Programmeren in Java — deel 2

Prof. K. Coolsaet

Universiteit Gent
Vakgroep Toegepaste Wiskunde en Informatica
2006–2007

Inhoud

Woord vooraf

Deze tekst bevat de cursusnota's bij het vak 'Programmeren 2' voor de 1e bachelor informatica aan de Universiteit Gent. Dit vak vormt een logisch vervolg op het vak 'Programmeren 1' uit het 1e semester.

Nu je de basisprincipes kent van het programmeren in Java, is het eindelijk tijd om op een meer professionele manier met deze taal om te springen. Je weet ondertussen dat een goed programma uit heel wat klassen bestaat en dat het niet altijd eenvoudig is om die onderling met elkaar te laten samenwerken. Het is met name heel belangrijk om voldoende tijd te besteden aan het ontwerp van de juiste klassenstructuur.

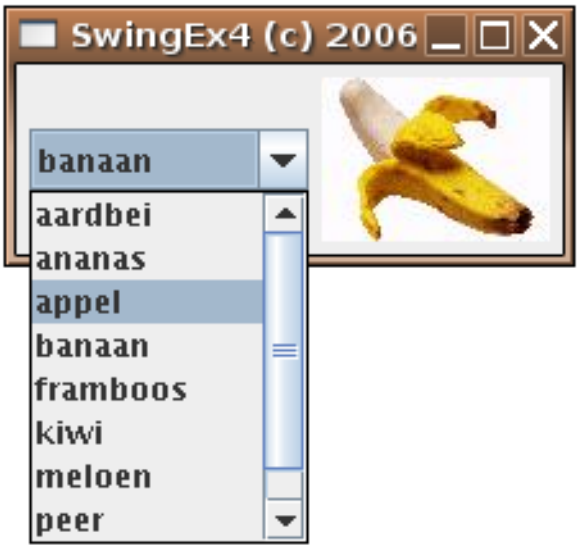
Als illustratie van enkele veelgebruikte ontwerpspatronen concentreren we ons in deze cursus voornamelijk op de Swing-bibliotheek die klassen en methoden bevat voor het programmeren van toepassingen met een grafische gebruikersinterface. Daarnaast besteden we ook aandacht aan invoer, uitvoer en databanktoegang.

Zoals bij de meeste technische vaardigheden, is ook bij het programmeren de praktijk zeer belangrijk. We raden je ten zeerste aan om vele oefeningen op deze leerstof te maken. Je vind verschillende opgaven op de website van de opleiding (*http://twizz.UGent.be/student/prog2*). De site bevat ook een aantal programmavoorbeelden, errata bij deze cursus en andere informatie over Java.

KC, mei 2006

1. Programmeren van GUIs met Java Swing

Moderne informaticatoepassingen hebben vaak een handige grafische gebruikersinterface (*graphical user interface*, *GUI*, spreek uit 'goe-ie') die fel in contrast staat met de toetsenbordgeoriënteerde programma's van weleer.



Het schrijven van een dergelijke toepassing vraagt niet alleen een aantal moderne hulpmiddelen — een objectgerichte programmeertaal zoals Java met een ondersteunende klassenbibliotheek zoals Swing — maar ook een moderne programmeermethodiek.

Dit en volgende hoofdstukken introduceert je tot deze nieuwe manier van programmeren. Het eenvoudige voorbeeldprogramma dat we in dit inleidend hoofdstuk zullen bespreken en dat hierboven is afgebeeld, opent een venster waarin de gebruiker met een aantal eenvoudige muisbewerkingen verschillende afbeeldingen kan te voorschijn brengen.

1.1. Basisconcepten

GUI-toepassingen beelden altijd één of meer *vensters* op het scherm af. Vensters bezitten een *titelbalk*, een *kader* (een getekende omtrek die je kan 'vastpakken' om het venster te verplaatsen of van grootte te veranderen) en een *binnengebied*. Java zal met elk venster een afzonderlijke object associëren.

Java bevat twee verschillende klassenbibliotheken waarmee je een grafische gebruikersinterface kan opbouwen. Enerzijds is er *Swing* en anderzijds bestaat ook *AWT* (*Abstract Windowing Toolkit*). Deze laatste bibliotheek is ondertussen behoorlijk verouderd en komt in deze tekst niet aan bod. We moeten echter vermelden dat Swing voor een stuk steunt op AWT. Vandaar dat we toch hier en daar nog een aantal AWT-klassen zullen nodig hebben.

Het binnengebied van een venster bevat een aantal grafische *componenten*. In ons voorbeeld gebruiken we een zogenaamde *combobox* (een lijst van namen waaruit een keuze kan gemaakt worden) en een *label* (die de afbeelding bevat). Er is nog een derde (verborgen) component: een grijs achtergrond*paneel* dat het volledige binnengebied bestrijkt en waarop de twee andere componenten zich bevinden. Ook componenten stel je in een Java-programma voor als afzonderlijke objecten.

Wanneer je in ons voorbeeld met de muis de naam van een afbeelding selecteert, wordt dit op één of andere manier binnen de toepassing geregistreerd. Men zegt dat er zich een *gebeurtenis* (*event*) voordoet. Andere gebeurtenissen zijn bijvoorbeeld het indrukken van een knop, het aanvinken van een checkbox, het groter maken van een venster, enz.

Gebeurtenissen worden *opgevangen* door bijzondere objecten die men in Java *luisteraars* (*listeners*) noemt. Een luisteraar bepaalt het effect van een gebeurtenis. In ons voorbeeld zorgt de luisteraar ervoor dat er een nieuwe prent wordt afgebeeld wanneer de gebruiker een andere naam selecteert.

Een groot gedeelte van de toepassing hoef je niet echt zelf te programmeren — dit is de kracht van een bibliotheek zoals Swing en één van de belangrijke verschillen tussen het programmeren van een GUI-toepassing en de ontwikkeling van een traditioneel programma. Zo kan je in je programma een kant-en-klare comboboxcomponent uit de bibliotheek gebruiken waarin je enkel nog de lijst van mogelijke keuzes moet 'invullen'. Je hoeft je dus gelukkig niet druk te maken over het achterliggende programmeerwerk dat de combobox op de juiste manier 'naar beneden laat vallen', of dat de juiste tekstletters op het scherm tekent, op de juiste plaats en in de juiste kleuren.

Een eenvoudige GUI-toepassing begint meestal met de definitie van een aantal componenten en luisteraarobjecten die deze componenten met elkaar verbinden. Deze componenten worden in vensters geplaatst en het is uiteindelijk de gebruiker (en niet de programmeur) die door zijn interactie met de toepassing bepaalt wanneer welke luisteraars worden geactiveerd en in welke volgorde.

1.2. Vensters

Het volgende programmavoorbeeld is wellicht één van de kleinste GUI-toepassingen die je ooit zult ontmoeten. Het toont enkel een leeg venster op het scherm.

```
import javax.swing.JFrame;

public class SwingEx1 {

    public static void main (String[] args) {
        JFrame window = new JFrame ("SwingEx1 (c) 2006 KC");
        window.setSize (300, 50);
        window.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
        window.setVisible (true);
    }
}
```

De **import**-opdracht bovenaan geeft aan dat we hier een klass *JFrame* uit de Swing-bibliotheek gebruiken. Er zijn vier pakketten (*packages*) die we regelmatig zullen ontmoeten, namelijk *java.awt*, *java.awt.event*, *javax.swing* en *javax.swing.event*. Voor het gemak kan je dus bij elke GUI-klasse die je schrijft bovenaan de volgende vier **import**-opdrachten plaatsen:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
```

Strikt gezien heb je ze niet steeds alle vier nodig en in de elektronische documentatie kan je terugvinden welke klassen ze vertegenwoordigen. In de praktijk raadt men tegenwoordig aan om klassen zoveel mogelijk met hun volledige naam te importeren (en niet met een *-wildcard). Professionele programmeeromgevingen (zoals bijvoorbeeld NetBeans) kunnen je hierbij van dienst zijn.

Een venster wordt in een programma meestal voorgesteld als een object van de klasse *JFrame* (er zijn een aantal alternatieven, zie hoofdstuk 7). Je maakt een nieuw venster aan door een constructor op te roepen van deze klasse. De meestgebruikte constructor van *JFrame* neemt als argument een string die in de titelbalk van dat venster moet worden weergegeven.

Vele Swing-klassen (zoals *JFrame*) hebben een naam die begint met de hoofdletter J. Dezelfde naam, zonder de eerste J, verwijst dan vaak naar de AWT-klasse met dezelfde functionaliteit (bijvoorbeeld *Frame*). Zorg er dus voor dat je je niet per ongeluk vergist.

De methode *setSize* van *JFrame* legt de afmetingen van (het binnengebied van) het venster vast, gerekend in schermpixels. De eerste parameter is de breedte, de tweede de hoogte. Doorgaans bepaalt een venster zelf hoe groot het is door naar de componenten te kijken die het bevat. De methode *setSize* wordt daarom in de praktijk weinig gebruikt. Aangezien we nog geen componenten gebruiken, zou het venster bij dit eenvoudige voorbeeld zonder de oproep van *setSize* echter een grootte krijgen van 0 bij 0 pixels.

De methode *setDefaultCloseOperation* van *JFrame* vertelt wat er moet gebeuren wanneer de gebruiker het venster sluit door bijvoorbeeld met de muisknop op het 'sluit'knopje in het kader te klikken. De parameter *JFrame.EXIT_ON_CLOSE* is een constante die aangeeft dat het programma moet worden beëindigd wanneer het venster wordt gesloten. Zonder deze opdracht blijft het programma lopen wanneer het venster sluit, zelfs wanneer er geen enkel ander venster in de toepassing nog open staat. We komen hier later nog op terug (zie §7.2).

De methode *setVisible* van *JFrame* wordt gebruikt om een venster al dan niet zichtbaar te maken, klaar voor interactie met de gebruiker. Merk op dat deze methode *niet* wacht om verder te gaan totdat de gebruiker het venster heeft gesloten.

1.3. Componenten

We breiden de toepassing uit door twee componenten aan het venster toe te voegen: een *combobox* en een afbeelding (van een banaan). Een combobox is een object van de klasse *JComboBox*. De constructor die we gebruiken om dit object aan te maken, heeft één enkele parameter: een tabel met strings die moeten worden opgesomd in de *drop-down-list* van de combobox en waaruit de gebruiker een keuze kan maken.

```
String [] choices
    = { "aardbei", "ananas", "appel", "banaan", "framboos",
        "kiwi", "meloen", "peer", "watermeloen" };

JComboBox comboBox = new JComboBox (choices);
```

De afbeelding van een banaan wordt voorgesteld door een component van het type *JLabel*. Dergelijke *labels* kan je gebruiken om een afbeelding te tonen of om een vaste tekst af te beelden (of beide).

De afbeelding zelf is een object van het type *Icon* (een interface). *JLabel* heeft een constructor die een dergelijk object als parameter neemt. Swing voorziet verschillende implementatieklassen voor de interface *Icon*, waarvan voor ons *ImageIcon* wellicht de meest belangrijke is. Hiermee kan je een afbeeldingbestand (in GIF-, JPEG- of PNG-formaat) als afbeelding gebruiken. In het voorbeeldje gebruiken we voorlopig een zelfgeschreven methode *getIcon*. Deze methode bespreken we in meer detail in paragraaf §1.5.

```
Icon icon = getIcon ("banaan.jpg");
JLabel label = new JLabel (icon);
```

(Deze twee lijnen kan je gerust combineren tot één enkele opdracht.)

Eenmaal de componenten zijn aangemaakt, moet je ze op het venster plaatsen. In Swing gebeurt dit op een indirecte manier: je plaatst de componenten niet op het venster, maar op het zogenaamde inhoudspaneel (*content pane*) van het venster. Het inhoudspaneel is een component die overeenkomt met het binnengebied van het venster. Je bekomt het inhoudspaneel van een venster als waarde van de methode *getContentPane* (zie hieronder).

Het inhoudspaneel is een *container*: een component die ook andere componenten kan bevatten. Vandaar het type *Container* dat we gebruiken voor de variabele *cp* in onderstaand fragment.

```
JFrame window = new JFrame ("SwingEx2 (c) 2006 KC");
Container cp = window.getContentPane ();
cp.setLayout (new FlowLayout ());
cp.add (comboBox);
cp.add (label);
```

Om een component toe te voegen aan een container gebruik je de methode *add*. De uiteindelijke positie van de component binnen de container wordt bepaald door de zogenaamde *layout-manager* van deze container. Er bestaan verschillende soorten layout-managers (voor meer informatie, zie §6). In dit voorbeeld gebruiken we een zogenaamde *flow-layout*. Dit type layout-manager vult de container gewoon met componenten op van links naar rechts, in dezelfde volgorde als waarmee de componenten aan de container worden toegevoegd.

Merk op hoe we de layout-manager instellen met behulp van de methode *setLayout*. De layout-manager zelf is op zijn beurt een object dat wordt aangemaakt met de gepaste constructor.

Vooraleer we tenslotte het venster op het scherm afbeelden, laten we het venster zelf eerst zijn afmetingen bepalen. Dit gebeurt met de methode *pack*.

```
window.pack ();
window.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
window.setVisible (true);
```

De grootte van het venster — of liever, de grootte van het inhoudspaneel — wordt door de layout-manager van het inhoudspaneel bepaald aan de hand van de afmetingen en de schikking van de verschillende componenten die aan het paneel werden toegevoegd.

1.4. Luisteraars

De componenten die we zojuist op het scherm hebben afgebeeld, reageren op de verwachte manier op muishandelingen en toetsaanslagen van de gebruiker: je kan de combobox bijvoorbeeld 'open' en 'dicht' klikken met de muis en je kan er een bepaalde keuze in selecteren. Klikken op het label heeft geen enkel effect, maar ook dit is het gedrag dat je van een label verwacht.

In een volgende stap willen we het programma laten reageren wanneer de gebruiker een bepaalde keuze in de combobox maakt. Zoals we in de inleiding reeds kort hebben aangehaald, moeten we hiervoor een luisteraar creëren die op de gepaste gebeurtenis reageert.

In dit voorbeeld is de gebeurtenis een zogenaamde *actiegebeurtenis* (een object van de klasse *ActionEvent*) en moeten we een luisteraar aanmaken die de interface *ActionListener* implementeert. Anders gezegd, we moeten een nieuwe klasse schrijven (bijvoorbeeld *MyListener3*) die *ActionListener* implementeert en we maken een object van die klasse aan dat we dan als luisteraar gebruiken.

```
ActionListener listener = new MyListener3 (...);
```

Opdat Swing zou weten welke luisteraar er bij welke gebeurtenis hoort, moet je de luisteraar bovendien *registreren* bij de component die de gebeurtenis genereert (in dit geval dus bij de combobox).

comboBox.addActionListener (*listener*);

De interface *ActionListener* bezit slechts één methode:

public void *actionPerformed* (*ActionEvent e*);

De luisteraarklasse *MyListener3* moet deze interface implementeren en moet met andere woorden een definitie geven voor de methode *actionPerformed*. Swing zorgt ervoor dat deze methode automatisch wordt opgeroepen wanneer de gebeurtenis zich voordoet waarvoor deze luisteraar is geregistreerd. De parameter *e* kunnen we gebruiken om meer te weten te komen over de context waarin de gebeurtenis plaats vond.

Voor de eenvoud laten we de luisteraar in eerste instantie slechts de tekst die in de combobox werd geselecteerd op de console afdrukken. Dit is een mogelijke implementatie van *MyListener3*:

```
public class MyListener3 implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        JComboBox comboBox = (JComboBox)e.getSource();
        String text  = (String)comboBox.getSelectedItem();
        System.out.println (text );
    }
}
```

(Let op de verschillende casts die hier nodig zijn. De laatste twee opdrachten kan je combineren tot één en dan valt de cast naar *String* weg.)

De enige opdracht die nog aan het hoofdprogramma moet worden toegevoegd, is de volgende:

comboBox.addActionListener (**new** *MyListener3* ());

De uitdrukking *e.getSource()* geeft het object terug dat de gebeurtenis *e* heeft veroorzaakt. Het waardetype van deze methode is *Object*, maar omdat we weten dat het hier om een combobox gaat, kunnen we een *down cast* toepassen naar het type *JComboBox*.

De methode *getSelectedItem()* van *JComboBox* geeft het object terug dat op dat moment is geselecteerd. Omdat comboboxen ook andere elementen dan tekstelementen bevatten (afbeeldingen bijvoorbeeld), is het waardetype van deze methode *Object* en niet *String*.

We moeten de structuur van het programma enigszins aanpassen wanneer we ook wensen dat de luisteraar de afbeelding op het label verandert. Het luisteraarobject kan niets aan het label veranderen als het niet beschikt over een referentie naar dit label.

Daarom geven we een referentie mee op het moment dat de luisteraar wordt aangemaakt — en in één moeite meteen ook een aantal referenties naar andere objecten die voor de luisteraar van nut zijn. We hebben dus een aantal nieuwe objectattributen nodig en een constructor:

```
public class MyListener4 implements ActionListener {
    private JLabel label;
    private String[] choices;
    private JComboBox comboBox;

    ...
    public MyListener4
            (JLabel label, String[] choices, JComboBox comboBox) {
        this.label = label;
        this.choices = choices;
        this.comboBox = comboBox;
    }
}
```

Bij het opvangen van de gebeurtenis, gebruiken we de drie referenties die door de constructor werden opgeslagen:

```
    public void actionPerformed (ActionEvent e) {
        int index = comboBox.getSelectedIndex ();
        Icon icon = SwingEx4.getIcon (choices[index]+".jpg");
        label.setIcon (icon);
    }
```

De methode *getSelectedIndex* geeft het volgnummer aan van de tekst die in de combobox werd geselecteerd. Met *setIcon* kan je de afbeelding op een label veranderen.

1.5. Bronnen in het class path

Vooraleer we verder gaan, is het nuttig om ook eens de implementatie van de methode *getIcon* van dichterbij te bekijken. Deze methode neemt de naam van een afbeeldingsbestand als argument en geeft het corresponderende *ImageIcon* terug. De meest eenvoudige implementatie hiervan is de volgende:

```
public static Icon getIcon (String naam) {
    return new ImageIcon (naam); // niet zo geschikt
}
```

Volgens de elektronische documentatie neemt de constructor van *ImageIcon* die we hier gebruikt hebben de naam van het overeenkomstige bestand als parameter. Er wordt echter niet verteld waar ergens op de computer naar dit bestand zal worden gezocht. In de praktijk blijkt dit in de huidige directory te zijn, tenzij de naam met een schuine streep begint, in welk geval de string dan als absolute padnaam wordt geïnterpreteerd.

Beide oplossingen zijn misschien voldoende goed voor een klein testprogramma, maar in de praktijk zijn ze niet zo handig: je kan er niet zeker van zijn dat de eindgebruiker de huidige directory wel correct zal instellen, en een absolute padnaam is niet de meest geschikte oplossing wanneer je de toepassing gemakkelijk naar andere toestellen of besturingssystemen wil kunnen overdragen.

Dergelijke externe bestanden (afbeeldingen, hulpteksten, configuratie-informatie, enz.) die je in je programma nodig hebt, noemt men *bronnen* (Engels: *resources*). Java biedt een aantal handige manieren om met dergelijke bronnen om te gaan. Hierbij maken we gebruik van het zogenaamde class path.

Het *class path* is het geheel van 'locaties' waar Java op zoek gaat naar klassen. Deze locaties kunnen directories zijn, maar ook zogenaamde *JAR archieven* (zie appendix A), of locaties op het Internet. Het class path kan je opgeven in de omgevingsvariabele `CLASSPATH` of als `-cp`-argument bij het opstarten van `java`.

Je kan elke klasse (of eigenlijk aan zijn *class loader*) vragen om voor jou een bepaalde bron op te sporen. In de praktijk betekent dit meestal dat de bron wordt gezocht op dezelfde plaats waar ook het *.class*-bestand van die klasse is gevonden (dus ergens in het class path).

In ons voorbeeld doen we dit op de volgende manier:

```
public static Icon getIcon (String naam) {
    return new ImageIcon (SwingEx2.class.getResource (naam));
}
```

Dit vraagt wellicht een woordje uitleg.

De uitdrukking *MijnKlasse*.**class** geeft een object terug van de klasse *Class* dat alle informatie over de klasse *MijnKlasse* bevat, dus onder andere ook informatie over waar het *.class*-bestand van die klasse zich ergens in het class path bevindt.

De methode

```
public URL getResource (String naam);
```

uit de klasse *Class* laat het klassenobject de bron opzoeken met de gegeven naam en geeft er een zogenaamde *URL* voor terug (een *Uniform Resource Locator*, dezelfde URL die je ook van het Internet kent).

Tot slot gebruiken we dan nog de constructor

```
public ImageIcon (URL url);
```

waarmee *ImageIcon* het afbeeldingsbestand ophaalt.

In de praktijk betekent dit dus dat de opdracht *getIcon*(`"banaan.jpg"`) het bestand *banaan.jpg* zal opzoeken op dezelfde plaats (in dezelfde directory) waar ook het bestand *SwingEx2.class* terug te vinden is, zonder zich iets te hoeven aantrekken van wat de huidige directory is, of onder welk besturingssysteem het programma wordt opgestart. Dit is bijzonder handig wanneer je toepassing uit verschillende pakketten (*packages*) bestaat, omdat je dan de afbeeldingen kan opslaan in dezelfde pakketdirectory als de klasse die ze nodig heeft.

Als alternatief kan je er ook voor kiezen om al je afbeeldingen in dezelfde map te bewaren. In dat geval geef je voor de bron een naam op die met een schuine streep begint. Dergelijke absolute namen worden gerekend vanaf de basis van het class path. Hebben we een pakket *prog2.images* waarin we alle afbeeldingen bewaren, dan gebruiken we de volgende definitie voor *getIcon*:

```
public static Icon getIcon (String naam) {
    return new ImageIcon
        (SwingEx4.class.getResource ("/prog2/images/" + naam));
}
```

Merk op dat hier, zoals gebruikelijk bij pakketten, de puntjes in een pakketnaam zijn vervangen door schuine strepen. (En hoewel het besturingssysteem Windows traditioneel omgekeerde schuine strepen gebruikt als scheidingsteken, moet je ook in dat geval de UNIX-conventies blijven toepassen. Dit maakt je code gemakkelijker overdraagbaar.)

1.6. Zelfgemaakte componenten

Het is in Swing zeer gebruikelijk om zelf nieuwe componentenklassen te ontwerpen. Je doet dit door een gekende componentenklasse uit te breiden en er je eigen methoden aan toe te voegen. Als voorbeeld definiëren we een nieuw soort paneel met daarin een combobox en een afbeelding met dezelfde functionaliteit als voorheen.

Het paneel is een extensie van de klasse *JPanel*, het standaardpaneel uit de Swing-bibliotheek.

```
public class MyPanel5 extends JPanel {

    private static final String [] choices
        = { "aardbei", "ananas", "appel",
            "banaan", "framboos",
            "kiwi", "meloen",
            "peer", "watermeloen"
        };

    private static final Icon [] icons;

    static {      // initializeer tabel icons
        int nr = choices.length;
        icons = new Icon [nr];
        for (int i=0; i < nr; i++)
            icons[i] = new ImageIcon
                        (MyPanel5.class.getResource
                            ("/prog2/images/" + choices[i] + ".jpg"));
    }
    ...
}
```

Om redenen van efficiëntie creëren we op voorhand reeds een tabel *icons* met alle mogelijke afbeeldingen.

Bij constructie van een nieuw paneel maken we de twee componenten aan en plaatsen die op het paneel.

