Programmeren II

Daan Pape
Universiteit Gent

18 juni 2012

1 Constanten

Volgende constanten zijn belangrijk:

Klasse	Constante	Omschrijving
`JFrame`	`EXIT_ON_CLOSE`	Het programma wordt gesloten bij klikken op het kruisje
	`DISPOSE_ON_CLOSE`	Het venster wordt automatisch verwijderd nadat de `windowClosing` is verwerkt.
	`DO_NOTHING_ON_CLOSE`	Er gebeurt niets als je op het kruisje klikt, behalve een `windowClosing` event
	`HIDE_ON_CLOSE`	Het venster wordt verborgen, het programma blijft lopen
`SwingConstants`	`CENTER`	Plaatst componenten gecentreerd tegenover elkaar
	`BOTTOM`	Plaatst componenten onder elkaar
`KeyEvent`	`VK_X`	Staat voor de toets `X`
`ListSelectionModel`	`MULTIPLE_INTERVAL_SELECTION`	Je kan in een lijst nu meerdere intervallen selecteren
	`SINGLE_INTERVAL_SELECTION`	Je kan in een lijst nu maar één interval selecteren
	`SINGLE_SELECTION`	Je kan in een lijst nu maar één item selecteren
`JComponent`	`WHEN_FOCUSED`	De gefocuste staat van het component. Wordt bij `InputMap` gebruikt.
`MouseEvent`	`BUTTON1`	De eerste muisknop is ingedrukt
	`BUTTON2`	De tweede muisknop is ingedrukt
	`BUTTON3`	De derde muisknop is ingedrukt
`Cursor`	`HAND_CURSOR`	De hand cursor
`JOptionPane`	`INFORMATION_MESSAGE`	Een informatiebericht
	`WARNING_MESSAGE`	Een waarschuwing
	`ERROR_MESSAGE`	Een foutbericht
	`QUESTION_MESSAGE`	Een vraag
	`PLAIN_MESSAGE`	Een gewoon bericht

2 Methodes

Klasse	Methode	Parameters	Return	Omschrijving
`JFrame`	`GetContentPane`		`Container`	Geeft het contentpane van een venster terug
`JButton`	`set...TextPosition`	`(int position)`	`void`	Meet een `SwingConstant` wordt de plaats van
`JMenu`	`addSeparator`		`void`	Plaatst een scheidingslijn tussen menu items
`KeyStroke`	`getKeyStroke`	`(String keystroke)`	`KeyStroke`	Als parameter worden de toetsen opgegeven zoals `"ctrl o"`
`JList`	`getValueIsAdjusting`		`boolean`	Geeft `true` terug als de verandering in de selectie nog bezig is, de muisknop werd nog niet losgelaten
`JTextArea`	`insert`	`(String str, int pos)`	`null`	Voegt de tekst tussen vanaf de opgegeven positie
`Component`	`requestFocusInWindow`		`null`	Vraagt focus op dit component aan als het venster waarin dit wordt weergegeven al focus heeft.
`FocusEvent`	`isTemporary`		`boolean`	Geeft true terug als deze focus verandering maar tijdelijk is.

Klasse	Methode	Parameters	Return	Omschrijving
`Property`	`setProperty`	`(String key, String value)`	`String`	Als de key bestaat wordt een nieuwe waarde toegekend, anders wordt er een nieuwe entry gemaakt. De methode geeft de vorige waarde van de sleutel terug
	`getProperty`	`(String key)`	`String`	Geeft de waarde terug behorende bij de sleutel
	`store`	`(OutputStream o, String c)`	`void`	Het `property` object wordt uitgeschreven met datum en commentaar.
`Element`	`getChildren`		`Object[]`	Geeft alle kinderen van dit object terug
	`getChildren`	`(String name)`	`Object[]`	Geeft alle kinderen met de opgegeven naam terug
	`getAttributeValue`	`(String name)`	`String`	Geeft, indien het bestaat, de waarde van de attribute terug
`Graphics`	`fillRect`	`(int x, int y, int w, int h)`	`void`	Tekent een rechthoek met hoogte h en breedte w met linkerbovenhoek in x,y
	`fillOval`	`(x, y, w, h)`	`void`	Tekent een ovaal met hoogt h en breedte w met linkerbovenhoek in x,y
	`fillArc`	`(x, y, w, h, s, a)`	`void`	Tekent een boog met hoogt h en breedte w met linkerbovenhoek in x,y. Dit met startpunt s en hoek a.
`JPanel`	`setCursor`	`(Cursor cursor)`	`void`	Stelt de cursor in

