);
// adaugarea unei poze aflata in subpachetul poze al pachetului curent
m.adaugaPoza(new ImageIcon(getClass().getResource("poze/pic1.jpg")));
// ca urmare, tabelul se actualizeaza automat gratie apelarii metodei fireTableRowsInserted()
```