```
    private final JComboBox comboBox;

    private final JLabel label;

    public MyPanel5 () {
        comboBox = new JComboBox (choices);
        comboBox.addActionListener (new MyListener ());
        label  = new JLabel (icons[0]);
        setLayout (new FlowLayout ());
        add (comboBox);
        add (label);
    }
```

Eigenlijk is de oproep van *setLayout* niet nodig, omdat een *JPanel* standaard een flow-layout als layout-manager gebruikt. Merk ook op dat een combobox per default de eerste keuze op het scherm toont. Je kan een ander element selecteren met de methode *setSelectedIndex*.

Merk op dat we met het paneel een luisteraar hebben geassocieerd van het type *MyListener*. Dit keer hebben we de klasse *MyListener* als *binnenklasse* gedefinieerd van *MyPanel5*. Dit biedt het bijkomend voordeel dat elk object van die klasse rechtstreeks toegang heeft tot de attributen van het paneel.

```
public class MyPanel5 extends JPanel {
    private class MyListener implements ActionListener {
        public void actionPerformed (ActionEvent e) {
            label.setIcon  (icons[comboBox.getSelectedIndex ()]);
        }
    }  ...
}
```

Met andere woorden, de identifier *comboBox* in de methode *actionPerformed* van de klasse *MyListener* verwijst naar het veld *comboBox* van het object van de klasse *myPanel5* dat de luisteraar (van het type *MyListener*) heeft gecreëerd (en analoog voor de variabelen *label* en *icons*).

Tot slot drukken we nog een programmafragment af waarin we een object van het type *MyPanel5* gebruiken.

```
JFrame window = new JFrame ("Titel");
window.setContentPane (new MyPanel5());
window.pack ();
window.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
window.setVisible (true);
```

Let hier vooral op de tweede opdracht, die een paneel van het gevraagde type als inhoudspaneel van het venster installeert. Dit is iets eleganter dan het nieuwe paneel bovenop het oorspronkelijke inhoudspaneel te plaatsen:

```
window.getContentPane ().add (new MyPanel5()); // minder goed
```

1.7. Alternatieve klassenstructuur

Wanneer, zoals in bovenstaand voorbeeld, de luisteraarklasse slechts een eenvoudige rol speelt, zal men vaak een *anonieme* klasse gebruiken in plaats van een benoemde binnenklasse zoals *MyListener*. Bijvoorbeeld:

```
public MyPanel6 () {
    comboBox = new JComboBox (choices);
    comboBox.addActionListener
        (new ActionListener () {
                public void actionPerformed (ActionEvent e) {
                    label.setIcon  (icons[comboBox.getSelectedIndex ()]);
                } });

    label  = new JLabel (icons[0]);
    setLayout (new FlowLayout ());
    add (comboBox);
    add (label);
}
```

Een ander alternatief bestaat erin om het nieuwe paneel zelf naar de gebeurtenis te laten luisteren. We zorgen ervoor dat de paneelklasse de interface *ActionListener* implementeert en we voorzien een methode *actionPerformed* voor deze klasse:

```
public class MyPanel7 extends JPanel implements ActionListener {
    ...
    public void actionPerformed (ActionEvent e) {
        label.setIcon  (icons[comboBox.getSelectedIndex ()]);
    }
    ...
}
```

Wanneer we een nieuw paneel van dit type aanmaken, registreren we dit nieuwe paneel dan zelf als luisteraar van zijn eigen combobox.

```
public MyPanel7 () {
    comboBox = new JComboBox (choices);
    comboBox.addActionListener (this);
    label  = new JLabel (icons[0]);
    setLayout (new FlowLayout ());
    add (comboBox);
    add (label);
}
```

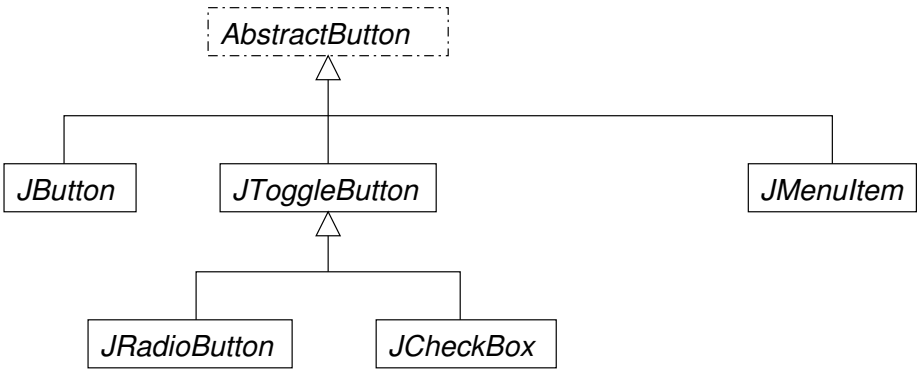
Zoals je ziet, bestaan er vaak meerdere manieren om hetzelfde resultaat te bekomen en het is ook niet altijd meteen duidelijk wat nu de 'beste' manier is. Ook onder professionelen zijn de meningen hierover vaak verdeeld. We willen echter wel meteen kwijt dat wanneer de toepassingen iets moeilijker worden dan dit inleidend voorbeeld, het zogenaamde *MVC patroon* veelal de voorkeur geniet. We komen hierop terug in hoofdstuk 4.

2. Eenvoudige componenten

In dit hoofdstuk bespreken we kort een aantal *atomaire* componenten van Swing — dit wil zeggen, componenten die geen andere componenten als onderdeel hebben. Het is zeker niet de bedoeling om volledig te zijn en voor meer details verwijzen we naar de (elektronische) documentatie. Een aantal meer complexe componenten komt later nog aan bod in hoofdstuk 4.

2.1. Knoppen

Swing ondersteunt verschillende soorten 'knoppen', zoals aangegeven in het onderstaande klassendiagram.



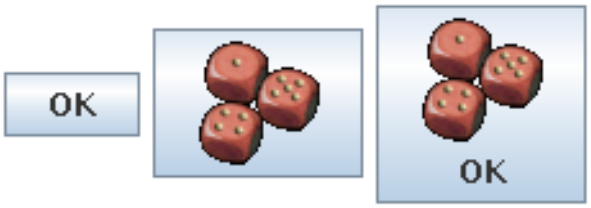
De klasse *AbstractButton* is een abstracte klasse die de gemeenschappelijke functionaliteit van de andere knoppenklassen groepeert. Je hebt deze klasse in principe nooit rechtstreeks nodig, maar het is wel soms nuttig om haar documentatie te raadplegen.

De klasse *JMenuItem* (en haar afgeleide klassen) dient voor het opbouwen van *menus*.

2.1.1. De klasse *JButton*

Een object van de klasse *JButton* stelt een *drukknop* voor: een knop die je met de muis kan aanklikken en die slechts ingedrukt blijft zolang ook de muisknop is ingedrukt. Dit soort knop wordt in de GUI-wereld heel vaak gebruikt.

Knoppen kunnen tekst als opschrift hebben, een afbeelding, of allebei.



Bovenstaande knoppen maak je aan met de volgende opdrachten:

JButton button1 = **new** *JButton* (`"OK"`);

JButton button2 = **new** *JButton* (*stenenIcon*);

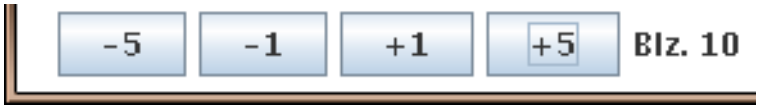
JButton button3 = **new** *JButton* (`"OK"`, *stenenIcon*);
button3.setHorizontalTextPosition (*SwingConstants.CENTER*);
button3.setVerticalTextPosition (*SwingConstants.BOTTOM*);

De laatste twee opdrachten bepalen de onderlinge positie van de tekst ten opzichte van de afbeelding op de knop. (Anders komt de tekst rechts van de afbeelding terecht in plaats van eronder.)

Wanneer je op een drukknop klikt, veroorzaakt dit een gebeurtenis van het type *ActionEvent*. Je vangt een dergelijke *actiegebeurtenis* op door met de methode *addActionListener* een luisteraar bij de knop te registreren van het type *ActionListener*. Swing roept automatisch de methode *actionPerformed* op van elke luisteraar die met de knop is verbonden. Dit is precies dezelfde techniek als bij de combobox in het voorbeeld van hoofdstuk 1.

2.1.2. Meerdere gelijkaardige knoppen

Vaak bevat een paneel meerdere knoppen met een gelijkaardige functie. Als voorbeeld bekijken we het onderstaande stel knoppen dat een onderdeel zou kunnen vormen van een documentenbrowser. De knoppen dienen om 1 of 5 pagina's vooruit of achteruit te springen in een document.



Er zijn verschillende manieren waarop je een dergelijke functionaliteit kan implementeren. Het ligt wellicht het meest voor de hand om voor elke knop een afzonderlijke klasse te definiëren waarvan telkens een object als luisteraar kan dienen. Dit lukt wellicht het best met behulp van vier anonieme binnenklassen:

```
public ManyButtons1 () {
    pageNr = 0;
    label  = new JLabel ("Blz.  " + pageNr);

    JButton buttonMinus5 = new JButton ("-5");
    add(buttonMinus5);
    buttonMinus5.addActionListener
        (new ActionListener () {
                public void actionPerformed (ActionEvent e) {
                    incPageNr (-5);
                }
            });
     ...

    JButton buttonPlus5 = new JButton ("+5");
    add(buttonPlus5);
    buttonPlus5.addActionListener
        (new ActionListener () {
                public void actionPerformed (ActionEvent e) {
                    incPageNr (5);
                }
            });

    add (label);
}
```

De klasse *ManyButtons1* (waarvan je hierboven slechts de constructor ziet) is een extensie van *JPanel*. Zoals de naam aangeeft wordt de methode *incPageNr* verondersteld het paginanummer te verhogen met het opgegeven aantal.

Deze code is niet echt een voorbeeld van een goede programmeerstijl. Als alternatief gebruiken we misschien beter telkens hetzelfde object (het paneel zelf bijvoorbeeld) als luisteraar voor de vier verschillende knoppen. We moeten dan wel een manier vinden om uit te vissen welke van de vier knoppen op een gegeven moment de luisteraar heeft geactiveerd. Dit kan bijvoorbeeld door naar de bron van de gebeurtenis te vragen met behulp van *getSource*.

```
public class ManyButtons2 extends JPanel implements ActionListener {
    // Plaatst knoppen op het paneel en registreert ze
    public ManyButtons2 () {
        ...
        for (int i=0; i < buttons.length; i++) {
            buttons[i] = new JButton (captions[i]);
            buttons[i].addActionListener (this);
            add(buttons[i]);
        }
        ...
    }

    // Implementatie van ActionListener
    public void actionPerformed (ActionEvent e) {
        int nr = 0;
        while (nr < 4 && buttons[nr] != e.getSource())
            nr ++;
        if (nr < 4)
            incPageNr (increments[nr]);
        else
            throw new IllegalStateException
                ("Action event from unexpected source");
    }

    ...

}
```

Met elke gebeurtenis van het type *ActionEvent* komt er een zogenaamde *actieopdracht* (*action command*) overeen. Dit is een string die de gebeurtenis be-

schrijft. Bij knoppen neemt Swing per default het opschrift van de knop als actie-opdracht.

Je kan de actie-opdracht aan de gebeurtenis opvragen met behulp van de methode *getActionCommand*. Dit biedt je een alternatieve manier om te bepalen welke knop de luisteraar heeft geactiveerd:

```
public void actionPerformed (ActionEvent e) {
    String command = e.getActionCommand ();
    if (command.equals ("-5"))
        incPageNr (-5);
    else if (command.equals ("-1"))
        incPageNr (-1);
    else if (command.equals ("+1"))
        incPageNr (1);
    else if (command.equals ("+5"))
        incPageNr (5);
    else
        throw new IllegalStateException
            ("Action event with unexpected action command");
}
```

Met *setActionCommand* kan je aan een knop een andere actie-opdracht toekennen dan zijn opschrift. Dit is vooral handig bij knoppen die een afbeelding als opdruk gebruiken in plaats van tekst, maar kan ook nuttig zijn in andere gevallen.

Wanneer we in ons voorbeeld de knoppen met opschrift '+1' en '+5' als actie-opdracht de strings '1' en '5' meegeven, kan de incrementwaarde rechtstreeks vanuit de actie-opdracht worden afgeleid.

```
public ManyButtons4 () {
    ...
    buttons[2].setActionCommand ("1");
    buttons[3].setActionCommand ("5");
}

public void actionPerformed (ActionEvent e) {
    incPageNr (Integer.parseInt (e.getActionCommand ()));
}
```

(Merk op dat *Integer.parseInt* geen strings aanvaardt die met een plusteken beginnen.)

Wanneer de string die je als opschrift van een label, knop of andere component gebruikt, begint met de zes tekens '`<html>`' dan wordt deze string geïnterpreteerd als HTML-code. Op die manier kan je in je opschriften cursieve of vette tekst gebruiken, of andere kleuren en lettertypes. Bijvoorbeeld:

```
JButton boldButton = new JButton ("<html><b>Vet</b></html>");
```

Tot slot vermelden we nog een 'gulden middenweg' die in vele situaties kan worden toegepast: als luisteraars gebruik je verschillende objecten die tot dezelfde klasse behoren.

In ons voorbeeld gebruiken we hiervoor de volgende binnenklasse:

```
private class ButtonListener implements ActionListener {
    private final int increment;

    public ButtonListener (int increment) {
        this.increment = increment;
    }

    public void actionPerformed (ActionEvent e) {
        incPageNr (increment);
    }
}
```

De vier luisteraars worden in de constructor van het paneel aangemaakt en geregistreerd.

```
for (int i=0; i < buttons.length; i++) {
    buttons[i] = new JButton (captions[i]);
    buttons[i].addActionListener (new ButtonListener (increments[i]));
    add(buttons[i]);
}
```

2.1.3. De methode *setEnabled*

In sommige situaties heeft het indrukken van een bepaalde knop geen zin — een 'vorige pagina'-knop bijvoorbeeld wanneer de eerste pagina reeds is geselecteerd.

In een dergelijk geval kan je een knop *desactiveren* (Engels: *disable*) om hem dan later terug te *activeren* (*enable*). Een dergelijke knop zal op het scherm *gedimd* worden weergegeven en kan niet worden ingedrukt.

Je doet dit met de methode *setEnabled* die als argument **true** of **false** neemt. De vier knoppen uit het voorbeeld van paragraaf §2.1.2 kan je als volgt activeren of desactiveren bij de start van het programma en telkens wanneer het paginanummer verandert:

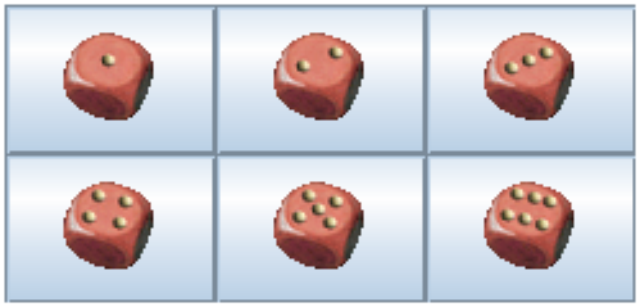
buttons[0].*setEnabled* (*pageNr* >= 5);
buttons[1].*setEnabled* (*pageNr* >= 1);
buttons[2].*setEnabled* (*pageNr* <= 99);
buttons[3].*setEnabled* (*pageNr* <= 95);

2.1.4. De klassen *JToggleButton*, *JRadioButton* en *JCheckBox*

Schakelknoppen (*toggle buttons*) van de klasse *JToggleButton* hebben hetzelfde uitzicht als gewone drukknoppen, maar gedragen zich anders. Een schakelknop kan zich in twee verschillende *toestanden* bevinden — 'aan' en 'uit', 'ingedrukt' en 'niet ingedrukt', 'geselecteerd' en 'niet geselecteerd'. Je verandert de toestand van een schakelknop door er met de muis op te klikken.

Schakelknoppen worden vaak gegroepeerd in zogenaamde *knoppengroepen*. Wanneer je de ene knop in een dergelijke groep selecteert, worden de andere knoppen in dezelfde groep automatisch uitgeschakeld. Hiermee kan je de gebruiker één keuze laten aanduiden uit meerdere mogelijkheden.

Een knoppengroep is een object van de klasse *ButtonGroup*. Je gebruikt de methode *add* van die klasse om een schakelknop aan die groep toe te voegen. Verwar dit niet met de *add* die nodig is om de knop op een paneel te plaatsen.



Het paneel met dobbelstenen uit de bovenstaande figuur kan je bijvoorbeeld op de volgende manier aanmaken.

```
JPanel panel = new JPanel (new GridLayout (2,0)); // zie §6.4
ButtonGroup group = new ButtonGroup ();

for (int i=1; i <= 6; i++) {
    JToggleButton button
        = new JToggleButton (new ImageIcon
            (ButtonDemo.class.getResource
                ("/prog2/images/steen" + i + ".gif")));
    group.add (button);
    panel.add (button);
}
```

In een knoppengroep kan er tegelijkertijd ten hoogste één knop zijn ingedrukt, maar het is ook mogelijk dat er geen enkele knop van de groep is geselecteerd. Dit doet zich meestal enkel voor na het opstarten van de toepassing, want zodra de gebruiker één knop aanklikt, kan hij die niet meer 'uitklikken' zonder een andere knop van de groep in te drukken. Om dit te voorkomen, kan je een schakelknop ook op voorhand vanuit het programma selecteren,

```
button.setSelected (true);
```

of je kan bij de constructie van de knop aangegeven dat hij moet geselecteerd zijn

```
button = new JToggleButton (icon, true);
```

Swing voorziet in twee bijzondere soorten schakelknoppen die veel worden toegepast. Een *radioknop* (type *JRadioButton*) bestaat uit tekst die kan worden

aangestipt (links op de figuur) en een *checkbox* (type *JCheckBox*) is een tekstlijn die kan worden aangevinkt (rechts op de figuur).

Het is een traditie om radioknoppen altijd in een knoppengroep te plaatsen, terwijl checkboxen steeds op zichzelf worden gebruikt. Voor de rest gedragen ze zich zoals elke andere schakelknop.

Er zijn verschillende manieren om in je programma te reageren op de stand van de schakelknoppen. Zo kan je met behulp van de methode *isSelected* te weten komen of een schakelknop al dan niet is ingedrukt.

```
if (boldButton.isSelected()) {
    // verander het lettertype in 'vetjes'
}
if (italicButton.isSelected()) {
    // verander het lettertype in 'cursief'
}
```

Deze techniek is vooral bruikbaar wanneer de betreffende schakelknop zich niet in een knoppengroep bevindt. Bij een knoppengroep is het immers handiger om meteen te weten welke knop er is ingedrukt dan telkens de volledige knoppengroep te overlopen en van elke knop te controleren of hij al dan niet is geselecteerd.

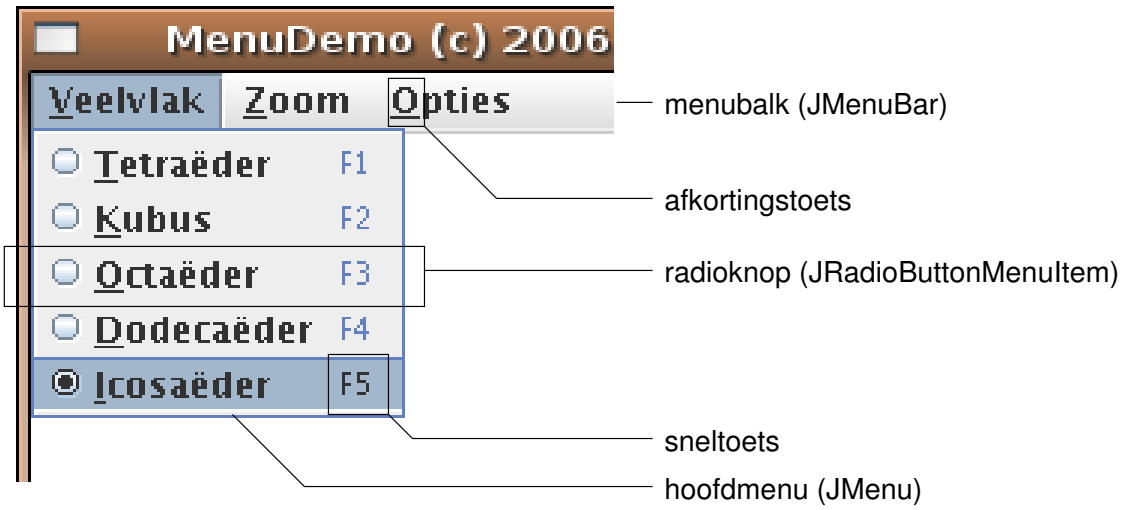
Net zoals bij drukknoppen veroorzaakt het klikken op een schakelknop ook een actiegebeurtenis — zowel bij het in- als bij het uitschakelen. Deze gebeurtenis laat je toe om gemakkelijk bij te houden welke van een groep van knoppen is ingedrukt. Daartoe laat je de luisteraar bijvoorbeeld het nummer of de naam van de knop die werd geactiveerd in een variabele opslaan, een variabele die je dan later vanuit het programma kan raadplegen.

Wens je niet enkel te reageren op de gebruiker die een knop indrukt, maar ook op toestandsveranderingen die onrechtstreeks worden veroorzaakt — zoals het uitschakelen van een knop doordat een andere knop in dezelfde knoppengroep werd ingeschakeld — dan kan je een luisteraar registreren van het type *ItemListener*.

Bij een goed programma-ontwerp blijk je deze mogelijkheid echter slechts zelden nodig te hebben.

2.2. Menu's

Aan elk venster in Swing kan je een menubalk hechten waaruit verschillende menu's kunnen worden geactiveerd. Onderstaande figuur en de figuur op de volgende bladzijde illustreren de verschillende concepten die een rol spelen bij het werken met dergelijke menu's.



2.2.1. Basisklassen

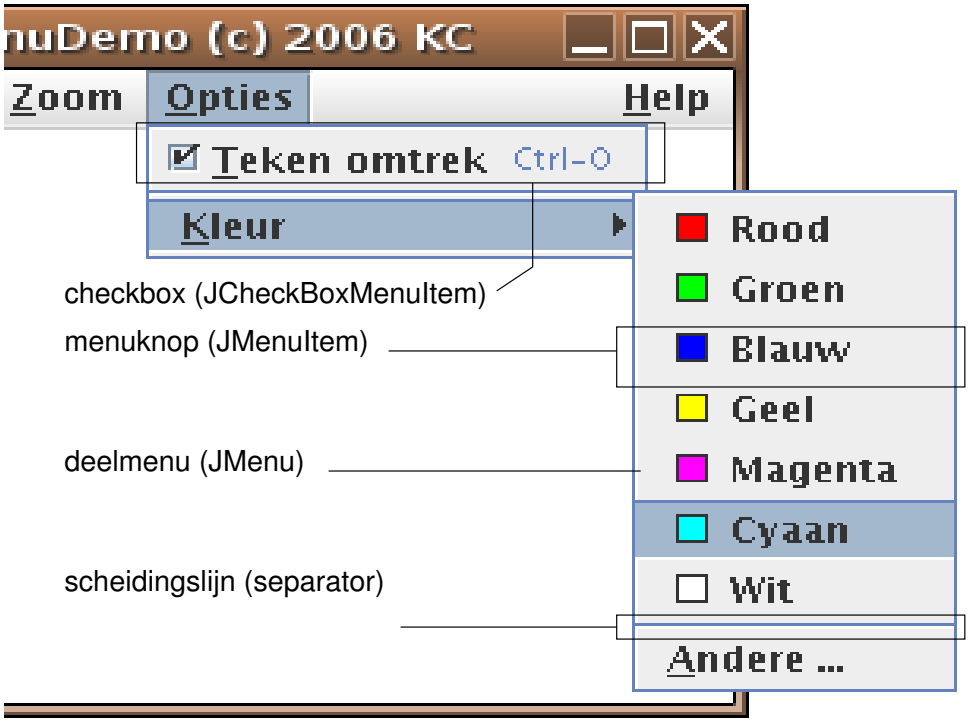
De menubalk bovenaan een venster is een component van het type *JMenuBar*. Je kan geen menu's gebruiken zonder een dergelijke menubalk. Om een menubalk te creëren en met een venster te verbinden, gebruik je de volgende opdrachten:

```
JMenuBar menuBar = new JMenuBar ();
window.setJMenuBar (menuBar);      // NIET setMenuBar!
```

Merk op dat de menubalk zich buiten het inhoudspaneel bevindt, en dus niet zoals andere componenten aan dit inhoudspaneel mag worden toegevoegd. De methode *setJMenuBar* wordt dus toegepast op het venster zelf en niet op een component.

De menu's die de gebruiker vanuit de menubalk kan oproepen, zijn objecten van het type *JMenu*. Elk menu groepeert op zijn beurt een aantal menu-elementen. We maken hierbij onderscheid tussen verschillende types.

Gewone menuknoppen van het type *JMenuItem* bestaan uit tekst en eventueel een afbeelding. Ze gedragen zich op dezelfde manier als drukknoppen — *JMenuItem* is immers een deelklasse van *AbstractButton*.



Je creëert een menuknop met één van de vele constructoren van *JMenuItem* en je voegt hem aan het menu toe zoals je een component aan een container toevoegt — met behulp van *add*.