3 Constructors

Contstructor	Parameters	Omschrijving
`Timer`	`(int delay, ActionListener listener)`	De tijd is de pauze waartussen de events worden gegenereerd
`JTextField`	`(int columns)`	Het aantal kolommen (lettertekens) die het tekstveld afbeeldt, het kan er wel meer bevatten
	`(String text, int columns)`	Breedte en standaardtekst
`JTextArea`	`(int rows, int columns)`	Het aantal lijnen en het aantal tekens per lijn
`JFormattedTextField`	`(Object value)`	Als aanvaard type wordt het type van het object gebruikt
`AbstractAction`	`(String name, Icon icon)`	Een abstracte knopactie met opschrijft name en icoon icon
`JDialog`	`(Frame/Dialog, String title)`	Maakt een niet modaal dialoogvenster
	`(Frame/Dialog p, String s, true)`	Maakt een modaal dialoogvenster
`JProgressBar`	`(int min, int max)`	Maakt een progressbar tussen min en max
`ProgressMonitor`	`(Component p, Object m, String n, int mi, int ma)`	maakt een progressmonitor

4 Weetjes

Een `Swing` component aanvaardt als opschrift html code, de code `<html><b>vet</b></html>` bijvoorbeeld geeft de tekst op een component in het vet weer. Als je `ToggleButtons` in een

`ButtonGroup` steekt kan er maar 1 ingedrukt worden.

De `Timer` klasse genereerd om de x aantal seconden een event die door een `ActionListener` wordt opgevangen. Een timer wordt als volgt gestart: `new Timer(1000, listener).start();` Let ervoor op om `javax.swing.Timer` en niet `java.util.Timer` te gebruiken daar deze laatste vooral voor niet-grafische toepassingen dient.

Als in een `JList` iets wordt geselecteerd wordt een `ListSelectionEvent` gegooit welke door een `ListSelectionListener` met de methode `valueChanged` wordt opgevangen.

Swing componenten bezitten standaard geen scrollbars, om deze functionaliteit toe te voegen moeten ze namelijk op een `JScrollPane` worden geplaatst.

Het inlasteken of *caret* bestaat in Java twee delen:

1. De **punt** of **dot**: het zichtbare merkteken, zoals de knipperende horizontale lijn.
2. Het **merkteken** of **mark**: het niet zichtbare teken, maar alles tussen deze mark en de dot wordt als geselecteerd beschouwd.

Een object van de klasse `JTextComponent` bevat nu twee methodes:

1. `setCaretPosition(int pos)`: verplaatst de dot en de mark naar de opgegeven positie.
2. `moveCaretPosition(int pos)`: verplaatst enkel de dot naar de opgegeven positie en laat dus toe tekst te selecteren.

Met een `FocusListener` wordt naar `FocusEvent`'s geluisterd, deze interface bevat twee methodes: `focusGained` en `focusLost`. Aan een tekstveld kan nu een `InputVerifier` gegeven worden met de methodes `verify(JComponent comp)` en `shouldYieldFocus(Jcomponent comp)`. De eerste methode moet `true` teruggeven bij geldige invoer, de tweede moet actie ondernemen bij foute invoer.