```
JMenu mnKleur = ...;  // zie later
JMenuItem mnBlauw = new JMenuItem ("Blauw", blauwIcon);
mnKleur.add (mnBlauw);
```

Net zoals bij een gewone drukknop, registreer je een luisteraar van het type *ActionListener* bij een menuknop zodat het programma kan reageren wanneer de gebruiker het menu-element selecteert. Er bestaat geen afzonderlijke menugebeurtenis of menuluisteraarklasse. Bijvoorbeeld:

Als het selecteren van een menuknop een nieuw (dialoog)venster doet verschijnen, dan plaatst men traditioneel drie puntjes achter het opschrift van die knop.

```
mnBlauw.addActionListener (new MyActionListener ());
```

In plaats van drukknoppen kan je ook radioknoppen in een menu plaatsen, of checkboxen. De klassen die je hiervoor gebruikt, zijn *JRadioButtonMenuItem* en *JCheckBoxMenuItem*. Deze klassen gedragen zich op dezelfde manier als *JRadioButton* en *JCheckBox* — ze ondersteunen dezelfde methoden en gebeurtenissen.

```
ButtonGroup grp = new ButtonGroup ();
for (int i=0; i < names.length; i++ ) {
    JRadioButtonMenuItem item
        = new JRadioButtonMenuItem (names[i]);
    menu.add (item);
    grp.add (item);
    item.addActionListener (new MyRadioActionListener ());
    item.setSelected (true); // enkel laatste blijft geselecteerd
}
...
JCheckBoxMenuItem checkbox
    = new JCheckBoxMenuItem ("Teken omtrek");
checkbox.setSelected (true);
checkbox.addItemListener (new MyCheckBoxActionListener ());
menu.add (checkbox);
```

Menu-elementen worden van boven naar onder aan een menu toegevoegd. Je kan een scheidingslijn tussen opeenvolgende elementen plaatsen met behulp van *addSeparator*.

```
menu.add (itemRood);
...
menu.add (itemWit);
menu.addSeparator ();
menu.add (itemAndere);
```

Een deelmenu is (net zoals een hoofdmenu) een object van het type *JMenu*. Een object van dit type representeert zowel het deelmenu als de knop die dit deelmenu

activeert. De constructor van *JMenu* neemt het opschrift van de knop (en eventueel een afbeelding) als parameter. Een druk op de knop brengt het deelmenu vanzelf op het scherm, je hoeft daarvoor geen eigen luisteraar te definiëren.

Je voegt een deelmenu toe aan een menu met behulp van *add*.

```
JMenu mnOptions = new JMenu ("Opties");
menubar.add (mnOptions);
 ...
JMenu mnColor = new JMenu ("Kleur");
mnOptions.add (mnColor);
```

Zoals je uit bovenstaand fragment kan zien, voeg je op dezelfde manier een hoofdmenu toe aan de menubalk. Je kan trouwens ook gewone menuknoppen, radioknoppen en checkboxen rechtstreeks aan de menubalk toevoegen (zonder bovenliggend hoofdmenu).

De elementen in de menubalk worden automatisch van links naar rechts aan de balk toegevoegd. Het is echter een traditie om een 'Help'-menu uiterst rechts in de menubalk te plaatsen. Dit kan je doen met de volgende opdracht:

```
menuBar.add (Box.createHorizontalGlue ());
menuBar.add (helpMenu ());
```

(Deze techniek houdt verband met het feit dat een menubalk een *BoxLayout*-manager gebruikt om zijn menu-elementen te schikken.)

2.2.2. Afkortingen en sneltoetsen

Een menu kan ook geactiveerd worden met behulp van het toetsenbord in plaats van de muis. Om dit gemakkelijker te maken, associeer je met elk menu-element een (letter)toets, afkortingstoets of *mnemonic* genaamd. Als toets kiest men hiervoor meestal de eerste letter van het opschrift van het menu-element, tenzij er reeds een element is met dezelfde afkorting binnen hetzelfde (deel)menu. Deze letter wordt door Swing op het scherm automatisch onderlijnd. (Je kan dergelijke afkortingstoetsen trouwens ook gebruiken bij gewone knoppen.)

Je stelt een afkortingstoets in met behulp van de methode *setMnemonic*.

```
JMenuItem zoomIn = new JMenuItem ("Zoom in");
zoomIn.setMnemonic (KeyEvent.VK_I);
```

De constanten *KeyEvent.VK_A* tot en met *KeyEvent.VK_Z* stellen de verschillende lettertoetsen voor.

Er is ook een constructor van *JMenuItem* met de afkortingstoets als tweede parameter:

```
JMenuItem zoomIn = new JMenuItem ("Zoom in", KeyEvent.VK_I);
```

Enigzins van gelijke aard, maar vooral bedoeld voor ervaren gebruikers, zijn de zogenaamde *sneltoetsen* (*accelerators*). Dit zijn toetscombinaties die menu-elementen activeren *zonder* dat het menu-element zelf, of één van de bovenliggende menu's, op het scherm hoeft te verschijnen.

Je registreert een sneltoets bij het menu-element met de methode *setAccelerator* van het menu-element.

```
JCheckBoxMenuItem checkbox
    = new JCheckBoxMenuItem ("Teken omtrek");
checkbox.setAccelerator (KeyStroke.getKeyStroke ("ctrl O"));
```

Als argument verwacht *setAccelerator* een object van het type *KeyStroke*. Er bestaan verschillende manieren om dergelijke objecten te creëren, maar in deze context is de statische methode *getKeyStroke* met één enkel *String*-argument wellicht te verkiezen (zie fragment). Voor meer informatie verwijzen we naar de (elektronische) handleiding.

Wanneer je een sneltoets met een menu-element verbindt, dan verschijnt de benaming van die toetscombinatie automatisch in het opschrift van dit element. Merk ook op dat je geen sneltoetsen kan associëren met (deel)menu's.

2.2.3. Popup-menu's

De meeste besturingssystemen voorzien nog een andere manier om een menu te activeren.

Door het indrukken van de middelste of de rechter muistoets, kan je een zogenaamd *popup*-menu doen verschijnen naast de muiswijzer. De inhoud van dit

menu hangt soms af van de plaats waar de muiswijzer zich op dat moment bevindt.

Popup-menu's in Swing behoren tot de klasse *JPopupMenu*. Behalve dat een popup-menu geen opschrift en geassocieerde knop bezit, gebruik je deze klasse op dezelfde manier als *JMenu*.

Je voegt een popup-menu niet toe aan een menubalk of aan een deelmenu, maar aan een component. Je doet dit met behulp van *setComponentPopupMenu*:

```
JPopupMenu popup = new JPopupMenu ();
 ...
panel.setComponentPopupMenu (popup);
```

Het popup-menu wordt dan automatisch getoond wanneer de platformafhankelijke *popup trigger* wordt geactiveerd voor deze component. Bij het ene platform is dit bij het loslaten van een bepaalde muisknop, bij het andere bij het indrukken. Dit kan de rechter muisknop zijn, of de middelste of de linker in combinatie met de CTRL-toets. Soms is er ook een afzonderlijke toets op het toetsenbord die het popup-menu activeert. Het is een belangrijk voordeel van *setComponentPopupMenu* dat je je als programmeur niets van deze verschillen hoeft aan te trekken.

(In eerdere versies van Java 5.0 werkt het popup-menu enkel op componenten waar ook andere muisgebeurtenissen worden verwerkt. Dit is een bug. Je kan die bijvoorbeeld omzeilen door een tool tip aan de component te hechten.)

Indien nodig kan je popup-menu's ook zelf activeren vanuit je programma.

```
JPopupMenu popup = ...;
popup.show (comp, x, y);
```

De methode *show* van *JPopupMenu* vraagt drie argumenten: de component waarboven het popup-menu moet verschijnen en de X- en Y-coördinaten binnen die component van de positie waarop dit moet gebeuren.

2.3. De klasse *T*imer

Je gebruikt een *Timer* wanneer je een bepaalde taak op geregelde tijdstippen wil herhalen. Hoewel we een timer moeilijk een GUI-component kunnen noemen,

hebben we deze klasse toch in dit hoofdstuk opgenomen omdat ze net zoals *JButton* gebruik maakt van actieluisteraars.

Een object van de klasse *Timer* genereert op vaste tijdstippen automatisch een actiegebeurtenis. Je implementeert de taak die je wil herhalen als een luisteraar voor dit object. Je geeft de luisteraar en het tijdsinterval op wanneer je het timerobject creëert. De timer wordt opgestart met de methode *start*. (Het tijdsinterval wordt opgegeven in milliseconden.)

```
Timer timer = new Timer (1000, listener);        // 1 seconde
...
timer.start ();
```

Als voorbeeld implementeren we een klasse *CountdownLabel*, een extensie van *JLabel*. Dit label toont een getal dat elke seconde met één wordt verminderd totdat de waarde nul wordt bereikt. Je kan het aantal seconden (de startwaarde) opgeven bij het aanmaken van het label of met behulp van de methode *setCount*. Het aftellen begint wanneer je de methode *start* oproept.

De klasse *CountdownLabel* heeft een veld *value* die de huidige stand van de teller bijhoudt. De klasse gebruikt de volgende methode om deze waarde te veranderen:

```
private void setValue (int value) {
    this.value = value;
    setText (value + " sec");
}
```

Merk op dat ook het opschrift van het label telkens wordt aangepast.

Het label doet dienst als actieluisteraar voor een timer die het zelf beheert. De timer vermindert de waarde van *value* elke seconde met 1, totdat de waarde nul wordt bereikt, waarna de timer wordt stilgelegd.

```
public void actionPerformed (ActionEvent e) {
    if (value > 0)
        setValue (value − 1);
    if (value == 0)
        timer.stop ();
}
```

De timer wordt aangemaakt in de constructor en opgestart in de methode *start* van het paneel.

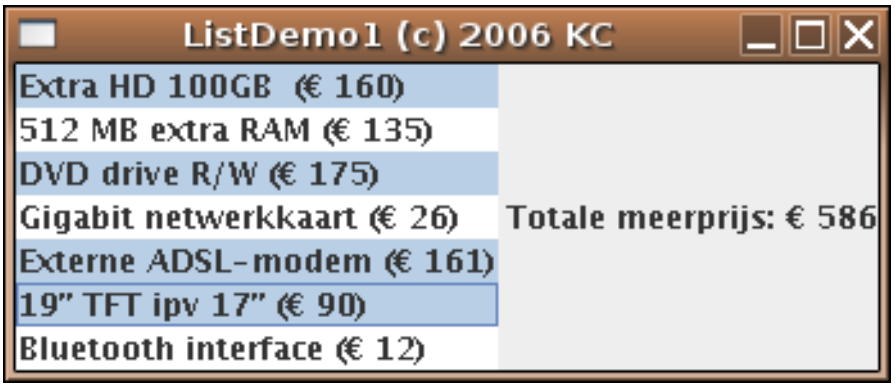
```
public CountdownPanel (int maxValue) {
    super (null);
    this.maxValue = maxValue;
    value = maxValue;
    timer = new Timer (1000, this);
}

public void start () {
    setValue (maxValue);
    timer.restart  ();  // of timer.start ();
}
```

Een timer is niet zo precies als de milliseconden in de constructorparameter doen vermoeden. Timergebeurtenissen worden soms uitgesteld wanneer de machine het te druk heeft, of kunnen zelfs worden weggelaten als een tweede gebeurtenis zich voordoet vooraleer de eerste is verwerkt. Om meer precisie te bekomen, kan je eventueel als onderdeel van de verwerking van de timergebeurtenis de werkelijke tijd opvragen met behulp van *System.currentTimeMillis*.

```
public void actionPerformed (ActionEvent e) {
    long current = System.currentTimeMillis ();
    long trueInterval = current − last;
    last  = current;
    // doe iets met trueInterval
     ...
}
```

Het pakket *java.util* bevat ook een klasse met de naam *Timer*. Deze klasse heeft ongeveer dezelfde functionaliteit, maar wordt op een andere manier gebruikt. Ze hoort eerder thuis bij niet-grafische toepassingen.

2.4. Keuzelijsten



Wanneer je de gebruiker een keuze tussen verschillende mogelijkheden wil aanbieden, kan je hiervoor een groep van schakel- of radioknoppen gebruiken, maar ook een lijst of een combobox. Omdat een combobox eigenlijk niets anders is dan een combinatie van een knop en een lijst die verschijnt wanneer je op de knop drukt, zullen we ons hier beperken tot eenvoudige lijsten. Een voorbeeld van een combobox kwam reeds aan bod in hoofdstuk 1.

Met de term *keuzelijst* duiden we een lijst aan met vaste keuzemogelijkheden die tijdens de loop van het programma niet veranderen. De klasse *JList* waarmee keuzelijsten in Swing worden voorgesteld, kan echter ook voor andere, meer complexe toepassingen worden ingezet. Hiervoor verwijzen we naar een later hoofdstuk (§9.1).

Net zoals bij een combobox, geef je bij een lijst de verschillende keuzemogelijkheden op als parameter van de constructor, in de vorm van een tabel van strings.

```
String[] items = { ... };
...
JList list = new JList (items);
```

Swing laat hier eigenlijk een tabel van willekeurige objecten toe, maar herkent slechts de types *String* en *Icon*.

Wanneer de gebruiker een selectie maakt in de lijst, veroorzaakt dit een gebeurtenis van het type *ListSelectionEvent*. Een dergelijke gebeurtenis vang je op met een luisteraar van het type *ListSelectionListener*.

```
list .addListSelectionListener
    (new ListSelectionListener (){
        public void valueChanged (ListSelectionEvent e) {
            if (! list .getValueIsAdjusting()) {
                int total = 0;
                for (int i=0; i < items.length; i++)
                    if ( list .isSelectedIndex (i ))
                        total += prices[i ];
                label .setText ("Totale meerprijs: \u20ac " + total);
            }
        }
    });
```

De methode *valueChanged* van een dergelijke luisteraar wordt opgeroepen telkens wanneer er iets aan de huidige selectie verandert. Meestal zijn we enkel geïnteresseerd in 'definitieve' veranderingen. Vandaar dat we in het bovenstaande fragment de methode *getValueIsAdjusting* hebben opgeroepen. Deze methode heeft waarde **true** wanneer de verandering in selectie nog niet voorbij is — m.a.w., wanneer de muisknop tijdens het selecteren nog niet is losgelaten.

Je kan de methode *isSelectedIndex* gebruiken om na te gaan of de keuzemogelijkheid met het gegeven volgnummer is geselecteerd. Als alternatief kan je ook een lijst opvragen van alle geselecteerde volgnummers:

```
public void valueChanged (ListSelectionEvent e) {
    if (! list .getValueIsAdjusting()) {
        int total = 0;
        for (int index : list .getSelectedIndices ())
            total += prices[index];
        label .setText ("Totale meerprijs: \u20ac " + total);
    }
}
```

Andere bruikbare methoden van *JList* zijn *getSelectedIndex* die de *eerste* geselecteerde index teruggeeft of -1 als er niets is geselecteerd (handig wanneer je slechts één selectie toelaat), *getSelectedValue* die rechtstreeks het *object* teruggeeft dat is geselecteerd in plaats van de index van dit object (wellicht moet dit achteraf worden ge*cast* naar het type *String*) en tenslotte *getSelectedValues* die een tabel met alle geselecteerde objecten teruggeeft.

Normaal kan je in een lijst tegelijkertijd meerdere items selecteren. Dit doe je bijvoorbeeld door de muisknop in te drukken samen met de CTRL- of de SHIFT-

Met elke component kan je een zogenaamde *tool tip* associëren. Dit is een korte tekst die automatisch bovenaan de component verschijnt wanneer de muis gedurende een zekere tijd boven die component blijft stilstaan. Je doet dit met een opdracht zoals deze:

```
list.setToolTipText("Selecteer hier je aankopen");
```

toets. Je kan dit echter wijzigen door een gepaste selectiemodus in te stellen. Er zijn drie opties: je laat een volledig willekeurige selectie toe (het defaultgedrag), een aaneensluitend interval van selecties, of slechts één selectie tegelijkertijd. Je stelt dit in met *setSelectionMode*:

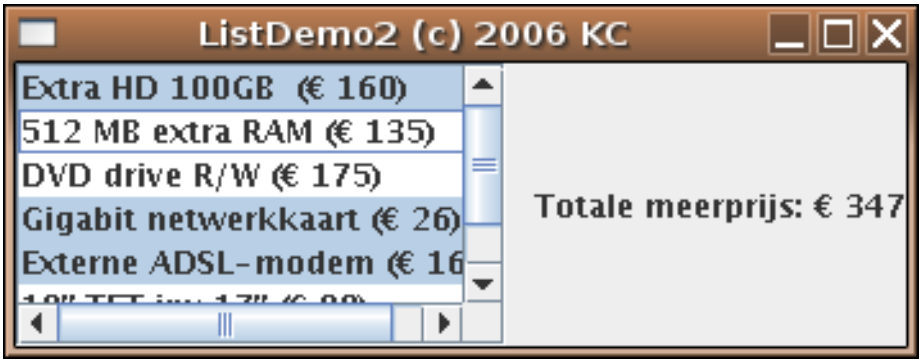
```
list.setSelectionMode
    (ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
list.setSelectionMode
    (ListSelectionModel.SINGLE_INTERVAL_SELECTION);
list.setSelectionMode (ListSelectionModel.SINGLE_SELECTION);
```

Een combobox laat steeds slechts één enkele selectie toe.

Tot slot merken we op dat een keuzelijst niet automatisch over scrollbars beschikt. Je kan dit oplossen door de lijst zelf in een zogenaamd *scrollpaneel* te stoppen (van het type *JScrollPane*). Je doet dit bijvoorbeeld met de volgende eenvoudige **new**-opdracht:

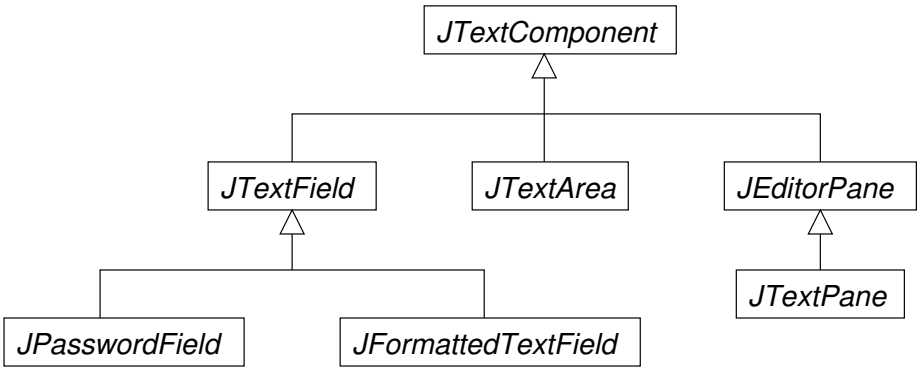
```
JList list = new JList (items);
window.getContentPane().add (new JScrollPane (list));
```

Het effect zie je in onderstaande momentopname.



2.5. Tekstvelden en tekstgebieden

Onderstaand klassendiagram geeft een overzicht van alle tekstcomponenten die Swing rijk is. We beperken ons in deze tekst echter tot de meest eenvoudige: tekstgebieden (van de klasse *JTextArea*), tekstvelden (*JTextField*), wachtwoordvelden (*JPasswordField*) en geformatteerde tekstvelden (*JFormattedTextField*). Omdat deze klassen allemaal uitbreidingen zijn van de klasse *JTextComponent* is ook hun gedrag zeer gelijkaardig.



2.5.1. Inleiding

Een tekstveld dient voor het intikken en editeren van één enkele lijn tekst. Een wachtwoordveld heeft hetzelfde doel, maar beeldt sterretjes af voor elke letter die je intikt. Een tekstgebied kan tekst bevatten die uit meerdere lijnen bestaat.

Ter illustratie bespreken we een kort programma dat de lijnen die worden ingetikt in een tekstveld, onder elkaar afdrukt in een tekstgebied op hetzelfde venster. Een knop onderaan laat toe om het tekstgebied terug leeg te maken (zie figuur op de volgende bladzijde).

De tekstveldconstructor die het meest wordt toegepast, neemt als enige parameter het aantal lettertekens dat dit tekstveld moet kunnen afbeelden. Het aantal lettertekens dat het tekstveld kan *bevatten* is daarentegen (in theorie) onbegrensd.

field = **new** *JTextField* (30);

Je kan ook een string als extra parameter opgeven met de tekst die alvast in het tekstveld moet worden ingevuld.

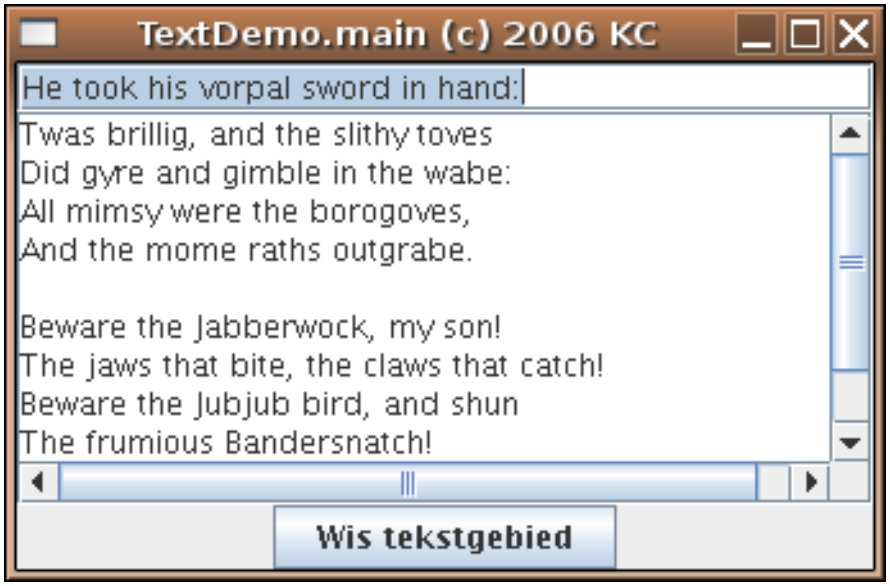
field = **new** *JTextField* ("Voorbeeldtekst", 30);

De constructoren van *JPasswordField* hebben dezelfde vorm.

Bij tekstgebieden (*JTextArea*) neemt de constructor twee numerieke parameters: het aantal lijnen en het aantal tekens per lijn.

area = **new** *JTextArea* (10, 30);

Ook in dit geval kan je vooraan een string als extra parameter opgeven. Deze string bevat de tekst die op voorhand in het tekstveld moet worden ingevuld. Lijnen in deze string worden gescheiden door *linefeed*-tekens (genoteerd als `\n`).



In onze toepassing willen we niet dat de gebruiker ook de inhoud van het tekstgebied kan veranderen (editeren). We schakelen deze optie daarom uit met de volgende opdracht:

area.setEditable (**false**);

Tekstgebieden bezitten geen scrollbars van zichzelf. Daarom plaats je het tekstgebied best in een scrollpaneel — net zoals bij een keuzelijst.

cp.add (**new** *JScrollPane* (*area*));

Wanneer de gebruiker een tekstveld (of een wachtwoordveld) heeft ingetikt, genereert Swing bij het indrukken van de afsluitende ENTER-toets een actiegebeurtenis. Je vangt die op met een luisteraar zoals gebruikelijk.

In ons voorbeeld wordt dit:

```
public void actionPerformed (ActionEvent e) {
    area.append (field.getText() + "\n");
    field .setCaretPosition (field .getText().length());
    field .moveCaretPosition (0);                    // zie §2.5.2
}
```

Met de methode *append* kan je tekst toevoegen aan het einde van een tekstgebied en met *insert* kan je tekst tussenvoegen — notatie: *area.insert(string, pos)*.

Met *setText* kan je de inhoud van een tekstgebied (of een tekstveld) wijzigen in een opgegeven string. De waarde **null** heeft hetzelfde effect als een lege string en maakt het gebied dus leeg, zoals bijvoorbeeld in de luisteraar van de wis-knop:

```
public void actionPerformed (ActionEvent e) {
    area.setText (null);
}
```

Bij *JPasswordField* heet deze methode *setPassword*. De overeenkomstige methoden *getText* en *getPassword* geven je de huidige inhoud van de tekstcomponent terug.

Er zijn verschillende manieren om tekst af te beelden die niet rechtstreeks door de gebruiker moet worden aangepast.

- Je plaatst de tekst in een label van de klasse *JLabel*.
- Je gebruikt een tekstveld of tekstgebied waarvan je hebt gezorgd dat het niet editeerbaar is (zie tekst).
- Je 'tekent' de tekst zelf op een component (zie §5.1.5).

2.5.2. Het inlasteken

De methode *moveCaretPosition* verplaatst het zogenaamde *inlasteken* (Engels: *caret*). Dit duidt in een tekstcomponent de positie aan waar het volgende letterteken dat wordt ingetikt, zal worden tussengevoegd. (Visueel wordt het inlasteken vaak als een flikkerende verticale streep afgebeeld.)

Intern wordt dit inlasteken opgeslagen als een object dat *twee* verschillende posities in de tekst bijhoudt: het *punt* (Engels: *dot*) en het merkteken (*mark*). Het punt correspondeert met het zichtbare inlasteken, het merkteken is niet direct zichtbaar, maar de tekst die zich tussen punt en merkteken bevindt, wordt beschouwd als *geselecteerde* tekst en op het scherm afgebeeld met een gekleurde achtergrond. Wanneer punt en merkteken hetzelfde zijn, is er geen tekst geselecteerd.

Er zijn twee methoden om het inlasteken vanuit het programma te verplaatsen:

public void *setCaretPosition* (**int** *pos*);
public void *moveCaretPosition* (**int** *pos*);

De methode *setCaretPosition* verplaatst zowel punt als merkteken naar dezelfde opgegeven positie. De methode *moveCaretPosition* verplaatst enkel het punt en laat het merkteken achter waar het zich bevond. Op die manier verandert *moveCaretPosition* ook de huidige selectie. In ons voorbeeld heeft dit als gevolg dat de gebruiker zijn vorige invoer niet zal hoeven uit te wissen vooraleer hij een nieuwe lijn in het tekstveld invoert. De geselecteerde tekst zal namelijk automatisch worden vervangen door het eerste letterteken dat hij intikt.

2.5.3. Focus

In een GUI-omgeving is er op elk moment slechts één welbepaalde component op één welbepaald venster van één welbepaalde toepassing die invoer van het toetsenbord ontvangt. Men zegt dat deze component de *focus* bezit. De focus wordt vaak visueel aangeduid doordat er een kader rond de bewuste component wordt getekend of door een andere achtergrondkleur.

De focus kan op verschillende manier worden verplaatst, zowel rechtstreeks door de gebruiker (met een muisklik of door een druk op de TAB-toets) of door de toepassing zelf. In Swing kan je hiervoor de methode *requestFocusInWindow* oproepen van de component die de focus moet krijgen.

De abstracte klasse *FocusAdapter* implementeert de interface *FocusListener* op de meest triviale manier: het corpus van zowel *focusGained* als *focusLost* bevat geen enkele opdracht. Je kan deze klasse uitbreiden om je eigen focusluisteraar te definiëren. Dit is vooral interessant waneer je slechts één van de twee methoden *focusLost* en *focusGained* wenst in te vullen. Als je daarentegen *FocusListener* rechtstreeks implementeert, moet je ook de andere methode vermelden — met een lege definitie.

Swing biedt dergelijke adapterklassen aan voor elke luisteraarsinterface die meer dan één methode bevat.

Vaak is het nuttig te weten wanneer een component de focus krijgt of verliest. Dit doe je door een focusgebeurtenis (klasse *FocusEvent*) op te vangen met een focusluisteraar (interface *FocusListener*). Deze luisteraar moet de volgende twee methoden implementeren:

```
public void focusGained (FocusEvent e);
// Opgeroepen wanneer de component de focus krijgt.

public void focusLost (FocusEvent e);
// Opgeroepen wanneer de component de focus verliest.
```

De volgende focusluisteraar zorgt er bijvoorbeeld voor dat de inhoud van een tekstveld wordt geselecteerd zodra het veld de focus verwerft.

```
private class MyFocusListener extends FocusAdapter {
    public void focusGained (FocusEvent e) {
        if (! e.isTemporary ()) {
            JTextField field = (JTextField)e.getSource();
            field .setCaretPosition (field .getText (). length ());
            field .moveCaretPosition (0);
        }
    }
}
```

De **if**-opdracht negeert *tijdelijke* focusveranderingen, bijvoorbeeld wanneer de gebruiker een venster van een andere toepassing activeert.

2.5.4. Valideren van invoer

De methode *focusLost* kan je onder andere gebruiken om invoer te *valideren*. In de toepassing die hieronder is afgebeeld, kan je een IP-adres intikken. Een geldig IP-adres bestaat uit vier gehele getallen in het bereik 0 t.e.m. 255.

Het programma controleert de inhoud van de vier overeenkomstige tekstvelden telkens wanneer ze de focus verliezen. Dit gebeurt met een luisteraarobject van de binnenklasse *Listener* (zie hieronder). We gebruiken hetzelfde object voor de vier verschillende velden.

(De klassenmethode *stringToByte* is een hulproutine van de toepassing die een string omzet naar een getal tussen 0 en 255, grenzen inbegrepen. Ze geeft -1 terug als de string een ongeldige waarde bevat.)

```
public class Listener implements FocusListener {

    public void focusLost (FocusEvent e) {
        if (! e.isTemporary ()) {
            JTextField field = (JTextField)e.getSource();
            int value = stringToByte (field.getText ());
            if (value < 0) {
                JOptionPane.showMessageDialog
                    (field,
                     "De waarde van dit veld moet liggen\n" +
                     "tussen 0 en 255 (grenzen inclusief)",
```

```
                    "Ongeldige invoer",
                    JOptionPane.ERROR_MESSAGE);
                field.requestFocusInWindow();
            }
        }
    }

    public void focusGained(FocusEvent e) {
        ...
    }
}
```

De methode *JOptionPane.showMessageDialog* zorgt voor het foutbericht in een afzonderlijk dialoogvenster. We bespreken deze routine opnieuw in paragraaf §7.1.

2.5.5. De klasse *InputVerifier*

Een andere manier om invoer te valideren is door middel van de (abstracte) klasse *InputVerifier*. Wanneer je een object van dit type met een component associeert (met behulp van de methode *setInputVerifier*), roept Swing automatisch de methode *shouldYieldFocus* op van dit object telkens wanneer de component de focus verliest. Je moet deze methode zo implementeren dat ze **false** teruggeeft bij elke ongeldige invoer.

Daarnaast bevat *InputVerifier* ook nog een methode *verify* met ongeveer dezelfde functie. *Verify* mag echter geen neveneffecten hebben (zoals het openen van een venster) en *shouldYieldFocus* wel. De defaultimplementatie van *shouldYieldFocus* roept gewoon *verify* op en geeft dezelfde waarde terug.

In de onderstaande code slaan we twee vliegen in één klap: de klasse *Manager* dient tegelijkertijd als focusluisteraar en om de invoer te valideren. De vier velden voor het IP-adres worden op de volgende manier aangemaakt:

```
Manager manager = new Manager();
for (int i=0; i < 4; i++) {
    field[i] = new JTextField("0", 3);
    field[i].addFocusListener(manager);
    field[i].setInputVerifier(manager);
}
```

De klasse *Manager* implementeert *verify* en *shouldYieldFocus* als volgt:

```
public class Manager extends InputVerifier implements FocusListener {

    public boolean verify (JComponent comp) {
        JTextField field = (JTextField)comp;
        return stringToByte (field.getText ()) >= 0;
    }

    public boolean shouldYieldFocus (JComponent comp) {
        if (! verify (comp)) {
            comp.setInputVerifier (null); // [!] Opgelet!
            JOptionPane.showMessageDialog
                (comp,
                 "De waarde van dit veld moet liggen\n" +
                 "tussen 0 en 255 (grenzen inclusief)",
                 "Ongeldige invoer",
                 JOptionPane.ERROR_MESSAGE);
            comp.setInputVerifier (this); // [!] Opgelet!
            return false;
        }
        else
            return true;
    }

    ... // implementatie van FocusListener
}
```

Let in het bijzonder op de twee programmalijnen die met [!] zijn gemarkeerd. We moeten de *InputVerifier* tijdens *shouldYieldFocus* tijdelijk uitschakelen, omdat het tonen van een dialoogvenster zelf de focus van de toepassing verandert en dit mag niet tot gevolg hebben dat *shouldYieldFocus* opnieuw wordt opgeroepen (met een oneindige recursie tot gevolg).

2.5.6. Geformatteerde tekstvelden

In plaats van invoer achteraf te valideren, kan je er ook voor kiezen om het de gebruiker onmogelijk te maken om verkeerde invoer in te tikken. Als je bijvoorbeeld enkel gehele getallen als invoer wil toestaan, zorg er dan voor dat het tekstveld enkel cijfertoetsen aanvaardt.

Hoewel dit niet zo eenvoudig is, is het in Swing inderdaad mogelijk om rechtstreeks in te grijpen in welke strings er al dan niet in een tekstveld kunnen worden ingebracht. Het zou ons echter te ver leiden om deze techniek hier te behandelen. De klasse *JFormattedTextField* biedt echter een eenvoudig alternatief.

Een dergelijke component, een *geformatteerd tekstveld*, ziet eruit als een gewoon tekstveld, maar laat enkel toe om bepaalde strings in te brengen. Het is bij constructie van het geformatteerd tekstveld dat de programmeur bepaalt welke invoer er allemaal is toegelaten. In deze tekst geven we hiervan één voorbeeld: een geformatteerd tekstveld voor de invoer van getallen. Dit is echter maar een greep uit het ruime aanbod.

Waar je bij *JTextField* de methoden *getText* en *setText* gebruikt om de inhoud van het tekstveld op te vragen of te wijzigen, pas je hiervoor bij *JFormattedTextField* de volgende methoden toe:

public *Object getValue* ();

public void *setValue* (*Object value*);

Het belangrijkste verschil is het gegevenstype: *Object* in plaats van *String*. In de praktijk is dit zelfs altijd een specifieke deelklasse van *Object*, bijvoorbeeld *Date* bij een geformatteerd datumveld, of *Number* bij een numeriek veld.

Dit betekent onder andere dat je geen 'verkeerde' gegevens uit een geformatteerd tekstveld kan uitlezen: zelfs wanneer de gebruiker een alfabetische tekst in een numeriek veld inbrengt, zal *getValue* steeds een object van het type *Number* teruggeven, desnoods een defaultgetal of het getal dat zich in het veld bevond vooraleer de gebruiker er letters aan toevoegde.

Bij een geformatteerd tekstveld moet je steeds instellen welk soort formattering je wenst toe te passen. Dit kan bijvoorbeeld door bij de constructor een object van het gepaste type als parameter mee te geven. We maken bijvoorbeeld op de volgende manier een numeriek veld aan:

```
JFormattedTextField numfield
    = new JFormattedTextField (new Integer (0));
```

of anders

```
JFormattedTextField numfield = new JFormattedTextField ();
numfield.setValue (new Integer (0));
```

Wanneer het object van het type *Number* is — de wikkelklassen *Integer* en *Double* zijn beide deelklassen van *Number* — krijg je een numeriek veld, bij *Date* wordt automatisch een datumveld aangemaakt.

In dit voorbeeld hebben we als initiële waarde het getal nul ingevuld. Dit zorgt voor een veld van één teken breed, wat wellicht niet wenselijk is. Je kan de breedte van een (tekst)veld echter ook expliciet instellen:

```
numfield.setColumns (5);
```

Merk op dat *getValue* een object zal teruggeven van het type *Number*, maar niet noodzakelijk van het type *Integer*, ook als was de eerste waarde van dit type. Het is zelfs mogelijk dat je een *Long*-object terugkrijgt, ook al tikt de gebruiker slechts een klein getal in. Verwerk de waarde dus zoveel mogelijk met behulp van de methoden die bij *Number* zelf horen:

```
Number wrapped = (Number)numfield.getValue ();
int value = wrapped.intValue ();
 ...
```

Wanneer je deze voorbeelden uitprobeert, zal je merken dat het veld nog altijd toelaat om letters in te tikken (ook al geeft het alleen getallen terug). Om te controleren of er toch geen 'verboden' lettertekens zijn ingevoerd, kan je opnieuw een *InputVerifier* toepassen. Er treedt hier echter een belangrijke complicatie op: wanneer de *InputVerifier* wordt opgeroepen, is de waarde van het veld (die je normaal met *getValue* zou opvragen) nog niet aangepast aan de ingetikte lettertekens.

Je roept de methode *commitEdit* op om waarde en tekst op elkaar af te stemmen. Als dit niet lukt omdat de ingetikte tekst niet van het juiste formaat is, veroorzaakt deze methode een uitzondering (van het type *java.text.ParseException*). Je kan hiervan handig gebruik maken.

Als voorbeeld drukken we een methode *verify* af waarmee gecontroleerd wordt of er in het geformatteerde tekstveld een getal tussen 0 en 255 werd ingetikt:

```
public boolean verify (JComponent comp) {
    JFormattedTextField field = (JFormattedTextField)comp;
```

```
        try {
            field .commitEdit ();
            return numberToByte ((Number)field.getValue ()) >= 0;
        }
        catch (ParseException ex) {
            return false;
        }
    }
```

De methode *numberToByte* doet hetzelfde als *stringToByte* uit paragraaf §2.5.4 maar neemt een *Number* als argument in plaats van een string.

2.6. Andere componenten

Er bestaan in Swing nog andere atomaire componenten dan deze die we hier hebben besproken. Zo is er bijvoorbeeld de klasse *JSlider* waarmee je een *schuiver* kan creëeren om getalwaarden in te stellen tussen vooraf opgegeven grenzen. Wanneer de gebruiker de stand van de schuiver verandert, hetzij met de muis, hetzij met bepaalde toetsencombinaties, genereert Swing dit keer niet een actiegebeurtenis maar een zogenaamde veranderingsgebeurtenis (*change event*). Je kan deze gebeurtenis opvangen met een luisteraar van het type *ChangeListener* die je bij de schuiver met *addChangeListener* registreert. De methode die je daarbij moet implementeren heet *stateChanged* (in plaats van *actionPerformed*). Meer informatie vind je, zoals altijd, in de elektronische documentatie.

Voor de meeste andere componenten, zoals tabellen en boomdiagrammen bijvoorbeeld, die je echter een andere techniek te gebruiken (het zogenaamde *MVC patroon*). Dit zullen we uitvoerig bespreken in hoofdstukken 4 en 9.

3. Invoer en uitvoer in Java (deel 1)

We verlaten even het domein van de grafische gebruikersinterfaces om ons te concentreren op een ander aspect van professionele softwaretoepassingen: het gebruik van *persistente informatie*, m.a.w., het uitschrijven van gegevens naar een bestand of naar een databank en het ophalen van gegevens die op een dergelijke manier werden opgeslagen. In dit hoofdstuk bespreken we de standaardklassen voor in- en uitvoer in Java, in hoofdstuk 8 behandelen we enkele meer gevorderde aspecten.

3.1. Standaardklassen voor in- en uitvoer

De standaardklassen voor in- en uitvoer in Java bevinden zich in het pakket *java.io*. Aangezien deze klassenbibliotheek wel 50 klassen en 10 interfaces bevat (uitzonderingsklassen niet meegerekend), kunnen we hier onmogelijk volledig zijn. We raden je dan ook aan om de elektronische documentatie te raadplegen vooraleer je deze klassen in je eigen toepassingen gebruikt.

De meeste klassen uit *java.io* komen in paren: er is een klasse die gebruikt wordt voor invoer en een andere die gebruik maakt van dezelfde technologie en conventies, maar die in de plaats dient voor uitvoer. De enige klasse uit dit pakket die tegelijkertijd invoer- en uitvoerbewerkingen toelaat op dezelfde bron is de *RandomAccessFile*, een niet onbelangrijke klasse die zogenaamde *directe toegang* (*random access*) implementeert voor bestanden.

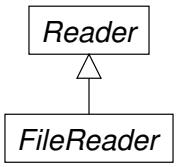
Java maakt een onderscheid tussen gegevens die in *tekstvorm* kunnen worden verwerkt (het getal -12345 neemt dan bijvoorbeeld zes bytes in beslag) of in *binaire* vorm (en dan hebben we aan twee bytes voldoende, als we het type **short** gebruiken). Bij invoer en uitvoer van tekst maken we dan bovendien nog het verschil tussen 'byte-georiënteerd' en 'karaktergeoriënteerd'. Dit komt ongeveer overeen met het onderscheid tussen ASCII en UNICODE, hoewel ook andere internationale coderingen hier een rol spelen.

Behalve bij *RandomAccessFile* gebeurt invoer en uitvoer in Java altijd *sequentieel*. Invoer- en uitvoerkanalen worden beschouwd als gegevens*stromen* (we gebruiken hiervoor de Engelse term *streams*) van en naar de Javatoepassing. Concreet betekent dit bijvoorbeeld dat we het 2000ste element van een invoerstroom niet kunnen bekijken zonder dat we de eerste 1999 elementen hebben ingelezen, of dat we uitvoer naar een stroom achteraf niet zomaar kunnen corrigeren.

Streams kunnen gekoppeld zijn aan bestanden, maar ook aan de standaardinvoeren uitvoerkanalen (toetsenbord en 'opdracht'venster), aan tabellen in het geheugen, aan webpagina's, aan netwerkverbindingen, enz. Het pakket *java.io* biedt ook een aantal hulpklassen dat slechts onrechtstreeks met invoer- en uitvoerbewerkingen heeft te maken.

Zoals alles in Java is ook de invoer/uitvoer-bibliotheek opgebouwd volgens objectgerichte principes, en dat heeft als gevolg dat de klassen uit *java.io* niet onafhankelijk zijn van elkaar. Een bestand waarvan we gegevens willen inlezen, wordt bijvoorbeeld voorgesteld als een object van de klasse *FileReader*, een extensie van de abstracte klasse *Reader*.

Zoals je weet, betekent dit dat je parameters van het type *Reader* ook mag invullen met argumenten van de klasse *FileReader* of dat je een object van de klasse *FileReader* ook kan toewijzen aan een variabele van het type *Reader*. Deze hiërarchische verhouding is soms heel handig.



In het volgende programmafragment lezen we gegevens in van een bestand of van het standaardinvoerkanaal, al naar gelang er een opdrachtlijnparameter aan het programma werd meegegeven.

```
public static void main (String[] args) throws IOException {
    Reader invoer;
    if (args.length == 0)
        invoer = new InputStreamReader (System.in);
    else
        invoer = new FileReader (args[0]);
    ...
}
```

De meeste methodes van de klassen uit *java.io* genereren uitzonderingen. Omdat invoer en uitvoer voor een gedeelte buiten het programma plaatsvindt, is er nu eenmaal meer kans dat er iets fout loopt. Een professioneel programmeur probeert dan ook altijd zo goed mogelijk deze uitzonderingen op te vangen. Zoals

je ziet, durven we dat in deze tekst wel eens 'vergeten' — opdat de voorbeelden eenvoudig zouden blijven.

Invoer/uitvoer-uitzonderingen zijn van het type *IOException*. Deze klasse bezit een aantal deelklassen voor meer specifieke uitzonderingen, zoals bijvoorbeeld *FileNotFoundException* wanneer je een bestand probeert te lezen met een naam die niet bestaat, of *EOFException* wanneer onverwacht het einde van een invoerstroom wordt bereikt.

3.2. Basisstreamklassen

3.2.1. De klassen *Reader* en *Writer*

Vaak zijn de gegevens die via streams worden verwerkt, opgesteld in leesbare tekst: denk maar aan invoer van het toetsenbord, aan uitvoer naar het scherm of aan tekstbestanden. Bij dergelijke streams is het basisgegevenstype waarmee wordt gewerkt de **char** — het 16-bits UNICODE letterteken. Streams die op deze manier functioneren, heten *Reader* of *Writer*, al naargelang het om invoer dan wel om uitvoer gaat.

De klasse *Writer* heeft de volgende basismethode:

public void *write* (**int** *c*) **throws** *IOException*;

Deze methode stuurt het letterteken *c* naar het opgegeven writer-object. Het 'officiële' parametertype van *write* is een 32-bits **int**, hoewel er effectief slechts een 16-bits **char** aan de stream wordt doorgegeven. Door de automatische typeconversies in Java is het echter geen enkel probleem om *write* op te roepen met een argument van het type **char**. Wanneer je tegelijkertijd meerdere lettertekens wil uitschrijven, gebruik je best één van de volgende methodes — dit is doorgaans efficiënter. (Intern maken de laatste drie methodes gebruik van de eerste.)

public void *write* (**char**[] *cbuf*, **int** *off*, **int** *len*) **throws** *IOException*;

public void *write* (**char**[] *cbuf*) **throws** *IOException*;

public void *write* (*String str*, **int** *off*, **int** *len*) **throws** *IOException*;

public void *write* (*String str*) **throws** *IOException*;

Met deze methodes kan je in één keer een gedeelte van een *String* of van een tabel van lettertekens naar een writer doorsturen. De parameter *off* geeft de index in de string of de tabel aan van het eerste letterteken dat moet worden verstuurd. De parameter *len* bepaalt het *aantal* lettertekens. Wanneer je deze parameters weglaat, wordt de volledige tabel of de volledige string weggeschreven.

Vaak zijn streams in Java 'gebufferd' (zie ook §3.2.5). Dit betekent dat de stream de tekens die hij ontvangt niet onmiddellijk uitschrijft, maar wacht totdat hij een zeker aantal tekens heeft verzameld in een interne tabel (een buffer). Met de methode *flush* kan je de stream verplichten om deze buffer meteen uit te schrijven — vol of niet. Er bestaat ook een methode *close* die de stream eerst 'flusht' en daarna afsluit. (Een afgesloten stream laat verder geen enkele invoer- of uitvoerbewerking meer toe.)

public void *flush*() **throws** *IOException*;

public void *close*() **throws** *IOException*;

Bij uitvoer is het belangrijk dat je een stroom daadwerkelijk afsluit met *close* nadat je de laatste gegevens er naartoe hebt geschreven. De kans is namelijk groot dat de laatste kilobytes anders niet worden geregistreerd.

De basismethode van de klasse *Reader* heet *read*:

public int *read* () **throws** *IOException*;

Deze methode leest één enkel 16-bits letterteken van het opgegeven stream-object en geeft dit terug als waarde. Deze methode geeft -1 terug wanneer het einde van de stream wordt bereikt. Om dit mogelijk te maken heeft *read* als waardetype **int** in plaats van **char**. Soms betekent dit dat je in je programma het resultaat expliciet van type moet veranderen:

```
String result = "";
for (int i=0; i < 10; i++)
    result += (char)invoer.read();
```

Je kan ook meerdere lettertekens in één keer inlezen:

public int *read* (**char**[] *cbuf*, **int** *off*, **int** *len*) **throws** *IOException*;

public int *read* (**char**[] *cbuf*) **throws** *IOException* ;

De eerste methode probeert *len* lettertekens in te lezen en stopt die in de tabel *cbuf* vanaf index *off*. De tweede probeert de opgegeven tabel *volledig* op te vullen.

Merk op dat Java niet garandeert dat het gevraagde aantal lettertekens ook effectief wordt ingelezen. Daarom geven beide methodes het werkelijke aantal steeds terug als waarde. Deze waarde is negatief wanneer het einde van de invoerstroom wordt bereikt. In principe kan de waarde nooit nul zijn: *read* wacht totdat er minstens één letterteken ter beschikking staat.

De klasse *Reader* bevat ook een methode *close* die de reader afsluit.

```
public void close () throws IOException;
```

De volgende eenvoudige methode kopieert de inhoud van de ene stream naar de andere, in blokken van 256 lettertekens.

```
public static void streamCopy (Reader invoer, Writer uitvoer)
        throws IOException {
    char[] buffer = new char[256];
    int len = invoer.read (buffer);
    while (len >= 0) {
        uitvoer.write (buffer, 0, len);
        len = invoer.read (buffer);
    }
}
```

3.2.2. De klassen *FileReader* en *FileWriter*

De objecten van de klasse *FileReader* en *FileWriter* zijn de readers en writers die specifiek gebruikt worden voor de sequentiële verwerking van tekstbestanden. Omdat deze klassen extensies zijn van *Reader* en *Writer* ondersteunen ze ook alle bewerkingen uit die klasse. Beide klassen hebben een constructor die een bestandsnaam als parameter neemt en ook een constructor die een *File* als parameter neemt (zie §3.4.2).

Omdat de meeste besturingssystemen tekstbestanden nog steeds als ASCII-code opslaan — of als één of andere 8-bits extensie van die code — treedt er hier echter een belangrijke complicatie op. Bekijk bijvoorbeeld het volgende korte voorbeeldprogramma:

```
public class Alfabet {

    public static void main (String[] args) throws IOException {
        Writer uitvoer = new FileWriter ("alfabet.out");
        for (char ch='A'; ch <= 'Z'; ch++)
            uitvoer.write (ch);
        uitvoer.close  ();
    }
}
```

Dit programma stuurt de opeenvolgende lettertekens `A` t.e.m. `Z` naar het tekstbestand *alfabet.out*. We stellen ons de vraag hoeveel bytes het bestand *alfabet.out* zal bevatten na afloop van dit programma.

Omdat Java, en dus ook de klasse *FileWriter*, intern met UNICODE werkt, kan je redelijkerwijs verwachten dat het programma 52=2×26 bytes genereert. Als je het programma echter uitprobeert, blijken het er slechts 26 te zijn. De klasse *FileWriter* zet immers als laatste stap de interne UNICODE om naar de code die door het lokale bedrijfssysteem wordt gehanteerd, en dit is in de meeste gevallen nog steeds ASCII. Omgekeerd zet de klasse *FileReader* eerst de lokale code om naar UNICODE vooraleer gegevens aan je programma door te geven.

Zolang je je tot het Engelse alfabet beperkt, zal deze manier van werken je wellicht geen problemen bezorgen, maar met vreemde lettertekens en symbolen kan je toch onverwachte resultaten bekomen. We geven een oplossing voor dit probleem in de volgende paragraaf.

3.2.3. De klassen *InputStream* en *OutputStream*

Naast *Reader* en *Writer* bestaan er ook stream-klassen die **byte**-georiënteerd zijn in plaats van **char**-georiënteerd — de klassen *InputStream* en *OutputStream*. Deze klassen zijn nuttig bij het sequentieel verwerken van binaire bestanden zoals GIF-afbeeldingen of MIDI-bestanden, t.t.z., bestanden die niet onmiddellijk uit leesbare tekst hoeven te bestaan.

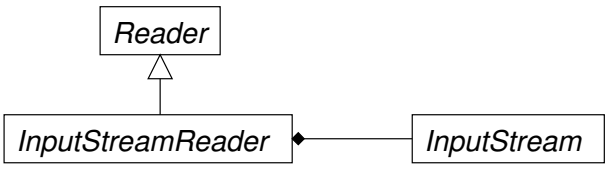
De *read*-methode van *InputStream* leest één enkele byte in en geeft die terug in de vorm van een **int** — met waarde -1 wanneer het einde van de stroom werd bereikt. De *write*-methode van *OutputStream* neemt een getal van het type **int** als argument en schrijft daarvan één enkele byte (de minst significante) naar de uitvoer.

In principe kan je deze klassen ook toepassen bij verwerking van tekstuele gegevens in ASCII-code (zoals programmabroncode) maar het getuigt van een betere programmeerstijl om in dat geval toch een *Reader* of een *Writer* te gebruiken. Jammergenoeg gebruikt Java — om historische redenen — nog steeds *InputStream* en *OutputStream* voor de standaardinvoer- en uitvoerkanalen (t.t.z., de objecten *System.in*, *System.out* en *System.err*).

Er bestaat een eenvoudige manier om deze complicatie te omzeilen: gebruik de klasse *InputStreamReader* om van een inputstream een reader te maken, en gebruik *OutputStreamWriter* om een outputstream naar een writer om te zetten.

```
InputStreamReader invoer = new InputStreamReader (System.in);
```

De constructor voor *InputStreamReader* neemt een willekeurige inputstream als argument en construeert daaruit een nieuwe reader. Elke invoeroperatie op een inputstreamreader wordt automatisch doorgegeven aan de geassocieerde inputstream.



Bij het omzetten van een *InputStream* of *OutputStream* naar een *InputStreamReader* of *OutputStreamReader* kan je ook een *encodering* opgeven die aangeeft op welke manier internationale lettertekens worden omgezet naar UNICODE (en omgekeerd):

```
Reader invoer = null;
try {
    invoer = new InputStreamReader (System.in, "ISO-8859-1");
}
catch (UnsupportedEncodingException ex) {
    System.err.println ("Onbekende encodering: " + ex.getMessage());
    System.exit (0);
}
```

ISO-8859-1 is de officiële afkorting voor de codering die in grote delen van Europa wordt gebruikt en ook gekend staat als LATIN1.

3.2.4. De klasse *PrintWriter*

Je kan een analoge constructie gebruiken om van een outputstream een outputstreamwriter te maken, maar doorgaans gebruikt men meteen de klasse *PrintWriter* waarvan de bewerkingen iets meer geavanceerd zijn. (*PrintWriter* is *geen* extensie van *OutputStreamWriter*.)

De basismethodes heten *print* en *println* en laten toe om getallen, lettertekens en in principe alle objecten in tekstvorm uit te schrijven — op dezelfde manier waarop we gewoon zijn met *System.out* te werken. (Nochtans behoort *System.out* tot de klasse *PrintStream* en niet tot *PrintWriter*.) De *print*- en *println*-methodes hebben het bijkomende voordeel dat ze geen uitzonderingen genereren.

Je kan elke writer tot een *PrintWriter* omvormen met behulp van één van de volgende constructoren:

public *PrintWriter* (*Writer w*);

public *PrintWriter* (*Writer w*, **boolean** *autoflush*);

Wanneer je het *autoflush*-argument als waarde **true** geeft, zal de printwriter automatisch een *flush* oproepen na elke *println*. Bij de eerste constructor is *autoflush* **false**.

Er bestaan ook gelijkaardige constructoren die een outputstream als argument nemen.

public *PrintWriter* (*OutputStream out*);

public *PrintWriter* (*OutputStream out*, **boolean** *autoflush*);

Achter de schermen wordt de outputstream eerst omgezet naar een outputstreamwriter en dan verder naar een gebufferde writer (zie §3.2.5):

```
// Uit de broncode van de klasse PrintWriter
public PrintWriter (OutputStream out, boolean autoFlush) {
    this (new BufferedWriter (new OutputStreamWriter (out)), autoFlush);
}
```

Sinds Java 5.0 bevat *PrintWriter* een aantal *printf* methoden die nog meer flexibiliteit bieden voor geformateerde uitvoer. Voor meer informatie verwijzen we naar de elektronische documentatie.

3.2.5. Filterstreamklassen

Een stream die gegevens inleest of uitschrijft van of naar een andere stream noemen we een *filterstream*. *InputStreamReader* is een voorbeeld van een dergelijke klasse, alsook de gebufferde streams die we hieronder zullen bespreken. Je kan ook je eigen filterstreamklassen bouwen door één van de klassen *FilterReader*, *FilterWriter*, *FilterInputStream* of *FilterOutputStream* uit te breiden.

Een invoerfilter leest gegevens van een andere stream en verwerkt die eerst vooraleer ze aan het programma door te geven. Wanneer we bijvoorbeeld met *read* een letterteken vragen aan een *InputStreamReader* dan wordt er een byte van de corresponderende *InputStream* gehaald (of meerdere bytes, afhankelijk van de gebruikte internationale codering) en omgezet naar één enkel UNICODE-karakter.

Omgekeerd schrijft een uitvoerfilter gegevens naar een geassocieerde uitvoerstroom nadat ze eerst op één of andere manier werden bewerkt.

Het is ook toegelaten om meerdere filters achter elkaar te koppelen, en dit wordt vaak toegepast — zoals in het volgende voorbeeld:

Reader in = **new** *BufferedReader* (**new** *InputStreamReader* (*System.in*));

Wanneer we een leesopdracht aan deze variabele *in* doorgeven, worden een aantal opeenvolgende bytes die van het standaardinvoerkanaal afkomstig zijn eerst omgezet naar karakters en daarna ook nog gebufferd.

Filters van het type *BufferedWriter* (of het byte-gerichte *BufferedOutputStream*) laten toe om streamgegevens te 'bufferen': in de plaats van informatie telkens in kleine hoeveelheden naar een uitvoerstroom te sturen, wordt die informatie tijdelijk opgeslagen in een interne buffer die pas wordt uitgeschreven wanneer hij vol is.

Ook een invoerstroom kan nut hebben van een dergelijke buffer (door middel van een filter van het type *BufferedReader* of *BufferedInputStream*). Als je één letterteken nodig hebt van een invoerstroom kan het efficiënter zijn om meteen een veel groter aantal lettertekens in te lezen en het overschot tijdelijk in het geheugen op te slaan, vanwaar het dan later veel sneller kan worden geraadpleegd.

Je maakt van een reader of writer een dergelijke gebufferde reader of writer met behulp van constructoren zoals deze:

BufferedReader invoer = **new** *BufferedReader* (*reader*);

BufferedWriter uitvoer = **new** *BufferedWriter* (*writer*);

Behalve om te bufferen, wordt *BufferedReader* ook vaak gebruikt omwille van de zeer bruikbare methode *readLine* die de klasse aanbiedt:

public *String readLine*() **throws** *IOException*;

Deze routine leest een volledige lijn in van de invoerstroom en geeft die terug als string — nadat de linefeed en eventuele carriage return zijn verwijderd. De waarde van *readLine* is **null** aan het einde van de invoerstroom.

In het volgende voorbeeld berekenen we de som van een aantal ingelezen getallen. De gebruiker geeft één enkel getal op per lijn en eindigt met een lege lijn of door de stream af te sluiten.

```
public static void main (String[] args) throws IOException {
    BufferedReader invoer =
        new BufferedReader (new InputStreamReader (System.in));
    double som = 0.0;
    String lijn  = invoer.readLine ();
    while (lijn != null && lijn.length() != 0) {
        som += Double.parseDouble (lijn);
        lijn  = invoer.readLine ();
    }
    System.out.println ("De som van deze getallen is " + som);
}
```

3.2.6. Bestanden lezen uit het class path

In het voorbeeld uit paragraaf §2.4 introduceerden we een keuzelijst met goederen en hun overeenkomstige prijzen. De inhoud van deze lijst was 'hard gecodeerd' in het programma. Het zou iets handiger zijn als we deze gegevens uit een afzonderlijke bestand zouden kunnen halen, dat we dan kunnen aanpassen telkens wanneer er aan de prijzen iets verandert, of wanneer er nieuwe goederen ter beschikking komen.

We gebruiken een bestand *goederen.txt* met de volgende inhoud:

```
160 Extra HD 100GB
135 512 MB extra RAM
175 DVD drive R/W
 26 Gigabit netwerkkaart
161 Externe ADSL-modem
 90 19" TFT ipv 17"
 12 Bluetooth interface
```

Dit is een tekstbestand met op elke lijn eerst de prijs en daarna een tekstuele beschrijving.

De verwerking van dit bestand gebeurt in twee fasen. Eerst lezen we het bestand lijn per lijn in met behulp van een *BufferedReader* en slaan deze lijnen op in een lijst van strings.

```
BufferedReader reader = ...                    // Zie verder
ArrayList<String> lines = new ArrayList<String> ();
String line = reader.readLine ();
while (line != null) {
    lines.add (line);
    line = reader.readLine ();
}
reader.close ();
```

Daarna overlopen we de lijst en maken we de twee tabellen aan die we in het programma nodig hebben: een tabel *items* met de beschrijvingen van de goederen, en een tabel *prices* (van gehele getallen) met de prijzen.

```
int aantal = lines.size ();
items = new String[aantal];
prices = new int[aantal];
for (int i=0; i < aantal; i++) {
    line = lines.get (i);
    int pos = 0;
    while (line.charAt (pos) == ' ')
        pos ++;
    int pos2 = line.indexOf (' ', pos);
    prices[i] = Integer.parseInt (line.substring (pos, pos2));
    items[i] = line.substring (pos2+1) + " (\u20ac " + prices[i] + ")";
}
```

Merk op dat we de twee fasen niet tot één kunnen combineren omdat we niet kunnen voorspellen hoeveel lijnen het bestand bevat vooraleer we het volledig hebben gelezen.

Het opsplitsen van de lijn in een prijs en een beschrijving is nogal bewerkelijk. We zouden dit iets kunnen vereenvoudigen door gebruik te maken van de klasse *Scanner* (nieuw sinds Java 5.0) die enkele extra mogelijkheden biedt voor het inlezen van getallen vanuit een tekstbestand. We verwijzen opnieuw naar de elektronische documentatie voor meer informatie.

Rest ons nog enkel de *reader* aan te maken. Dit kan min of meer op dezelfde manier als in paragraaf §3.2.5, door eerst een *FileReader* aan te maken met de juiste bestandsnaam en die dan om te zetten naar een *BufferedReader*. We moeten enkel nog beslissen waar we het bestand ergens zullen plaatsen.

Omdat het bestand onlosmakelijk met het programma is verborgen, is het class path hiervoor wellicht de beste plaats. Net zoals bij afbeeldingen (zie §1.5) biedt ook hier de klasse *Class* een uitweg. De methode *getResourceAsStream* van *Class* geeft een *InputStream* terug aan de hand van zijn naam in het class path. We plaatsen het bestand *goederen.txt* dus in dezelfde folder als de klasse *ListDemo3* en maken *reader* aan op de volgende manier:

```
InputStream is
    = ListDemo3.class.getResourceAsStream ("goederen.txt");
reader = new BufferedReader (new InputStreamReader (is)));
```

Het mooie van de zaak is dat dit ook werkt wanneer *goederen.txt* samen met de klassebestanden in een JAR-archief werden verpakt.

We zullen dit voorbeeld hernemen in paragraaf §8.2 waar we de prijs- en goedereninformatie in een XML-bestand opslaan, en in paragraaf §8.3 waar we de gegevens rechtstreeks uit een databank halen.

3.3. Binaire gegevensverwerking

3.3.1. De interfaces *DataInput* en *DataOutput*.

Wanneer we gegevens opslaan die niet onmiddellijk leesbaar hoeven te zijn voor een menselijke gebruiker — tussentijdse gegevens bijvoorbeeld, of gegevens die

we toch enkel maar bekijken met behulp van speciale programmatuur — kunnen we vaak schijfruimte besparen en verwerkingssnelheid winnen door bij invoer en uitvoer zoveel mogelijk het oorspronkelijke *binaire* formaat van deze gegevens te behouden. Java voorziet hiervoor een aantal bijzondere klassen. Deze klassen implementeren de interfaces *DataInput* en *DataOutput*.

De interface *DataOutput* voorziet methoden met namen zoals *writeInt* en *writeDouble* die getallen in een binaire vorm uitschrijven naar een uitvoerkanaal. Een geheel getal van het type **int** neemt in Java steeds 32 bits in beslag en wordt daarom door *writeInt* als vier opeenvolgende bytes uitgeschreven. Om dezelfde reden schrijft *writeDouble* steeds acht bytes uit per reëel getal.

De volgorde waarin de vier bytes worden uitgeschreven, is niet echt van belang: Java zorgt er immers voor dat deze op elke computer hetzelfde blijft, ook al is de interne representatie van getallen verschillend van processortype tot processortype. Een binair bestand dat je op een Windows-toestel hebt gecreëerd, is dus evengoed leesbaar op de MacIntosh van je buren — zolang je maar bij Java blijft.

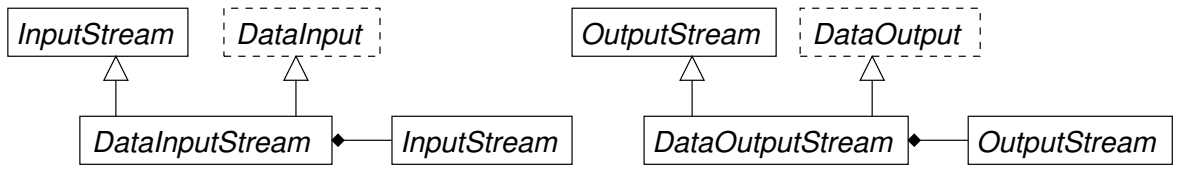
In het volgende voorbeeld slaan we alle priemgetallen kleiner dan 1 miljoen (*MAX*) op in het bestand *priem.dat*, voor later gebruik.

```
public static void main (String[] args) {

    boolean[] samengesteld = new boolean[MAX];

    // vul samengesteld op zodat samengesteld[i] false is
    // enkel en alleen als i priem is
    ...

    try {
        DataOutputStream out =
         new DataOutputStream(new FileOutputStream("priem.dat"));
        for (int i=2; i < MAX; i++)
            if (! samengesteld [i])
                out.writeInt (i );
        out.close ();
    }
    catch (IOException e) {
        System.err.println ("Invoer/uitvoer-fout: " + e);
    }
}
```

Gegevens die met *DataOutput* werden uitgeschreven, lees je opnieuw in met *DataInput*. De invoermethodes hebben namen zoals *readInt* en *readDouble*. Dus, om nu de som te bepalen van alle priemgetallen kleiner dan 1 miljoen, lezen we het bestand opnieuw in, op de volgende manier:

```
public static void main (String[] args) throws IOException {
    long som = 0;
    DataInputStream in = null;
    try {
        in = new DataInputStream
                (new FileInputStream ("priem.dat"));
        for (;;)
            som += in.readInt ();
    }
    catch (EOFException e) {
        System.out.println ("Som = " + som);
    }
    catch (IOException e) {
        System.err.println ("Invoer/uitvoer-fout: " + e);
    }
    finally {
        if (in != null)
            in.close ();
    }
}
```

Merk op dat we de filterstreamklassen *DataInputStream* en *DataOutputStream* gebruiken om van een *FileInputStream* of *FileOutputStream* een object te maken dat de interfaces *DataInput* of *DataOutput* ondersteunt. (Er bestaan geen klassen met de naam *DataReader* of *DataWriter*, dus gebruiken we hier ook niet *FileReader* of *FileWriter*.)



Wanneer je gegevens leest via *DataInput* geeft Java het einde van de invoerstroom niet aan met een speciale waarde (zoals -1 bij *Reader.read*), maar genereert het programma in de plaats een uitzondering van het type *EOFException* — een subklasse van de alomtegenwoordige *IOException*.

De interface *DataInput* bevat een methode *readFully* waarmee je meerdere bytes tegelijkertijd kan inlezen. Er zijn een aantal verschillen tussen deze methode en de gelijkaardige *read* uit *InputStream* (die echter *niet* door *DataInput* wordt ondersteund).

- *readFully* probeert altijd de gevraagde tabel van bytes *volledig* op te vullen. Lukt dit niet omdat het einde van de invoerstroom is bereikt, dan genereert Java een *EOFException*. Lukt dit niet om een andere reden, dan volgt er een meer algemene *IOException*.
- *read* geeft daarentegen een geheel getal terug met het aantal bytes dat effectief werd ingelezen — dit hoeft dus niet de ganse tabel te zijn. Bij het einde van de invoerstroom krijgt *read* de waarde -1. Het terugkeertype van *readFully* is **void**.

Merk op dat *DataInputStream* zowel *read* als *readFully* implementeert.

3.3.2. Directe toegang tot bestanden

Er bestaat behalve *DataInputStream* en *DataOutputStream* nog een andere klasse in *java.io* die de interfaces *DataInput* en *DataOutput* ondersteunt, en dan nog wel beide tegelijkertijd. Dit is de klasse *RandomAccessFile* die *directe toegang* tot bestanden mogelijk maakt.

Dankzij *RandomAccessFile* hoeven we een bestand niet meer telkens van voor naar achter te doorlopen, maar kunnen we gegevens inlezen of uitschrijven op verschillende plaatsen in dit bestand in een volgorde naar keuze. In zeker zin gedraagt het bestand zich als een tabel van bytes die zich rechtstreeks op de harde schijf bevinden — zij het dan met een toegangssnelheid die verschillende grootteordes trager is dan bij een gewone tabel in het geheugen.

Je gebruikt *seek* om een byte met een bepaald volgnummer te benaderen.

public void *seek* (**long** *pos*) **throws** *IOException*

Het volgnummer *pos* wordt steeds gerekend vanaf nul en geeft een aantal *bytes* weer. Heb je, zoals bij het voorbeeldbestand *priem.dat*, getallen in het bestand opgeslagen, dan moet je *pos* dus eerst vermenigvuldigen met het aantal bytes per getal (vier voor een **int**). Om dus gegevens in te lezen vanaf een bepaalde positie in een bestand, 'spring' je eerst naar deze positie met *seek* en lees je daarna de gegevens in met de *read*-methodes uit *DataInput*. Uitschrijven vanaf een bepaalde positie gebeurt op een gelijkaardige manier.

In het volgende voorbeeld gebruiken we een binaire zoekmethode om een getal op te zoeken in het bestand *priem.dat*. Het programma gaat na of de getallen die op de opdrachtlijn werden ingetikt al dan niet priemgetallen zijn.