Met een `JFormattedTextField` kan geen verkeerde invoer worden uitgelezen. Om de waarden op te vragen of in te stellen worden hier de methodes `getValue` en `setValue` gebruikt die een `Object` teruggeven of aanvaarden van het juiste type. Het type wordt opgegeven in de constructor zoals `Number` en `Date`. Er kan nu echter nog steeds een verkeerde waarde worden ingegeven. Om dit te voorkomen kan men weer van een `inputVerifier` gebruik maken. Men moet er nu op letten dat de waarde die `getValue` teruggeeft niet overeenstemt met de nieuw ingevulde waarde in het tekstvak. Om dit expliciet te laten gebeuren moet de methode `commitEdit` opgeroepen worden. Deze methode zal nu een `ParseException` gooien als de waarde en de tekst niet overeenstemmen.

`AbstractAction`'s worden veel gebruikt voor allerlei knoppen. Ze worden echter ook gebruikt voor sneltoetsen. Dit gebeurt volgens een speciaal mechanisme. Elke `Component` heeft een `ActionMap` en meerdere `InputMap`'s:

- `InputMap`: deze mapt `KeyStroke`'s op `String`'s. Elke `InputMap` heeft een parent waarin gezocht wordt naar sleutels als deze niet in de huidige `InputMap` staan. Standaard heeft elke component een `InputMap` waneer gefocust, een kind de focus heeft of waneer een andere component in het zelfde venster focus heeft. Je kan ook zelf een `InputMap` maken, je stelt wel best een parent in.

- `ActionMap`: deze mapt `String`'s op `Action`'s.

Bij een toetsaanslag wordt in de `InputMap` gezocht naar een naam voor de actie en dan wordt de actie opgezocht in de `ActionMap` waarbij vervolgens de `actionPerformed` methode wordt opgeroepen.

De `Reader` en `Writer` klassen werken karakter georiënteerd terwijl de `InputStream` en `OutputStream` klassen byte georiënteerd werken. Een goede programmeerstijl is om tekstuele data altijd met `Reader`s of `Writer`s te behandelen. Om een willekeurige `InputStream` te lezen wordt een `InputStreamReader` gebruikt die de `InputStream` naar een `Reader` omzet. Hierbij moet wel de juiste karaktercodering worden opgegeven als `String`. Veel gebruikt is `LATIN1` wat gelijk is aan `ISO-8859-1`

Men kan het zelfde doen om van een `OutputStream` een `OutputStreamWriter` te maken maar meestal gebruikt men direct een `PrintWriter` waardoor de methodes `print` en `println` beschikbaar worden. Een stream die gegevens inleest of uitschrijft van of naar een andere stream is een `FilterStream`.

Properties bestanden worden door middel van een `Properties` object voorgesteld. De bestanden bestaan uit key-value pairs waarbij commentaar met een hekje wordt aangeduidt en lege lijnen worden overgeslaan. Met de methode `getProperty(String key)` kan de waarde van die property worden opgevraagd. Met de methode `load(InputStream strm)` kan het bestand ingeladen worden.

Met behulp van een `ResourceBundle` kan een programma geïnternationaliseerd worden. Men maakt eerst een standaard properties bestand aan en dan per taalcode een nieuw bestand. De resourcebundle gaat nu aan de hand van de bestandsnaam van het standaardbestand en de landcode van de computer het juiste propertiesbestand laden. Dit gebeurt door
`ResourceBundle bundle = ResourceBundle.getBundle("class path")`. De waarden worden nu met de `getString("key")` methode opgevraagd. De namen van de propertiebestanden bestaan uit `naam_taal_land_extra.properties`. Zo staat `bestand_nl_BE_UNIX.properties` voor een nederlandstalige UNIX gebruikende Belg. De keuze kan met een
`Locale(String taal, String land, String extra)` worden geforceerd.

XML kan op vele manieren ingelezen worden, ook met JDOM waarbij het bestand door een `Document` klasse wordt voorgesteld. Een document bestaat dan uit `Element`'s waarin kinderen zitten die weer elementen zijn. `Document doc = new SAXBuilder().build(input)` laadt een document in en het rootElement kan met `doc.getRootElement()` worden opgevraagd.