```
public static void main (String[] args) throws IOException {
    RandomAccessFile file = new RandomAccessFile ("priem.dat", "r");
    int aantal = (int)file .length () / 4;
    for (String arg : args) {
        int getal = Integer.parseInt (arg);
        if (getal >=2 && getal < MAX) {
            // Binaire zoekmethode
            int b = 0, l = aantal;
            while (l - b > 1) {
                int m = (b + l) / 2;
                file .seek (4*m);
                if (getal < file .readInt())
                    l = m;
                else
                    b = m;
            }
            file .seek (4*b);
            if (getal == file .readInt())
                System.out.println (getal + " is priem");
            else
                System.out.println (getal + " is NIET priem");
        }
    }
    file .close ();
}
```

Merk op dat de constructor van *RandomAccessFile twee* stringparameters heeft. De eerste string is de bestandsnaam, net zoals bij *FileReader* en *FileWriter*, de tweede string bepaalt of je enkel kan lezen van dit bestand (`"r"`) of zowel lezen als schrijven (`"rw"`). De methode *length* vertelt je uit hoeveel bytes het bestand op dat moment bestaat.

Hoewel ze vele methodes ondersteunt die dezelfde naam dragen, is de klasse *RandomAccessFile geen* extensie van één van de streamklassen. Soms is dit vervelend: dit betekent onder andere dat je geen filterstromen kan koppelen aan een object van dit type — bijvoorbeeld om invoer en uitvoer te bufferen.

3.4. Andere klassen in *java.io*

3.4.1. Andere streams

Het pakket *java.io* ondersteunt nog meer interessante streamklassen voor in- en uitvoer waar we hier echter niet in detail kunnen op ingaan.

Zo kan je bijvoorbeeld met de klassen *PipedInputStream* en *PipedOutputStream* (of hun tegenhangers *PipedReader* en *PipedWriter*) communicatie binnen je programma verzorgen. Deze streams werken in paren: wat je op de ene schrijft, kan je uit de andere uitlezen. Deze klassen werken als een soort FIFO-wachtlijn en worden vaak gebruikt om gegevens uit te wisselen tussen verschillende draden (*threads*) van dezelfde toepassing.

Java ondersteunt ook enkele streamklassen voor in- en uitvoer van en naar geheugentabellen. De klasse *ByteArrayInputStream* kan je bijvoorbeeld gebruiken om gegevens te lezen uit een tabel van bytes alsof deze tabel een soort 'intern' bestand is. De klasse *ByteArrayOutputStream* biedt de omgekeerde functionaliteit: je kan nu *write*-opdrachten gebruiken om een tabel met bytes op te vullen.

Analoge klassen zijn *CharArrayReader* en *CharArrayWriter* waarmee je tabellen van karakters als streams kan behandelen, en *StringReader* en *StringWriter* voor strings (of stringbuffers).

Het pakket *java.util.zip* bevat een aantal filterstreams waarmee je andere streams kan comprimeren en decomprimeren. Met behulp van *GZipOutputStream* kan je bijvoorbeeld op een eenvoudige manier gegevens naar een gecomprimeerd bestand uitvoeren, en die met *GZipInputStream* terug inlezen. Er zijn ook klassen waarmee je *archieven* kan bewerken: files die tegelijkertijd *meerdere* gecomprimeerde bestanden bevatten. Het pakket *java.util.jar* bevat gelijkaardige filters voor *.jar*-bestanden.

3.4.2. De klasse *File*

De naam van de klasse *File* is een beetje verkeerd gekozen: *FileName* of *PathName* zou wellicht een betere keuze geweest zijn. Deze klasse heeft weinig met invoer en uitvoer te maken, verwar ze dus niet met *FileReader* of *FileWriter*.

De bedoeling van deze klasse is tweeërlei: enerzijds zorgt ze voor een behande-

ling van padnamen (*path names*) die onafhankelijk is van het besturingssysteem, en anderzijds voorziet ze een aantal bewerkingen op volledige bestanden, zoals wissen, verplaatsen en hernoemen.

De constructoren van *File* laten toe om een volledige padnaam van een bestand op te bouwen aan de hand van gegeven strings of door bestaande objecten van het type *File* uit te breiden met verdere onderdelen.

public *File* (*String pathname*);

public *File* (*String directory*, *String name*);

public *File* (*File directory*, *String name*);

De methode *toString* zet een *File*-object terug om naar een leesbare padnaam, volgens de conventies van het huidige besturingssysteem. Het volgende fragment

```
File dir = new File ("Windows", "Temp");
File naam = new File (dir, "test.dat");
System.out.println (naam);          // Roept toString op!
```

heeft dus deze uitvoer op een Windows-toestel:

```
Windows\Temp\test.dat
```

maar onder UNIX geeft dit

```
Windows/Temp/test.dat
```

Je hoeft dus niet te weten wat het scheidingsteken voor padnamen is op het besturingssysteem waarop je toepassing draait. In nood kan je het echter altijd terugvinden in de variabele *File.pathSeparatorChar*. De meeste constructoren in *java.io* die een bestandsnaam als parameter verwachten, laten in de plaats ook een *File*-object toe.

Merk op dat *File* de bestanden waarvan je de padnaam construeert niet gaat opzoeken. Je kan dus zonder problemen een padnaam opbouwen met daarin enkele onbestaande directories. De constructoren genereren geen uitzonderingen.

De volgende methodes interageren wel met het bestandssysteem:

```
// Maakt een directory aan met deze naam, t.t.z., een directory
// waarvan de naam in dit File-object is opgeslagen.
public boolean mkdir()

// Hernoemt het bestand of de directory met deze naam naar de
// nieuwe naam opgeslagen in dest.
public boolean renameTo (File dest)

// Probeert het bestand met deze naam te verwijderen
public boolean delete()
```

Deze methodes geven **true** terug als de gevraagde actie is gelukt en anders **false**. Merk op dat er ook hier geen *IOException* kan optreden.

Er zijn ook methodes waarmee je de inhoud van een directory kan opvragen. De methode *listFiles* geeft een tabel (array) van *File*-objecten terug die overeenkomen met de bestanden (en directories) die zich in de opgegeven directory bevinden, t.t.z., de directory waarvan de naam in het *File*-object is opgeslagen waarop de methode wordt toegepast. Je mag niet verwachten dat *listFiles* de bestanden in alfabetische volgorde teruggeeft.

De methode *isDirectory ()* kan je gebruiken om te weten of een *File*-object al dan niet een directory voorstelt.

Het pakket *java.io* is niet het enige dat in- en uitvoer verzorgt. Er is ook *java.net* voor invoer en uitvoer over een computernetwerk en *javax.comm* dat communicatie verzorgt via de seriële en parallelle poorten van de computer. Sinds versie 1.4 van Java, is er ook het 'nieuwe' invoer/uitvoer-pakket *java.nio*.

4. Modellen

In dit hoofdstuk geven we een eerste inleiding tot het zogenaamde *MVC-patroon*: een aantal bijzondere technieken die gebruikt worden bij het ontwerp van complexe componenten. Naast de componentklasse zelf en eventuele luisteraarklassen, speelt nu ook het zogenaamde *model* een belangrijke rol.

4.1. Model, view en controller

Bij het software-ontwerp van een grafische gebruikersinterface maakt men onderscheid tussen drie verschillende functionaliteiten: het *model*, de *controller* en de *view*.

Het *model* bevat de feitelijke informatie die grafisch zal worden afgebeeld: bijvoorbeeld de meetwaarden waarvan een grafiek wordt getoond of de woorden in keuzelijst. De *view* correspondeert met de grafische voorstelling van de gegevens uit het model: het staafdiagram met de meetwaarden of de keuzelijst zelf. De *controller* zorgt ervoor dat het model kan worden aangepast door interactie met de gebruiker: misschien kan je de staven in het diagram met de muis groter of kleiner maken of kan je elementen verwijderen uit de keuzelijst.

Het is niet ongebruikelijk dat hetzelfde model verschillende views toelaat: een staafdiagram of een taartdiagram voor dezelfde meetwaarden bijvoorbeeld. Uit bovenstaande voorbeelden zie je ook dat controllers niet altijd hoeven aanwezig te zijn. Soms is er helemaal geen gebruikersinteractie mogelijk met het model.

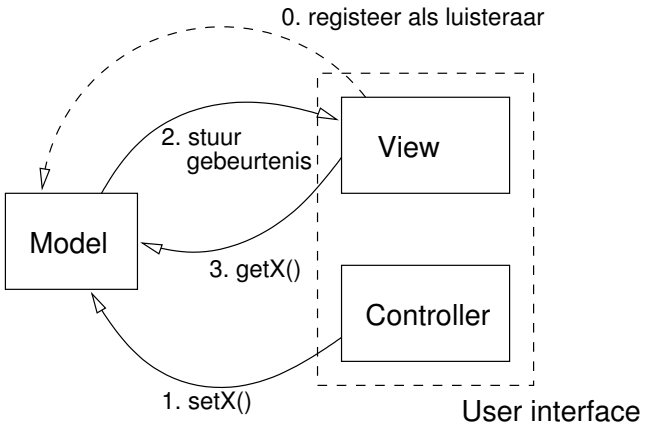
Bij de meeste componenten uit Swing wordt het model voorgesteld door één object en worden view en controller samen gegroepeerd in een ander object. Men spreekt in dit geval ook van het *model* en de *interface* (niet te verwarren met het begrip interface uit de programmeertaal zelf).

Het MVC-patroon heeft als bedoeling om model en view (interface) zoveel mogelijk van elkaar te scheiden. Het moet eenvoudig zijn om de interne voorstelling

van het model aan te passen zonder dat de view daarom opnieuw moet worden geprogrammeerd en je moet een view kunnen aanpassen zonder dat dit gevolgen heeft voor de klasse die het model voorstelt.

Men zorgt er zelfs voor dat nieuwe, bijkomende views aan een model kunnen worden gekoppeld zonder dat het model daarvan echt op de hoogte hoeft te zijn. Hiervoor gebruikt men een *luisteraar*mechanisme: elke view registreert zichzelf als luisteraar bij het model. We gebruiken hierbij de volgende algemene strategie:

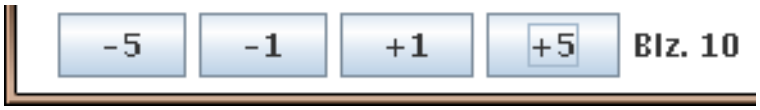
- Telkens wanneer de gegevens in het model veranderen, signaleert het model dit aan al zijn luisteraars. Vaak gebeurt dit met behulp van gebeurtenissen.
- Wanneer een view merkt dat het model is veranderd, vraagt hij aan het model de nieuwe gegevens op en past de corresponderende afbeelding op het scherm aan. Deze gegevens worden opgevraagd met behulp van (getter)methoden van het model.
- De controller gebruikt (setter)methoden van het model om aanpassingen aan te brengen. Het model kan trouwens op dezelfde manier ook rechtstreeks vanuit het programma worden gemanipuleerd.



Merk op dat gebruikerinteractie op die manier slechts onrechtstreeks een visueel effect heeft: eerst wordt het model aangepast en dit heeft op zijn beurt als gevolg dat de view verandert, omdat de view een luisteraar is van het model. Het is niet toegelaten dat een controller of een ander deel van het programma de view rechtstreeks aanpast.

4.1.1. Een voorbeeld: nieuwe paginaknoppen

Om dit allemaal te verduidelijken, bekijken we opnieuw het knoppenvoorbeeld uit hoofdstuk 2.



We bespreken een nieuw ontwerp dat de MVC-strategie toepast. Hierbij gebruiken we de volgende klassen:

- Een klasse *IntegerModel* waarvan één object zal dienst doen als model. De enige informatie die we nodig hebben om alle knoppen en labels te kunnen afbeelden, is het huidige bladzijdenummer, een geheel getal. Dit is dan ook het enige gegeven dat door een object van het type *IntegerModel* wordt bijgehouden en beheerd.
- Een klasse *BlzLabel* die zorgt voor een view van het model in de vorm van een label met daarop het overeenkomstige bladzijdenummer.
- De klasse *BlzKnop* waarvan we vier objecten zullen gebruiken. In de eerste plaats zijn dit controllers. Ze dienen namelijk om de gebruiker toe te laten de waarde van het model aan te passen. Tegelijkertijd doen ze ook dienst als minimale view: de waarde van het model bepaalt immers of de knoppen actief (*enabled*) zijn, of niet.
- De interface *ChangeListener* waaraan alle views voldoen. In dit voorbeeld gebruiken we een interface uit de Swing-bibliotheek. In andere gevallen zouden we hiervoor een eigen interface ontwerpen. Het model hoeft enkel te weten dat alle views van dit type zijn. Je zal de namen *BlzLabel* en *BlzKnop* dus nergens terugvinden in de definitie van *IntegerModel*.

4.1.2. Het model

De klasse *IntegerModel* is vrij eenvoudig van opzet. Het geheel getal dat door het model wordt beheerd, bevindt zich in een private variabele *waarde* die met een *getter* kan worden opgevraagd.

```
public int getWaarde () {
    return waarde;
}
```

Om het geheel getal aan te passen, gebruik je een corresponderende *setter*.

```
public void setWaarde (int waarde) {
    if (waarde != this.waarde) {
        this.waarde = waarde;
        fireStateChanged ();
    }
}
```

De methode *setWaarde* bevat wellicht iets meer Java-opdrachten dan je op het eerste zicht had verwacht. Het MVC-patroon schrijft immers voor dat een model elke verandering moet signaleren aan al zijn luisteraars. Dat is de taak van de methode *fireStateChanged* waarvan we later de implementatie zullen tonen. Merk op dat luisteraars slechts worden ingelicht wanneer de waarde werkelijk verandert — en dit is een niet onbelangrijk detail.

De overige methoden van de klasse dienen voor het beheer van de luisteraars van het model:

```
private EventListenerList listenerList = new EventListenerList();
    // lijst van geregistreerde luisteraars

public void addChangeListener (ChangeListener l) {
    listenerList .add (ChangeListener.class, l);
}

public void removeChangeListener (ChangeListener l) {
    listenerList .remove (ChangeListener.class, l);
}
```

De klasse *EventListenerList* uit Swing is speciaal ontworpen voor het bijhouden van de luisteraars van een model of een component. Je kan bovendien in één enkele lijst luisteraars opnemen van verschillende types. Zoals je hierboven ziet, gebruik je *add* om een luisteraar toe te voegen aan deze lijst en *remove* om hem er opnieuw uit te verwijderen. Beide methoden hebben twee parameters: behalve de luisteraar zelf wordt ook de klasse van de luisteraar in de lijst opgeslagen.

Eigenlijk moet je een object van het type *EventListenerList* interpreteren als een lijst die afwisselend een klasse en een object van die klasse bevat. Je overloopt de lijst dus in stappen van twee. Je ziet hiervan een voorbeeld in de implementatie van de methode *fireStateChanged*, de methode die aan alle luisteraars van het juiste type laat weten dat er iets aan het model is veranderd.

```
private final ChangeEvent changeEvent = new ChangeEvent (this);
    // uniek gebeurtenisobject met dit model als bron

private void fireStateChanged () {
    Object[] listeners = listenerList.getListenerList();
    for (int i=listeners.length−2; i >= 0; i−=2) {
        if (listeners[i] == ChangeListener.class)
            ((ChangeListener)listeners[i+1]).stateChanged (changeEvent);
    }
}
```

Merk op dat het model informatie naar zijn luisteraars verstuurd in de vorm van een gebeurtenis, hier van het type *ChangeEvent*. De interface *ChangeListener* bevat één methode, namelijk *stateChanged* die een *change*-gebeurtenis als parameter neemt.

De code voor de methoden die de luisteraars beheren, hebben we bijna letterlijk overgenomen uit de documentatie van de klasse *EventListenerList*. Je kan dezelfde code bij de meeste modellen hergebruiken. Het is zelfs een goed idee om deze code te bundelen in een klasse *Model* waarvan je dan alle andere modellen kan afleiden.

4.1.3. Het bladzijdelabel

Ook de klasse *BlzLabel* is zeer klein. Dit is een extensie van *JLabel* die tegelijkertijd als view van *IntegerModel* dient. Daarom moet *BlzLabel* aan de interface *ChangeListener* voldoen en dus de methode *stateChanged* implementeren.

```
public class BlzLabel extends JLabel implements ChangeListener {
    ...
    public void stateChanged (ChangeEvent e) {
        setText ("Blz. " + model.getWaarde ());
    }
}
```

Volgens het MVC-patroon wordt de methode *stateChanged* opgeroepen telkens wanneer de waarde van het model verandert, m.a.w., telkens wanneer er een nieuwe bladzijdenummer wordt geselecteerd (door de gebruiker of vanuit het programma). De view vraagt de nieuwe waarde op aan het model met behulp van *getWaarde* en gebruikt die om zijn eigen grafische voorstelling aan te passen.

Opdat dit zou werken, is het nodig dat de view het model kan terugvinden waarmee het communiceert. In dit geval zouden we dit kunnen doen aan de hand van de methode *getSource* van het *ChangeEvent*, maar het is gebruikelijk om het model (dat tijdens de levensduur van de view wellicht toch niet verandert) op te slaan als attribuut van het object.

Vaak wordt het model meegegeven als parameter aan de constructor voor de view, zoals ook bij ons:

```
public class BlzLabel extends JLabel implements ChangeListener {
    private IntegerModel model;

    public BlzLabel (IntegerModel model) {
        super ("Blz. " + model.getWaarde());
        this.model = model;
        model.addChangeListener (this);
    }
}
```

Merk op dat de view wanneer hij wordt gecreëerd, zichzelf als luisteraar bij het model registreert.

4.1.4. De bladzijdeknop

De bladzijdeknop dient tegelijkertijd als view en als controller. Daarom implementeert de klasse *BlzKnop* zowel *ChangeListener*, om als view op veranderingen in het model te kunnen reageren, en *ActionListener*, om interactie met de gebruiker mogelijk te maken.

Hieronder schetsen we die gedeelten van de klasse die van *BlzKnop* een view maken.

```
public class BlzKnop extends JButton
                implements ChangeListener, ActionListener {
```

```
    private IntegerModel model;

    private int increment;          // zie §2.1.2

    public BlzKnop (IntegerModel model, int increment, String opschrift) {
        super (opschrift);
        this.model = model;
        this.increment = increment;
        model.addChangeListener (this);
        ...
    }

    public void stateChanged (ChangeEvent e) {
        setEnabled (model.getWaarde() <= 100 - increment &&
                    model.getWaarde() >= 0 - increment);
    }
}
```

Onderstaande code komt overeen met de controller-functionaliteit.

```
public class BlzKnop extends JButton
                implements ChangeListener, ActionListener {
    public BlzKnop (IntegerModel model, int increment, String opschrift) {
        ...
        addActionListener (this);
    }

    public void actionPerformed (ActionEvent e) {
        model.setWaarde (model.getWaarde() + increment);
    }
}
```

Merk op dat *actionPerformed* zelf niet zorgt voor het al dan niet activeren van de knop. Dit gebeurt onrechtstreeks in *stateChanged* wanneer de knop door het model van de verandering op de hoogte is gebracht.

4.1.5. Het hoofdprogramma

Tot slot illustreert het volgende fragment hoe views en model worden geïnitialiseerd en met elkaar verbonden.

```
IntegerModel model = new IntegerModel ();
panel.add (new BlzKnop (model, −5, "−5"));
panel.add (new BlzKnop (model, −1, "−1"));
panel.add (new BlzKnop (model, 1, "+1"));
panel.add (new BlzKnop (model, 5, "+5"));
panel.add (new BlzLabel (model));
model.setWaarde (50);
 ...
```

We maken eerst het model aan en daarna de verschillende views. Bij elke view geven we een verwijzing door naar hetzelfde model. De laatste opdracht stelt een bladzijdenummer in bij het model, en dit zal meteen de nodige veranderingen teweeg brengen bij de verschillende views. We hoeven de views dus niet afzonderlijk te initialiseren.

4.2. Acties

Vaak kan je aan een toepassing dezelfde opdracht doorgeven op verschillende manieren, bijvoorbeeld door het selecteren van een bepaald menu-element en door het drukken op een knop in een werkbalk.

Niet alleen hebben beide knoppen hetzelfde opschrift en dezelfde afbeelding (hoewel ze toch afzonderlijk moeten worden aangemaakt), bovendien wens je dat beide steeds tegelijkertijd geactiveerd (*enabled*) of gedesactiveerd (*disabled*) zijn — en het is niet zo eenvoudig om dit voor elkaar te krijgen.

4.2.1. De interface *Action*

Om dit te vereenvoudigen, voorziet Swing de interface *Action*. Een object van dit type, de zogenaamde *actie*, brengt verschillende functionaliteiten samen:

- Een actie bevat de noodzakelijke gegevens, zoals opschrift en afbeelding, om zich als knop of menu-element te kunnen gedragen.

  Om een actie naar een knop om te zetten gebruik je een (knop)constructor met deze actie als parameter. Er bestaan dergelijke constructoren voor *JButton*, *JCheckBox*, *JRadioButton* en *JToggleButton* en ook voor *JMenuItem*, *JCheckBoxMenuItem*, *JRadioButtonMenuItem* en *JMenu*. (Er volgt verder een voorbeeld.)

- De interface *Action* is een extensie van *ActionListener*. Een actie geldt meteen als luisteraar voor alle (menu)knoppen die ermee zijn aangemaakt. Acties zijn dus in zekere zin de *controllers* van Swing.

- Een actie houdt zelf bij of de overeenkomstige knoppen geactiveerd zijn of niet. Je kan alle knoppen die met dezelfde actie overeenkomen tegelijkertijd (des)activeren door de methode *setEnabled* op te roepen van de actie zelf.

- Je kan een actie rechtstreeks toevoegen aan een menu of aan een werkbalk en Swing zal ze dan zelf omzetten naar een menuknop of een gewone knop met de gepaste eigenschappen. (In een werkbalk zal bijvoorbeeld enkel de afbeelding van een actie te zien zijn en niet het opschrift.)

Voor elke actie die je aanmaakt, moet je zelf een nieuwe klasse definiëren. Meestal schrijft men hiervoor een extensie van *AbstractAction*, want dan hoef je enkel een implementatie van *actionPerformed* te voorzien en eventueel een constructor.

De volgende actieklasse heeft dezelfde functionaliteit als de *BlzKnop* uit paragraaf 4.1.1. Behalve het opschrift neemt de constructor nu ook een afbeelding als argument.

```
public class BlzAction extends AbstractAction
                                    implements ChangeListener {
    private IntegerModel model;

    private int increment;

    public BlzAction
            (IntegerModel model, int increment, String opschrift, Icon icon) {
        super (opschrift, icon);
        this.model = model;
        this.increment = increment;
        model.addChangeListener (this);
    }
```

```
    public void actionPerformed (ActionEvent e) {
        model.setWaarde (model.getWaarde () + increment);
    }

    public void stateChanged (ChangeEvent e) {
        setEnabled (model.getWaarde () <= 100 − increment &&
                    model.getWaarde () >= 0 − increment);
    }
}
```

(Het feit dat deze klasse ook *ChangeListener* is niet eigen aan een actie, maar is het gevolg van het feit dat we de actie ook willen automatisch activeren of desactiveren naargelang de waarde van het model.)

We maken vier instanties aan van deze klasse en stoppen die in de tabel *actions* :

```
Action[] actions = new Action[4];
actions[0] = new BlzAction (model, −5, "5 terug", new ImageIcon (...));
 ...
actions[3] = new BlzAction (model, 5, "5 vooruit",
                                              new ImageIcon (...));
```

De figuur op de volgende bladzijde illustreert hoe elk van deze vier acties drie keer wordt gebruikt : om een menu-element aan te maken, om een knop aan te maken in het paneel onderaan het venster, en om een knop aan te maken in de werkbalk aan de rechterkant.

Het menu en de werkbalk worden dan als volgt aangemaakt:

```
JMenu menu = new JMenu ("Bladzijde");
menu.setMnemonic (KeyEvent.VK_B);
for (Action action : actions)
    menu.add (action);

JToolBar toolBar = new JToolBar ();
for (Action action : actions)
    toolBar.add (action);
```

en de knoppen in het paneel onderaan op de volgende manier:

```
JPanel panel = new JPanel ();
for (Action action : actions)
    panel.add (new JButton (action));
```

Je kan aan een actie ook een sneltoets of een afkortingstoets hechten die dan automatisch wordt overgenomen door de knop of het menu-element. Meer informatie vind je in de elektronische documentatie.

4.2.2. Acties en toetsenbordinvoer

Acties worden door Swing ook voor andere dingen gebruikt dan (menu-)knoppen. Een belangrijke toepassing van acties zijn de zogenaamde *action maps* (en de hieraan gerelateerde *input maps*) waarmee toetsaanslagen rechtstreeks kunnen verbonden worden met bepaalde handelingen.

Stel bijvoorbeeld dat je in je toepassing een bepaalde kopie-actie wil oproepen telkens wanneer de gebruiker CTRL-C tikt. Je gaat dan als volgt te werk : schrijf een klasse *CopyAction* die *Action* implementeert (wellicht door *AbstractAction* uit te breiden). Vul de methode *actionPerformed* van deze nieuwe actie in met de code die nodig is om de kopie-actie uit te voeren. (In dit geval hoef je geen hoofding, mnemonic en dergelijke aan de actie toe te kennen.) Kies een beschrijvende naam

voor de actie (bijvoorbeeld `copy`) en registreer de actie onder deze naam in de *action map* van de component die op je invoer moet reageren (zie verder). Gebruik daarna de *input map* van dezelfde component om de naam `copy` te verbinden met de correcte toetsaanslag (van het type *KeyStroke*).

Verwar het begrip 'naam voor de actie' niet met de naam van de klasse die de actie implementeert. Het is immers goed mogelijk dat er verschillende actie-objecten bestaan (met verschillende namen, bijvoorbeeld `left`, `right`, `up` en `down`) die tot dezelfde klasse behoren.

Het indrukken van een toets wordt in verschillende stadia afgewerkt : eerst zoekt Swing of de toetsaanslag in een input map voorkomt en haalt hij de corresponderende actienaam op, daarna zoekt hij deze actienaam op in de action map van de component en haalt hij het corresponderende *Action*-object op. Tot slot voert hij de *actionPerformed* uit van dit *Action*-object.

Een action map is van het type *ActionMap*. Met *getActionMap* haal je ze op bij een component. Je registreert je acties met behulp van de methode *put*:

```
ActionMap actionMap = panel.getActionMap ();
actionMap.put ("copy", new CopyAction());
```

Input maps zijn analoog, alleen geef je nu bij *put* een *KeyStroke* en een *String* mee als argumenten, in plaats van een *String* en een *Action*:

```
InputMap inputMap = panel.getInputMap ();
inputMap.put (KeyStroke.getKeyStroke("ctrl C"), "copy");
```

Om heel precies te zijn, bezit elke component eigenlijk drie verschillende input maps: één die gebruikt wordt wanneer de component de focus heeft (dit is de default), één voor wanneer één van de kinderen de focus heeft en één voor wanneer een andere component in hetzelfde venster de focus heeft. Deze laatste input map wordt door Swing intern gebruikt voor de implementatie van sneltoetsen. Voor meer informatie verwijzen we naar de elektronische documentatie.

In plaats van de bestaande input map (of action map) van de component te gebruiken, heb je ook de mogelijkheid om zelf een input (of action) map aan te maken en bij de component te registreren. Elk input (of action) map heeft een andere map als *ouder*. Wanneer een toets (of actienaam) niet in de map zelf wordt gevonden, wordt automatisch verder gezocht bij de ouder. Wanneer je dus zelf een map aanmaakt, kan je best de oorspronkelijke map van de component als ouder gebruiken, zodat de bestaande functionaliteit niet verloren gaat.