Databasefunctionaliteit wordt in java door de JDBC-API (Java DataBase Connectivity) verzorgt. Men moet, afhankelijk van de gebruikte databank, wel de juiste driver inladen. Dit gebeurt vaak in een static block met de methode `Class.forName(String driver)` welke een `ClassNotFoundException e` kan opgooien. Meerbepaald registreerd deze methode een instantiatie van de driver bij de DriverManager, men kan dus perfect meerdere drivers laden om met verschillende databanken te werken. Er kan nu met `DriverManager.getConnection(jdbc_url, login, pass)` een connectie worden gemaakt. de JDBC-URL bestaat uit `protocol:subprotocol:subname` waarbij het protocol `jdbc` is, het subprotocol de gebruikte driver en de rest is door de producent van de databank bepaald. Voor de java databank is dit bv. `jdbc:derby://server/naamdatabank`.

Het is zeer belangrijk dat een connectie gesloten wordt na gebruik, de `close` methode kan echter

ook een `SQLException` opgooien. Opdrachten worden met `Statement`'s en `PreparedStatement`'s uitgevoerd. Ook deze moeten gesloten worden. Data wordt teruggegeven in een `ResultSet` object wat ook moet worden afgesloten.

Om zelf te tekenen wordt veelal een extensie op `JPanel` gemaakt en de `paintComponent(Graphics g)` wordt overschreven. Het is belangrijk om wel steeds de originele `paintComponent` uit te voeren (dit in verband met randen, ...). De `fill` methodes zullen steeds 1 pixel kleiner zijn dan de `draw` methoden.

`Icon` is een interface met 3 methoden:

1. `int getIconWidth()`: geeft de breedte in pixels van het icoon
2. `int getIconHeight()`: geeft de hoogte in pixels van het icoon
3. `int paintIcon(Component c, Graphics g, int x, int y)`: tekent het icoon op `g` met linkerbovenhoek in x,y met parent c.

Interacties met de muis zijn mogelijk met een `MouseListener` die naar de muisknoppen luistert en een `MouseMotionListener` die naar de beweging kijkt. Beide luisteren ze naar `MouseEvent`'s. Er bestaat ook een interface `MouseInputListener` die beide interfaces combineert.

De klasse `JOptionPane` bied verschillende standaarddialogen aan die met volgende methodes kunnen worden weergegeven:

1. `showMessageDialog`: toont een dialoogvenster met een bericht en daaronder een OK-knop waarmee het venster wordt afgesloten.
2. `showConfirmDialog`: een dialoogvenster met een bericht en een Yes- en No- knop.
3. `showOptionDialog`: een bericht met daaronder een aantal knoppen en een opschrift naar keuze.
4. `showInputDialog`: een bericht en één tekstveld of combobox. Er is echter geen flexibiliteit, zo kan men de invoer niet op juistheid controleren.

Deze hebben vrij lange constructors:

1. ```
   showMessageDialog(Component parent,
                     Object message,
                     String title,
                     int type,
                     Icon icon);
   ```
2. ```
   showOptionDialog(Component parent,
                    Object message,
                    String title,
                    int optionType,
                    int type,
                    Icon icon,
                    Object[] options,
                    Object defaultOption);
   ```

Van zodra de `options` parameter niet `null` is wordt het optie type genegeerd, maar dit moet wel geldig zijn. Daarom wordt hier bijna altijd voor `JOptionPane.DEFAULT_OPTION` gekozen.

Een venster genereerd verschillende `WindowEvent`'s die met een `WindowListener` worden opgevangen. De `WindowListener` bevat volgende methoden:

1. `windowOpened(WindowEvent e)`: als het venster voor de eerste keer wordt weergegeven.
2. `windowIconified(WindowEvent e)`: als het venser wordt geminimaliseerd.
3. `windowDeiconified(WindowEvent e)`: als het venster terug wordt vergroot.
4. `windowActivated(WindowEvent e)`: als het venster focus krijgt.
5. `windowDeactivated(WindowEvent e)`: als het venster focus verliest.
6. `windowClosing(WindowEvent e)`: als de gebruiker het venster probeert te sluiten, kan nog worden gestopt.
7. `windowClosed(WindowEvent e)`: als het venster effectief wordt gesloten, niet meer te stoppen.

De `windowClosing` gebeurtenis wordt enkel gegenereerd indien de gebruiker het venster sluit. Indien men het via `window.dispose()` of `window.setVisible(false)` deed niet.

Een lijst wordt in java door 3 klassen voorgesteld:

- `JList`: de view
- `ListModel`: model die de inhoud van de lijst bevat.
- `ListSelectionModel`: model die de selectiegegevens van de lijst bevat.

De interface `ListModel` bevat volgende methoden:

- `int getSize()`
- `Object getElementAt(int index)`
- `void addListDataListener(ListDataListener l)`
- `void removeListDataListener(ListDataListener l)`

Om het gemakkelijk te maken is er al een klasse `AbstractListModel` die reeds voor de juiste `fire`-methodes zorgt:

- `fireContentsChanged(Objecte source, int index0, int index1)`
- `fireIntervalAdded(Object source, int index0, int index1)`
- `fireIntervalRemoved(Object source, int index0, int index1)`

Opgelet de indexen zijn hier beide **inclusief**. Een lijst moet steeds alle elementen overlopen om zijn geprefereerde grootte te bepalen. Dit kan vermeden worden door de grootte met `setFixedCellHeight` en `setFixedCellWidth` of `setPrototypeCellValue` op te geven. Deze laatste is de geprefereerde oplossing.

Als men een dynamische lijst maakt kiest men best voor een `DefaultListModel`. Dit maakt men aan met een lege constructor en het gedraagt zich een beetje zoals een `List`. Zo kan men met de methode `addElement(Object element)` iets toevoegen. Zo is daar ook een `remove(int index)` methode. De geselecteerde index kan je aan de lijst met `getSelectedIndex()` vragen.

Als men met tabellen werkt moeten drie verschillende modellen in acht genomen worden:

- Gegevensmodel: de inhoud van de tabel maar ook de kolomtitels. We extenden meestal `AbstractTableModel` en moeten daarvoor de methodes `getRowCount`, `getColumnCount` en `getValueAt(int row, int column)` implementeren. Verder kan men er ook voor kiezen om `getColumnName(int column)`, `isCellEditable(int row, int column)` en `setValueAt(Object value, int row, int column)` te implementeren.
- Kolommodel: dit model houdt info bij over de breedte van de kolommen. We gebruiken altijd het autmatisch aangemaakte.
- Selectiemodel: dit is identiek aan dat van `JList`

Het `AbstractTableModel` bezit heel wat `fire`-methoden:

1. `fireTableCellUpdated(int row, int column)`
2. `fireTableRowsUpdated(int firstRow, int lastRow)`

3. `fireTableRowsInserted(int firstRow, int lastRow)`
4. `fireTableRowsDeleted(int firstRow, int lastRow)`
5. `fireTableDataChanged()` : de tabel moet volledig opnieuw worden afgebeeld
6. `fireTableStructureChanged()` : de structuur (bv. aantal kolommen) is veranderd

Een tabel kan meer dan `String`'s weergeven en past de weergave aan, aan het soort klasse dat moet worden weergegeven. met de methode `public Class getColumnClass(int column)` vraagt de lijst dit aan het model. In het `TableColumnModel` moeten enkel de kolombreedtes worden bijgehouden. Er zit in dit model een methode `getColumn` die een object van het type `TableColumn` teruggeeft waarop je dan de `setPreferredWidth` kan oproepen.