```
InputMap im = new InputMap ();
im.setParent (component.getInputMap());
im.put (KeyStroke.getKeyStroke ("UP"),  "up");
im.put (KeyStroke.getKeyStroke ("DOWN"), "down");
im.put (KeyStroke.getKeyStroke ("LEFT"), "left");
im.put (KeyStroke.getKeyStroke ("RIGHT"), "right");
component.setInputMap (JComponent.WHEN_FOCUSED, im);
```

De constante *WHEN_FOCUSED* duidt aan dat het hier om de input map gaat die gebruikt wordt wanneer de component de focus heeft.

4.3. De klasse *JSpinner*

Elke component uit de Swing-bibliotheek maakt gebruik van het model/view/controller principe, ook al is dit soms goed verborgen (zoals bij knoppen bijvoorbeeld). Als voorbeeld van een meer complexe component waarbij je het gebruik van een modelklasse niet kan vermijden, bespreken we hier de klasse *JSpinner*. Andere klassen met gelijkaardige vereisten komen aan bod in hoofdstuk 9.

Een object van het type *JSpinner* stelt een *spinner*-component voor, zoals in onderstaande figuur. Spinners laten de gebruikers toe om door een lijst van keuzes te stappen met behulp van kleine pijltjesknoppen (of met de pijltjestoetsen). Vaak is het ook mogelijk een keuze rechtstreeks in te tikken.

(De tekst 'Graad:' is een *JLabel* en hoort niet bij de spinner.)

4.3.1. Overzicht

Wanneer je een nieuwe spinner aanmaakt, geef je het model mee als parameter voor de constructor.

```
public JSpinner (SpinnerModel model);
```

Het model moet een object zijn van het type *SpinnerModel* (een interface).

Voor de meeste toepassingen kan je als modelklasse één van de standaardimplementaties van deze interface gebruiken:

- *S*pinnerNumberModel: hiermee kan de gebruiker kiezen uit een vooropgegeven getallenbereik, bijvoorbeeld alle getallen van 100 tot en met 2000, in stappen van 50.
- *S*pinnerDateModel: dit model doorloopt een reeks van opeenvolgende data (of tijdstippen).
- *S*pinnerListModel: het model doorloopt een vaste reeks van objecten, die werden opgegeven als een tabel (array) of een lijst (*java.util.List*).

Met behulp van deze laatste klasse kunnen we nu gemakkelijk een spinner construeren die door opeenvolgende *diplomagraden* loopt:

```
String[] graden
   = { "Niet geslaagd", "Voldoening", "Onderscheiding",
       "Grote onderscheiding", "Grootste onderscheiding" };
SpinnerModel model = new SpinnerListModel (graden);
JSpinner spinner = new JSpinner (model);
```

In enkele gevallen is het echter nuttig om zelf een modelklasse te ontwerpen. Deze klasse moet dan een implementatie zijn van de interface *SpinnerModel*. Deze interface bezit 6 verschillende methoden die we hieronder kort overlopen.

Een spinnermodel verwacht dat zijn views van het type *ChangeListener* zijn. Daarom moet je dergelijke luisteraars bij het model kunnen registreren (en hun registratie terug ongedaan kunnen maken).

```
public void addChangeListener (ChangeListener l);

public void removeChangeListener (ChangeListener l);
```

Om deze twee methoden te implementeren, kan je je net zoals bij *IntegerModel* (§4.1.2) baseren op een *EventListenerList*, of je kan je klasse een uitbreiding maken van de abstracte klasse *AbstractSpinnerModel* die dit voor jou doet. Deze abstracte klasse geeft je ook meteen een methode *fireStateChanged* cadeau.

Om te bepalen wat er in het spinnerveld moet worden afgebeeld, gebruikt *JSpinner* de volgende methode van *SpinnerModel*:

public *Object getValue* ();

Merk op dat de teruggegeven waarde van elk mogelijk (referentie)type mag zijn. Om dit object af te beelden, roept *JSpinner* zijn *toString*-methode op.

Aan een spinnermodel moet je ook kunnen opvragen wat het vorige object is in de reeks, en wat het volgende is. (Dit gebeurt wanneer men op één van de pijltjesknoppen van de spinner drukt.)

public *Object getNextValue* ();

public *Object getPreviousValue*();

Deze methoden geven **null** terug wanneer de grenzen van de reeks zijn bereikt.

Tot slot moet je ook een nieuwe spinnerwaarde kunnen instellen in het model:

public void *setValue* (*Object value*);

en zoals bij elk model moet je dan ook alle geregistreerde luisteraars op de hoogte stellen wanneer deze waarde is veranderd. (Wellicht met behulp van *fireStateChanged*.)

De spinnercomponent zal deze methode bijvoorbeeld oproepen wanneer er op één van de pijltjes toetsen werd gedrukt:

model.setValue (*model.getNextValue* ());

Het is ook mogelijk een spinner zo in te stellen dat de nieuwe spinnerwaarde rechtstreeks door de gebruiker kan worden ingetikt, en ook in dat geval zal deze nieuwe waarde aan het model worden doorgegeven met behulp van *setValue*. Dit betekent dat jouw implementatie van *SpinnerModel* ook 'onmogelijke' waarden kan binnenkrijgen. De documentatie stelt dat je in dat geval een *IllegalArgumentException* mag opgooien.

4.3.2. Een spinnermodel voor diplomagraden

Ter illustratie van wat we hierboven hebben verteld, zullen we een nieuwe klasse *GraadSpinnerModel* implementeren voor een spinner met diplomagraden. De graden zelf stellen we niet voor als strings maar als objecten van een eenvoudige klasse *Graad*.

```
public class Graad {

    public String naam;       // volledige benaming

    public String afkorting;  // bijv. 'go' voor 'grote onderscheiding'

    public Color bg;          // voorkeursachtergrondkleur

    public Color fg;          // voorkeursvoorgrondkleur

    public Icon icon;         // bijhorende afbeelding

    public int grens;         // bijv. 15 voor 'grote onderscheiding'

     ...
}
```

Zoals je ziet, houden we voor elke graad verschillende soorten informatie bij. (Kleuren en afbeeldingen zullen ons straks nog van pas komen — zie §4.3.3.)

Daarnaast bezit *Graad* nog enkele methoden die toelaten om voor een bepaalde graad de volgende of vorige graad in het rijtje te bekomen

```
    public Graad next ();

    public Graad previous ();
```

en een methode die de graad teruggeeft die overeenkomt met een opgegeven puntental.

```
    public static Graad getGraad (int punten);
```

Met behulp van deze klasse is het niet zeer moeilijk om een aangepast spinnermodel te ontwerpen. We houden de 'huidige' graad bij als een privaat veld *graad*

van het model.

```
public class GraadSpinnerModel extends AbstractSpinnerModel {

    private Graad graad = Graad.getGraad (10);

    public Graad getValue () {
        return this.graad;
    }

    public Graad getNextValue () {
        return graad.next ();
    }

    public Graad getPreviousValue () {
        return graad.previous ();
    }
     ...
}
```

(Merk op dat de drie methoden *Graad* als type hebben, en niet *Object* zoals de interface *SpinnerModel* eigenlijk verwacht. Omdat *Graad* een subtype is van *Object* laat Java dit toe — althans vanaf versie 5.0.)

Wanneer de graad wordt aangepast, mogen we niet vergeten dit aan alle luisteraars te melden, maar dan enkel wanneer de graad wel degelijk verandert.

```
    public void setValue (Object value) {
        if (value instanceof Graad) {
            if (value != graad) {
                this.graad = (Graad)value;
                fireStateChanged ();
            }
        }
        else
            throw new IllegalArgumentException ();
    }
```

Merk op dat we een gepaste uitzondering opgooien wanneer een 'onmogelijke' waarde wordt aangeboden.

Er rest ons nu nog enkel een spinner aan te maken die dit nieuwe model toepast:

JSpinner spinner = **new** *JSpinner* (**new** *GraadSpinnerModel* ());

4.3.3. Een aangepaste spinner-editor

Tot slot illustreren we kort hoe je het uitzicht van de spinner naar wens kan omvormen door een zogenaamde spinner-*editor* als view bij de spinner te installeren. We zullen ervoor zorgen dat elke graad in een ander kleur wordt afgebeeld, telkens met een aangepast ikoontje.

De spinner-editor is het gedeelte van de spinnercomponent dat zich links van de pijltjesknoppen bevindt. In principe kan je elke *JComponent* als spinner-editor gebruiken, maar in de praktijk gebruikt men meestal extensies van *JLabel*, *JTextField* of *JFormattedTextField*. (In de laatste twee gevallen kan je de gebruiker eventueel toelaten de inhoud van het veld te wijzigen — voor meer informatie verwijzen we naar de elektronische documentatie.)

Per default kiest *JSpinner* een editor die afhangt van het spinnermodel dat wordt gebruikt. Er zijn afzonderlijke editoren voor elk van de drie standaard spinnermodelklassen en een 'default'-editor voor alle andere gevallen. De default-editor gebruikt een niet-editeerbaar geformateerd tekstveld.

Voor dit voorbeeld hebben we ervoor gekozen een editor te schrijven als extensie van *JLabel*. De klasse implementeert ook de interface *ChangeListener* aangezien de editor als view bij het spinnermodel moet worden geregistreerd.

```
public class Editor extends JLabel implements ChangeListener {

    private GraadSpinnerModel model;

    public Editor (GraadSpinnerModel model) {
        this.model = model;
```

```
        model.addChangeListener (this);
        ...
    }

    public void stateChanged (ChangeEvent e) {
        Graad graad = model.getValue ();
        setText  (graad.naam);
        setIcon  (graad.icon);
        setBackground (graad.bg);
        setForeground (graad.fg);
    }
}
```

Merk op dat we tekst, afbeelding, voorgrond- en achtergrondkleuren van het label aanpassen telkens wanneer het model een nieuwe waarde krijgt.

De initialisatie van de editor vraagt nog wat extra werk: we moeten het label ondoorzichtig maken (anders krijg je de achtergrond niet te zien), we geven het label nieuwe voorkeursafmetingen (zie §6.1) en we configureren het label alvast volgens de huidige waarde van het model.

```
    public Editor (GraadSpinnerModel model) {
        ...
        setOpaque (true);
        setPreferredSize (new Dimension (200, 20));
        stateChanged (null);
    }
```

5. Grafische bewerkingen

Soms is het duidelijker om informatie op een grafische manier af te beelden dan enkel maar met tekst — denk maar aan grafieken, taartdiagrammen, schema's, enz. In dit hoofdstuk bespreken we enkele klassen en methoden die je hierbij kunnen van dienst zijn. We geven ook meer uitleg bij het interactief gebruik van de muis.

Merk op dat je je in vele gevallen kan redden zonder deze grafische bewerkingen. Je kan afbeeldingen gebruiken op labels of op knoppen, of zelfs tussen lopende tekst in een tekstcomponent. Er bestaan ook standaardvoorzieningen om rond componenten kaders aan te brengen van allerlei aard.

5.1. Tekenen met Swing

Als voorbeeld ontwerpen we een toepassing die een rooster afbeeldt en die toelaat om in dit rooster een bepaald veld aan te duiden met behulp van het toetsenbord.

Als model van deze toepassing gebruiken we een object van de klasse *RijKolom-Model*. Dit is een zelfgeschreven model dat een rij- en kolomnummer bijhoudt en toelaat deze aan te passen, binnen bepaalde grenzen. De klasse heeft een eenvoudige interface:

```
public int getRij ();
public int getKolom ();
// Geef het huidige rij- of kolomnummer terug zoals dit door het model
// wordt bijgehouden

public void setPosition (int rij, int kolom);
// Stel een nieuw rij- en kolomnummer in (en breng hiervan alle
// luisteraars op de hoogte)
```

```
public void setPositionRelative (int drij, int dkolom);
// Verplaats rij- en kolomnummer volgens de opgegeven relatieve
// verschillen

public void addChangeListener (ChangeListener l);
// Registreer een luisteraar bij het model
...
```

De controllers zijn objecten van het type *MoveAction*. Dit zijn eenvoudige acties waarmee rij- en kolomnummer van het model kunnen worden aangepast volgens een voorafgegeven relatieve verplaatsing:

```
public class MoveAction extends AbstractAction {
    ...
    public MoveAction (RijKolomModel model, int drij, int dkolom) {
        this.model = model;
        this.drij  = drij;
        this.dkolom = dkolom;
    }

    public void actionPerformed (ActionEvent e) {
        model.setPositionRelative (drij, dkolom);
    }
}
```

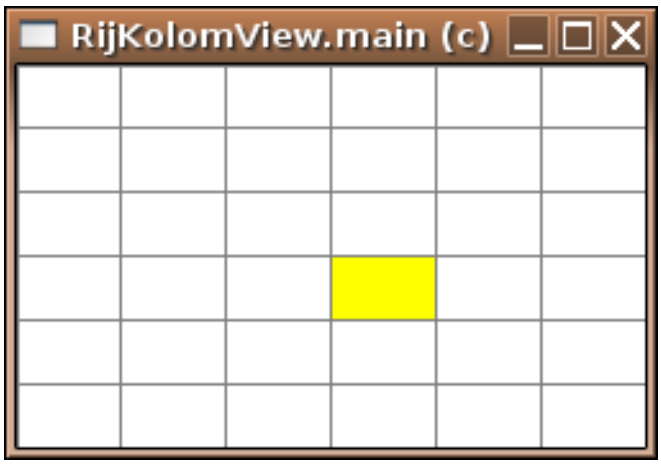
We hechten de juiste acties aan de juiste toetsen met behulp van een action map

```
ActionMap am = ...;
am.put ("up",    new MoveAction(model, −1, 0));
am.put ("down", new MoveAction(model, 1, 0));
am.put ("left", new MoveAction(model, 0, −1));
am.put ("right", new MoveAction(model, 0, 1));
```

die we dan verbinden met een input map, zoals uitgelegd in paragraaf 4.2.2.

5.1.1. Basistechnieken

Bekijken we nu de klasse *RijKolomView*, een view voor het model die een raster afbeeldt waarin het geselecteerde veld een andere kleur heeft, zoals afgebeeld in onderstaande figuur.



De dimensies van het paneel waarop het rooster wordt getekend, bepalen uiteindelijk de afmetingen van de afzonderlijke velden. Het totaal aantal rijen en kolommen kan je opvragen aan het model.

In tegenstelling tot wat je misschien verwacht, onthoudt Swing niet automatisch wat je op een paneel of een andere component hebt getekend. Wanneer tijdens de loop van het programma een stuk van je tekening door een ander venster wordt bedekt en daarna opnieuw vrijkomt, zal Swing de tekening dan ook niet zonder hulp van de programmeur kunnen reconstrueren. Het gebruik van grafische bewerkingen vraagt daarom een bijzondere programmeertechniek.

De inhoud van een component (een knop, een label, ...) wordt op het scherm getekend door de methode *paintComponent* van deze component. Wanneer we Swing iets anders wensen te laten tekenen dan gebruikelijk, moeten we zelf een nieuwe component ontwerpen als uitbreiding van een bestaande klasse en een nieuwe definitie geven van deze methode.

In ons voorbeeld creëren we de nieuwe klasse *RijKolomView* als extensie van *JPanel* — de klasse die hiervoor bij voorkeur voor gewone grafische toepassingen wordt gebruikt.

```
public class RijKolomView extends JPanel implements ChangeListener {
    ...

    protected void paintComponent (Graphics g) {
        super.paintComponent (g); // Niet vergeten !
        ...
    }
}
```

Wanneer je grafische uitvoer wenst te tekenen bovenop een vaste achtergrondfiguur, kan je in plaats van *JPanel* ook *JLabel* uitbreiden. De vaste achtergrond installeer je dan als *Icon* van het label. Occasioneel kan het ook nuttig zijn om grafische uitbreidingen te definiëren van andere componentklassen zoals *JButton* of *JTextArea*.

Swing zal de methode *paintComponent* telkens automatisch oproepen wanneer (een gedeelte van) het paneel door het programma moet worden hertekend. Deze methode moet alle opdrachten bevatten om het volledige paneel te hertekenen — behalve de achtergrond die de bovenklasse *JPanel* voor haar rekening neemt. Opdat *paintComponent* zijn taak naar behoren zou kunnen uitvoeren, moet je dus in je programma soms heel wat informatie opslaan over wat er allemaal op het paneel te zien is. Meestal doe je dit in het model dat bij de view hoort.

5.1.2. De klasse *Graphics*

De methode *paintComponent* krijgt als argument de zogenaamde *grafische context* mee, een object van het type *Graphics* dat je nodig hebt om grafische bewerkingen te kunnen uitvoeren.

Niet alleen biedt deze klasse verschillende methoden waarmee je lijnen, rechthoeken en cirkels kan tekenen, de grafische context houdt ook allerhande informatie bij, zoals bijvoorbeeld welke kleur er moet worden gebruikt bij de volgende grafische bewerking, of met welk lettertype de volgende tekst moet worden gezet.

We illustreren dit met behulp van de definitie van *paintComponent* uit de voorbeeldklasse *RijKolomView*.

```
protected void paintComponent (Graphics g) {

    super.paintComponent (g);
    int w = getWidth ();
    int h = getHeight ();
    int aantalRijen = model.getAantalRijen ();
    int aantalKolommen = model.getAantalKolommen ();

    // teken roosterlijnen
    g.setColor (Color.GRAY);
    for (int i = 1; i < aantalRijen; i++)
        g.drawLine (0, i * h / aantalRijen, w, i * h / aantalRijen);
```

```
        for (int i = 1; i < aantalKolommen; i++)
            g.drawLine (i * w / aantalKolommen,0, i * w / aantalKolommen,h);

        // vul het geselecteerde veld in
        g.setColor (Color.YELLOW);
        int rij = model.getRij ();
        int kolom = model.getKolom ();
        int aantalRijen = model.getAantalRijen ();
        int aantalKolommen = model.getAantalKolommen ();

        g.fillRect (...);  // zie §5.1.3
    }
```

De methoden *getWidth* en *getHeight* van *JPanel* geven de breedte en hoogte van het paneel terug, geteld in pixels. De methode *drawLine* tekent een lijn en *fillRect* tekent een opgevulde rechthoek.

De methode *setColor* stelt de kleur in die bij de volgende tekenopdracht(en) moet worden gebruikt. Kleuren zijn objecten van het type *Color*. Deze klasse bevat een aantal voorgedefinieerde kleuren (zoals *Color.RED*, *Color.BLACK* en *Color.WHITE* — je vindt de volledige lijst terug in de elektronische documentatie), maar je kan zelf ook nieuwe kleurobjecten aanmaken aan de hand van hun RGB-waarden:

```
Color pink = new Color (255, 128, 128);   // gehele waarden van 0–255
Color cyan = new Color (0.0f, 1.0f, 1.0f); // reële waarden tussen 0 en 1
```

(De reële parameters zijn van het type **float**, vandaar de 'f' in de getallen.)

Een vierde parameter (de transparantie) geeft aan hoe doorzichtig de kleur is. 0 betekent volledig doorzichtig, 255 (of 1.0) betekent volledig ondoorzichtig.