Een boomdiagram bestaat uit een view `JTree` en een model `TreeModel`. Het model genereerd `TreeModelEvent`'s die met een `TreeModelListener` worden opgevangen. Om een TreeModel te implementeren kan je ofwel van nul beginnen ofwel de klasse `DefaultTreeModel` uitbreiden. Een `TreeModel` bevat volgende methoden:

1. `addTreeModelListener(TreeModelListener l)`
2. `removeTreeModelListener(TreeModelListener l)`
3. `valueForPathChanged(TreePath path, Object newValue)`: dit kan van buitenaf worden aangroepen en het model moet dan al zijn luisteraars op de hoogte brengen.
4. `Object getRoot()`: geeft de wortel van de boom terug.
5. `int getChildCount(Object parent)`: vertelt hoeveel kinderen een bepaalde knoop heeft.
6. `Object getChild(Object parent, int index)`
7. `int getIndexOfChild(Object parent, Object child)`
8. `boolean isLeaf(Object node)`

Als we in een`JTree` de standaard `toString` methode van de objecten willen overriden zijn daar twee opties voor:

1. De methode `convertValueToText(Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)` te overschrijven.
2. Een eigen `CellRenderer` schrijven door de `DefaultTreeCellRenderer` te extenden. De enige methode die we moeten herschrijven is de `public Component getTreeCellRendererComponent (JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)`. Daar `DefaultTreeCellRenderer` een `JLabel` is bestaan alle methoden hiervan en kan je hiermee de tekst en icoon instellen. Uiteindelijk geeft deze methode de klasse terug.

Ook zonder eigen gegevensmodel kan een boom gemaakt worden, de klassestructuur zoals op figuur 1 moet dan in acht worden genomen. Meestal kiest men ervoor om `DefaultTreeModel` uit te breiden. De knopen moeten nu voldoen aan de `TreeNode` interface. Als knopen veranderlijk zijn moeten ze aan de interface `MutableTreeNode` voldoen en dit wordt meestal gedaan door de klasse `DefaultMutableTreeNode` te implementeren. Deze laatste klasse bevat een

`userObject` welke de data voorstelt. De constructor bevat twee parameters: `Object userObject` en `boolean allowsChildren`. De laatste is standaard `true`.

Selecties worden door een `TreeSelectionListener` opgevangen. Aan de boom kan men dan `getSelectionPath` vragen welke een `TreePath` teruggeeft. Dit object is het selectiepad en bevat het geselecteerde element tot de wortel. Met de methode `getLastPathComponent` kan je dan het effectief geselecteerde element opvragen.

Standaard lopen er in een Java Swing GUI twee `Thread`'s:

1. De hoofdprogramma thread: de `main` methode loopt hierin.
2. De gebeurtenis verwerkende thread: voert methodes zoals `actionPerformed` en `paintComponent` op.

Omdat Swing niet threadsafe is houden we als hoofdregel aan om alle GUI veranderingen vanuit de gebeurtenis verwerkende draad te doen, behoudens enkele uitzonderingen zoals repaint, luisteraars registreren/verwijderen. Daarom bied de klasse `EventQueue` de methode `invokeLater(Runnable runnable)` aan.

Een `JProgressBar(int min, int max)` toont een laadbalk. Indien `setStringPainted(true)` wordt opgeroepen wordt ook een percentage afgebeeldt. Met de methode `setValue` kan er voortgang worden weergegeven. Een `ProgressMonitor` is een afzonderlijke dialoog met daarin een progress bar. De constructor aanvaardt een `String` note die aangeeft wat er momenteel gebeurt. Een `ProgressMonitor` verschijnt pas op het scherm als een taak te lang duurt:

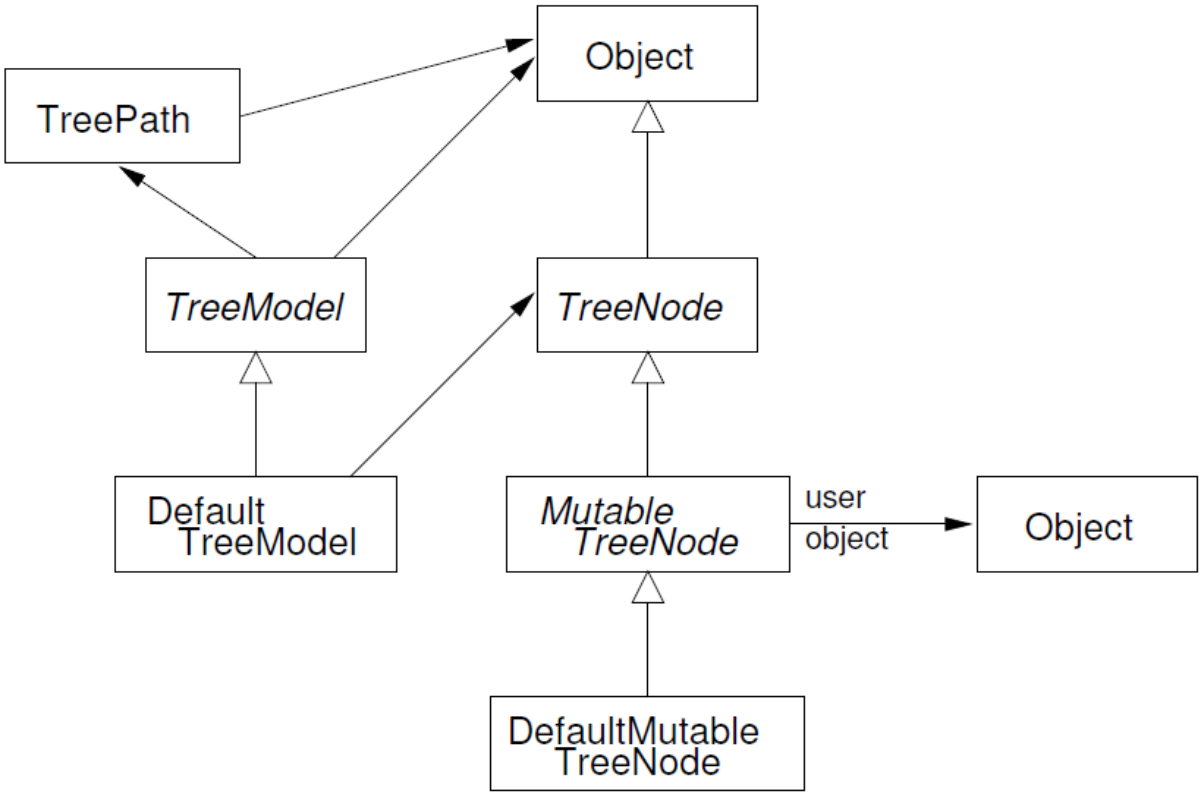
- `setMillisToDecideToPopup` : na deze tijd probeert de monitor te voorspellen hoe lang het nog zal duren.
- `setMillisToPopup` : als het langer dan dit zal duren wordt de monitor weergegeven.

Een `SwingWorker` maakt het zogezecht makkelijker om async dingen in te laden. De klasse moet uitgebreid worden en er zijn een aantal methoden te onthouden:

1. `doInBackground()`: dit is de taak die in de achtergrond moet worden uitgevoerd (de runnable).
2. `done()`: dit is de callback methode die in de gebeurtenisverwerkende draad wordt opgeroepen.
3. `publish(T())`: stuurt de objecten door (in een lijst) naar process dat in de gebeurtenis verwerkende draad gebeurt. Deze methode dient **niet** overschreven te worden.
4. `process()` aanvaard dingen van process en doet er iets mee.

Om de `SwingWorker` te starten wordt `execute()` gebruikt. Een Swingworker is generisch en bevat twee type argumenten. Het eerst duidt op het retourtype van `doInBackground()` en het tweede is het geen wat door publish en process wordt gebruikt. Er kan ook naar de `progress` eigenschap geluisterd worden met een `PropertyListener`. Telkens als men in de `doInBackground` de `setProgress([0,100])` methode oproept zal dit een `PropertyChangeEvent` genereren.

Het aanmaken van een `EnumMap` gebeurt anders dan verwacht. Men moet namelijk binnen de ronde haakjes nogmaals de `enum.class` schrijven.



Figuur 1: Het standaardmodel voor boomdiagrammen