```
Color reddish = new Color (255, 0, 0, 128);
Color misty   = new Color (1.0f, 1.0f, 1.0f, 0.25f);
```

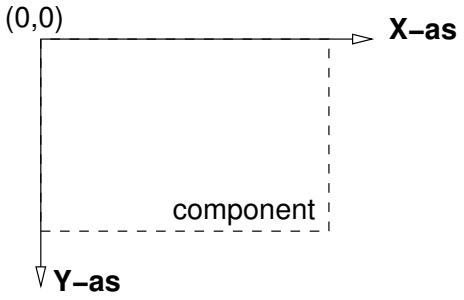
5.1.3. Elementaire tekenopdrachten

Met de opdracht *drawLine* uit de klasse *Graphics* trek je een lijnstuk tussen twee gegeven punten.

public void *drawLine* (**int** *x1*, **int** *y1*, **int** *x2*, **int** *y2*);

De parameters *x1* en *y1* zijn de X- en Y-coördinaten van het ene hoekpunt en *x2* en *y2* van het andere hoekpunt. Zoals bij alle grafische bewerkingen, wordt de lijn getrokken in de kleur die laatst werd ingesteld met *setColor*.

Normaal ligt de oorsprong van het coördinatenstelsel linksboven in de component. De X-as loopt horizontaal van links naar rechts en de Y-as loopt van boven naar onder (dus ondersteboven, wanneer je de wiskundige conventies gewoon bent). Lengte-eenheden worden uitgedrukt in schermpixels.



Je kan de oorsprong van het coördinatenstelsel verschuiven naar een willekeurig punt met behulp van de methode *translate* uit de klasse *Graphics*:

g.translate (*nieuweX*, *nieuweY*);

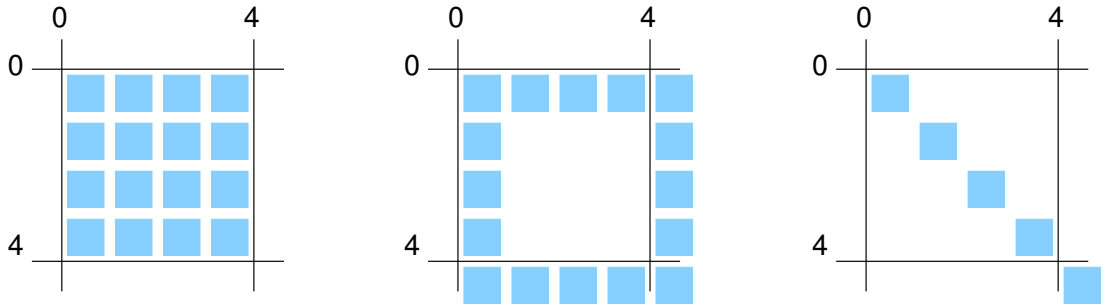
Herschalen of roteren van het assenstelsel kan niet. Hiervoor zal je zelf de nodige wiskundige formules moeten toepassen.

Er zijn twee methoden waarmee je een rechthoek kan tekenen. Bij *drawRect* tekent Swing de omtrek van een rechthoek, bij *fillRect* wordt het volledige oppervlak van de rechthoek ingevuld.

public void *drawRect*(**int** *x*, **int** *y*, **int** *w*, **int** *h*); // omtrek
public void *fillRect*(**int** *x*, **int** *y*, **int** *w*, **int** *h*); // oppervlak

In dit geval zijn *x* en *y* de coördinaten van de linkerbovenhoek van de rechthoek en *w* en *h* de breedte en hoogte van de rechthoek. Met andere woorden, het hoekpunt rechtsonder heeft als coördinaten (*x*+*w*, *y*+*h*). De zijden van de rechthoek zijn steeds evenwijdig met de coördinaatassen.

Coördinaten geef je in een programma steeds op als gehele getallen (type **int**). Coördinaten stellen eigenlijk oneindig kleine punten voor die *tussen* de schermpixels gelegen zijn. Onderstaande figuren schetsen het verband tussen de coördinaten en de pixels die effectief worden gekleurd.



Ze corresponderen met de volgende opdrachten:

```
g.fillRect   (0, 0, 4, 4);
g.drawRect   (0, 0, 4, 4);
g.drawLine   (0, 0, 4, 4);
```

Merk op dat er bij het opvullen van een rechthoek in de breedte en in de hoogte één pixel minder wordt getekend dan bij het tekenen van de omtrek.

Met deze informatie begrijp je nu wellicht beter waarom het gele veld op een *RijKolomView* op de volgende manier wordt geproduceerd:

```
g.fillRect
    (kolom * w / aantalKolommen + 1,
     rij  * h / aantalRijen + 1,
     (kolom + 1) * w / aantalKolommen - kolom * w / aantalKolommen - 1,
     ( rij  + 1) * h / aantalRijen - rij * h / aantalRijen - 1 );
```

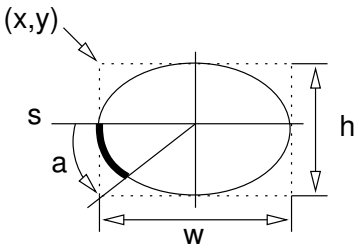
Merk op dat de roosterlijnen een pixel breed zijn en dat w en h niet noodzakelijk deelbaar zijn door het aantal kolommen en rijen van het rooster.

Swing laat ook toe om cirkels, ellipsen, cirkelbogen en ellipsbogen te tekenen — al dan niet opgevuld — met de volgende opdrachten:

```
g.drawOval (x, y, w, h);          // cirkels en ellipsen
g.fillOval (x, y, w, h);
```

g.drawArc $(x,\ y,\ w, h,\ s,\ a)$; // bogen
g.fillArc $(x,\ y,\ w, h,\ s,\ a)$; // sectoren

We hebben de parameters van deze methoden geschetst in nevenstaande figuur. Voor een cirkel (of ellips) geef je niet het middelpunt en de straal op, maar de afmetingen van het omhullende vierkant (of rechthoek). Dit betekent onder andere dat de hoofdassen van de ellips steeds evenwijdig moeten zijn met de coördinaatassen.



Bij *bogen* geef je ook nog twee hoeken s en a op. Deze hoeken worden opgegeven in graden, gerekend vanaf de positieve X-as, tegen de wijzers van de klok in (in de figuur geldt s=180 en a=45). De boog wordt in tegenwijzerszin getekend vanaf hoek s tot aan hoek s+a.

5.1.4. Andere krommen

Andere krommen dan cirkels en ellipsen moet je zelf construeren door ze te benaderen als een aaneensluiting van verschillende korte lijnstukjes die je afzonderlijk met een *drawLine* op het scherm tekent. In het volgende fragment trekken we op die manier een 'kromme' door n opeenvolgende punten waarvan de gehele coördinaten in de tabellen x en y zijn opgeslagen.

for (**int** i=1; $i\ <n$; i++)
    g.drawLine ($x[i-1], y[i-1], x[i\ ],\ y[i\ ]$);

Wanneer, zoals in dit voorbeeld, de coördinaten reeds op voorhand zijn berekend, kan je beter de methode *drawPolyline* gebruiken met dezelfde functionaliteit. Parameters zijn de tabellen met X- en Y-coördinaten en het aantal eindpunten n (dus één meer dan het aantal lijstukken).

De grafische methoden uit *Graphics* zijn eerder primitief. Er bestaat bijvoorbeeld geen mogelijkheid om stippellijnen en streepjeslijnen te tekenen, om lijndiktes aan te passen, om figuren op te vullen met gradueel veranderende kleuren of om tekst te zetten in andere richtingen dan de horizontale. Deze mogelijkheden — en nog veel meer — worden wel geboden door de klasse *Graphics2D* die een extensie is van *Graphics*. Deze klasse behoort tot de zogenaamde *Java 2D*-bibliotheek, een aanrader voor iedere programmeur die op een professionele manier met computergrafiek bezig is.

In sommige toepassingen is het nuttig te weten wanneer een component van grootte of van plaats verandert, of wanneer een component wordt verborgen of opnieuw zichtbaar gemaakt. Dergelijke *componentgebeurtenissen* kan je opvangen met een *componentluisteraar* van het type *ComponentListener*.

Hiermee kan je bijvoorbeeld vermijden dat allerhande afmetingen door elke oproep van *paintComponent* telkens opnieuw moeten worden berekend: je hoeft dit enkel te doen wanneer de component van grootte verandert.

g.drawPolyline (*x*, *y*, *n*);

Een gesloten kromme die uit lijnstukjes bestaat is niets anders dan een *veelhoek*. Een dergelijke veelhoek teken je met één van de volgende opdrachten:

```
g.drawPolygon (x, y, n-1);   // omtrek
g.fillPolygon  (x, y, n-1);   // oppervlak
```

In tegenstelling tot *drawPolyline* verbindt *drawPolygon* het laatste punt automatisch terug met het eerste. Het aantal punten is hier dus hetzelfde als het aantal lijnstukken.

Als variant hierop kan je ook een object van het type *Polygon* definiëren en dan *drawPolygon* (of *fillPolygon*) oproepen met dit object als enige parameter.

```
Polygon p = new Polygon (x, y, n-1);
g.drawPolygon (p);
```

Deze techniek is vooral interessant als je de veelhoek *p* niet telkens opnieuw tijdens *paintComponent* hoeft te reconstrueren. De klasse *Polygon* biedt bovendien een aantal extra meetkundige methoden waarmee je bijvoorbeeld kan nagaan of een bepaald punt zich binnen de veelhoek bevindt, wat de kleinste omsluitende rechthoek is, enz.

5.1.5. Werken met tekst

Je kan tekst plaatsen op je tekening met behulp van de volgende methode:

g.drawString (*str*, *x*, *y*);

Dit drukt de string *str* af met de linker*onderhoek* op positie (*x*,*y*). Het lettertype waarmee de tekst wordt gezet, stel je in met *setFont*.

g.*setFont* (**new** *Font* (`"Monospaced"`, *Font.BOLD*, 12));

De *Font*-constructor neemt drie argumenten, de *naam* van het lettertype, het *type* en de *puntgrootte*. De toegelaten lettertypenamen verschillen van versie tot versie, maar de meeste Java-installaties ondersteunen de zes *logische* lettertypenamen uit onderstaande tabel.

`Dialog`	`Serif`	`Monospaced`
`DialogInput`	`SansSerif`	`Symbol`

We raden de lezer aan om een klein testprogramma te schrijven om deze en andere lettertypes zelf uit te proberen.

Het tweede argument van de constructor bepaalt hoe de letters moeten worden afgedrukt. Er zijn vier mogelijkheden:

Font.PLAIN	Gewoon
Font.BOLD	Vetjes
Font.ITALIC	Cursief
Font.BOLD | *Font.ITALIC*	Tegelijk vetjes en cursief

De puntgrootte, het derde argument, geeft een indicatie van de grootte van het lettertype. Dit correspondeert ongeveer met de afmetingen van de hoogste letter, uitgedrukt in 'punt' (1 punt = 1/72 duim). Het is opnieuw afhankelijk van de installatie met hoeveel schermpixels dit correspondeert.

5.2. Het type *Icon*

Uit de vorige hoofdstukken weten we reeds dat we een object van het type *Icon* kunnen gebruiken om als afbeelding te dienen voor een label of een knop. Je kan een afbeelding echter ook (één of meerdere keren) op het scherm brengen als onderdeel van je eigen *paintComponent*, of je kan omgekeerd je eigen tekenroutines gebruiken om nieuwe objecten van het type *Icon* te construeren.

5.2.1. Basisbewerkingen

De interface *Icon* bevat de volgende drie methoden:

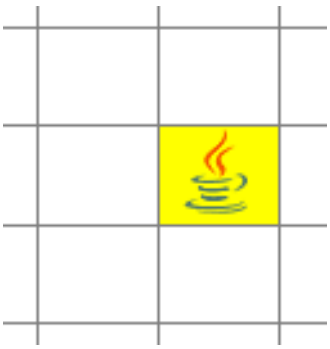
public int *getIconWidth* ();

public int *getIconHeight* ();

public void *paintIcon*(*Component c*, *Graphics g*, **int** *x*, **int** *y*);

Met behulp van de eerste twee kan je de afmetingen van een afbeelding opvragen, met de laatste methode kan je de afbeelding op het scherm plaatsen. Dit gebeurt dan als onderdeel van de methode *paintComponent* van één of andere component.

De parameter *g* is de grafische context waarop je wil tekenen — dezelfde *g* die je als parameter van *paintComponent* doorkrijgt. (Merk op dat *paintIcon* dus niet een methode is van *Graphics*.) Parameters *x* en *y* zijn de coördinaten van het punt waar de linkerbovenhoek van de afbeelding moet terechtkomen.



Uit de eerste parameter — de component waarop je aan het tekenen bent — kan *paintIcon* eventueel extra informatie halen, zoals bijvoorbeeld wat de standaard voorgrond- en achtergrondkleuren zijn voor deze component.

Bovenstaande figuur toont een variant van de *RijKolomView* waarin op het einde van *paintComponent* de volgende opdrachten werden tussengevoegd. De constante *JAVA_LOGO* is van het type *Icon*.

```
int xOffs = (w / aantalKolommen − JAVA_LOGO.getIconWidth ()) / 2;
int yOffs = (h / aantalRijen − JAVA_LOGO.getIconHeight ()) / 2;
JAVA_LOGO.paintIcon (this, g,
    kolom ∗ w / aantalKolommen + xOffs, rij ∗ h / aantalRijen + yOffs);
```

Het effect is dat het Java-logo steeds wordt afgedrukt in het midden van het geselecteerde veld.

5.2.2. Eigen implementaties van *Icon*

Tot nog toe hebben we als implementatieklasse van de interface *Icon* steeds de klasse *ImageIcon* gebruikt, waarmee afbeeldingen kunnen worden voorgesteld die op harde schijf of op het Internet in PNG-, GIF- of JPEG-formaat zijn opgeslagen.

Je kan echter ook zelf afbeeldingen construeren door een eigen implementatie voor *Icon* te schrijven. We illustreren dit aan de hand van een klasse *ArrowIcon* waarmee je eenvoudige driehoekige pijlen kan afbeelden, zoals op de knoppen in de figuur hiernaast. De knop linksboven werd bijvoorbeeld aangemaakt met de volgende eenvoudige opdracht:

```
JButton button =
  new JButton (new ArrowIcon (ArrowIcon.WEST+ArrowIcon.NORTH));
```

Bij de klasse *ArrowIcon* zijn breedte en hoogte steeds gelijk en worden ze opgeslagen in een veld *size*.

```
public class ArrowIcon implements Icon {
    ...
    private int size;

    // Implementatie van interface Icon.
    public int getIconWidth () {
        return size;
    }

    // Implementatie van interface Icon.
    public int getIconHeight () {
        return size;
    }
}
```

We definiëren ook een aantal constanten waarmee de richtingen van de pijlen kunnen worden opgegeven. Deze constanten zijn zodanig gekozen dat ze van *NORTH+WEST* tot *SOUTH+EAST* alle waarden tussen -4 en 4 doorlopen.

```
public static final int NORTH = −3; // driehoeken
public static final int SOUTH = +3;
public static final int WEST = −1;
```

```
public static final int EAST = +1;
public static final int CENTER = 0; // vierkant
```

De vorm van de pijl wordt opgeslagen als een veelhoek *p* van het type *Polygon*. Bij constructie van de *ArrowIcon* wordt deze veelhoek opgebouwd aan de hand van de gewenste grootte en een tabel *COORDINATEN* met beschrijvingen voor de negen verschillende mogelijkheden.

```
private static final double[][][] COORDINATEN =
{ { {−1, 1,  0}, { 1,  0, −1} },    // NORTH+WEST
  { {−1, 0,  1}, {−1, 1, −1} },    // NORTH
  ...
  { { 1, −1, 0}, {−1, 0,  1} }     // SOUTH+EAST
};
```

Er zijn drie constructoren (waarvan twee publiek):

```
// gebruikt opgegeven richting en grootte
public ArrowIcon (int dir, int size) {
    this (COORDINATEN[dir+4][0], COORDINATEN[dir+4][1], size);
}

// gebruikt opgegeven richting en vaste grootte van 12 pixels
public ArrowIcon (int dir) {
    this (dir, 12);
}

private Polygon p;

// gebruikt opgegeven beschrijving en grootte
private ArrowIcon (double[] x, double[] y, int size) {
    this.size = size;
    int[] xco = new int [x.length];
    int[] yco = new int [y.length];
    for (int i=0; i < x.length; i++) {
        xco[i] = (int)((1.0 + x[i])*size/2);
        yco[i] = (int)((1.0 − y[i])*size/2);
    }
    p = new Polygon (xco, yco, x.length);
}
```

Met al deze voorbereidingen wordt de definitie van *paintIcon* eenvoudig.

```
// Implementatie van interface Icon.
public void paintIcon(Component c, Graphics g, int x, int y) {
    g.setColor (c.getForeground());
    g.translate (x, y);
    g.fillPolygon (p);
    g.translate (−x, −y);
}
```

Merk op dat we de kleur van de afbeelding opvragen aan de component waarop er wordt getekend en dat we na afloop het coördinatenstelsel terugschuiven naar zijn oorspronkelijke plaats.

5.3. Interactie met de muis

5.3.1. Muisgebeurtenissen

De meeste componenten onder Swing zijn in staat om *muisgebeurtenissen* te herkennen. Je past hierbij dezelfde technieken toe als bij andere gebeurtenissen: je creëert een object van een gepaste luisteraarsklasse en je registreert dit object bij de component.

Er bestaan drie verschillende types luisteraar voor muisgebeurtenissen. Meestal volstaat een luisteraar van het type *MouseListener* die het indrukken en loslaten van de muis detecteert en die ook reageert wanneer de muiswijzer voor het eerst een component binnenkomt of opnieuw buitengaat. Deze interface heeft de volgende definitie:

```
public interface MouseListener {
    public void mouseClicked (MouseEvent e);

    public void mousePressed (MouseEvent e);
    public void mouseReleased (MouseEvent e);

    public void mouseEntered (MouseEvent e);
    public void mouseExited (MouseEvent e);
}
```

De namen van deze methoden spreken voor zich. Ter verduidelijking vermelden we nog dat *mouseClicked* wordt opgeroepen wanneer de muisknop wordt ingedrukt en terug losgelaten zonder dat de muiswijzer van plaats verandert. Het is best om deze niet tegelijkertijd met *mousePressed* en *mouseReleased* te gebruiken.

Wanneer een *mouseClicked* niet tot de mogelijkheden behoort, kan je in de meeste gevallen beter reageren op een *mouseReleased* dan op een *mousePressed*. Dit biedt de gebruiker de gelegenheid om een verkeerde muisklik te corrigeren door de wijzer buiten de component te bewegen. Je moet dit dan natuurlijk in je programma opvangen.

Wens je ook bewegingen van de muis te detecteren, dan gebruik je een luisteraar van het type *MouseMotionListener*:

```
public interface MouseMotionListener {
    public void mouseDragged (MouseEvent e); // met ingedrukte knop

    public void mouseMoved (MouseEvent e); // knop losgelaten
}
```

Merk op dat het hier nog steeds om gebeurtenissen van het type *MouseEvent* gaat en dat de klasse *MouseMotionEvent* niet bestaat!

Wanneer tijdens het bewegen van de muis met *ingedrukte* knop de component wordt verlaten, blijft Swing de muisgebeurtenissen *nog steeds* naar dezelfde component sturen, tot en met het loslaten van de muis. Men zegt dat de component de muis *vastgrijpt* (Engels: *to grab*). Met andere woorden, een *mouseReleased* of *mouseDragged* wordt soms opgeroepen terwijl de muiswijzer zich niet meer binnen de component bevindt.

Voor het gemak bestaan er twee klassen *MouseAdapter* en *MouseMotionAdapter* die alle methoden van *MouseListener* en *MouseMotionListener* op triviale wijze implementeren. Er is ook een interface *MouseInputListener* die *MouseListener* en *MouseMotionListener* combineert, en een overeenkomstige adapterklasse *MouseInputAdapter*.

De klasse *MouseEvent* bezit de volgende methoden waarmee je de coördinaten van de muiswijzer kan opvragen:

```
public int getX (); // X-coördinaat van de muis

public int getY (); // Y-coördinaat van de muis
```

Met de volgende methoden kan je nagaan of er tegelijkertijd met de muisknop een SHIFT-, CTRL- of ALT-toets werd ingedrukt:

public boolean *isShiftDown* ();

public boolean *isControlDown* ();

public boolean *isAltDown* ();

Om te weten welke muisknop er is ingedrukt, gebruik je *getButton* die één van de constanten *BUTTON1*, *BUTTON2* of *BUTTON3* teruggeeft. In het onderstaande fragment kijken we bijvoorbeeld of de tweede muisknop is ingedrukt:

```
if (event.getButton() == MouseEvent.BUTTON2) {
    // tweede knop is ingedrukt
}
```

Hou er in je programma's rekening mee dat er ook muizen bestaan met slechts één muisknop. Zorg er dus voor dat alles wat de gebruiker in je programma met de tweede muisknop kan doen, ook op een andere manier kan worden bekomen.

Moderne tweeknopsmuizen hebben een *muiswieltje* tussen beide knoppen dat doorgaans gebruikt wordt als alternatief voor de scrollbars. Dit muiswieltje genereert gebeurtenissen van het type *MouseWheelEvent* die je kan opvangen met een *MouseWheelListener*. We gaan hier verder niet dieper op in.

5.3.2. Een voorbeeld

Als voorbeeld construeren we een paneel met daarop een rode cirkel die de gebruiker met de muis kan 'vastgrijpen' en 'verslepen'. Tijdens het verplaatsen wordt de cirkel voorgesteld als een zwarte omtrek en blijft de rode cirkel op de oorspronkelijke positie staan. De cirkel wordt pas effectief verplaatst wanneer je de muisknop loslaat, en dan enkel wanneer je dit doet binnen het paneel. Als extraatje veranderen we de cursor in een 'hand' wanneer hij zich boven de cirkel bevindt.

De klasse *CirclePanel* die we hiervoor ontwerpen is een extensie van *JPanel* en tegelijkertijd een implementatie van *MouseInputListener* (of anders gezegd, van zowel *MouseListener* als *MouseMotionListener*). Aangezien al het tekenwerk binnen *paintComponent* gebeurt, moet een object van het type *CirclePanel* voldoende

informatie opslaan om zowel de rode cirkel als eventueel de zwarte cirkelomtrek op gelijk welk moment te kunnen (her)tekenen. We gebruiken hiervoor de volgende velden:

```
public class CirclePanel extends JPanel implements MouseInputListener {

    private static final int R = 25;
    // Straal van de cirkel

    private int circleX;
    private int circleY;
    // Coördinaten van het middelpunt van de rode cirkel

    private int outlineX;
    private int outlineY;
    // Coördinaten van het middelpunt van de zwarte cirkelomtrek

    private boolean outlineShown;
    // Dient de zwarte cirkel te worden getekend?
}
```

Het is de taak van de verschillende muisgebeurtenisroutines om de waarden van deze velden aan te passen. In een 'echte' toepassing zou het middelpunt van de cirkel, en eventueel de straal, waarschijnlijk worden bijgehouden in een afzonderlijk model.

De implementatie van *paintComponent* is vrij eenvoudig:

```
    protected void paintComponent (Graphics g) {
        super.paintComponent (g);
        g.setColor (Color.RED);
        g.fillOval (circleX − R, circleY − R, 2∗R, 2∗R);
        if (outlineShown) {
            g.setColor (Color.BLACK);
            g.drawOval (outlineX − R, outlineY − R, 2∗R, 2∗R);
        }
    }
```

De methode *mousePressed* activeert de zwarte cirkelomtrek wanneer de muisknop binnen de cirkel werd ingedrukt. Bovendien wordt het punt waarop de cirkel werd 'vastgegrepen' geregistreerd in twee variabelen *mouseX* en *mouseY*.

Hiermee zorgen we ervoor dat de relatieve positie van de muiswijzer en het centrum van de zwarte cirkelomtrek steeds dezelfde blijft tijdens het verschuiven van de cirkel.

```
public void mousePressed (MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    if (inCircle (x,y)) {
        mouseX = circleX − x;
        mouseY = circleY − y;
        outlineX = circleX;
        outlineY = circleY;
        outlineShown = true;
        repaint ();
    }
}
```

(De methode *inCircle* is een hulproutine die kijkt of de gegeven coördinaten zich binnen de rode cirkel bevinden.)

De laatste opdracht (*repaint*) zorgt ervoor dat Swing het paneel uiteindelijk met behulp van *paintComponent* zal hertekenen zodra alle gebeurtenissen zijn verwerkt. We komen later op dit mechanisme terug (zie §5.3.3).

Wanneer de gebruiker uiteindelijk de muisknop loslaat, roept Swing de methode *mouseReleased* op:

```
public void mouseReleased (MouseEvent e) {
    if (outlineShown) {
        int x = e.getX();
        int y = e.getY();
        if (x >= 0 && x < getWidth() && y >= 0 && y < getHeight()) {
            circleX = outlineX;
            circleY = outlineY;
        }
        outlineShown = false;
        repaint ();
    }
}
```

(De **if**-opdracht zorgt ervoor dat een muisklik *buiten* de rode cirkel geen effect heeft.)

Het verplaatsen van de muis met ingedrukte knop zorgt ervoor dat de zwarte cirkelomtrek verschuift:

```
public void mouseDragged (MouseEvent e) {
    if (outlineShown) {
        outlineX = e.getX() + mouseX;
        outlineY = e.getY() + mouseY;
        repaint ();
    }
    else
        mouseMoved (e);
}
```

Wanneer de muis wordt verplaatst zonder dat de knop is ingedrukt, controleert het programma enkel maar of de cursor niet van vorm hoeft te veranderen. Zoals je kan zien aan de implementatie van *mouseDragged* hierboven, gebeurt dit ook wanneer de knop wel is ingedrukt, maar de zwarte cirkelomtrek niet actief is.

```
private static final Cursor HAND_CURSOR
    = Cursor.getPredefinedCursor (Cursor.HAND_CURSOR);

private Cursor cursor;

public void mouseMoved (MouseEvent e) {
    Cursor newCursor;
    if (inCircle (e.getX(), e.getY()))
        newCursor = HAND_CURSOR;
    else
        newCursor = null;
    if (cursor != newCursor) {
        cursor = newCursor;
        setCursor (cursor);
    }
}
```

Tot slot mag je niet vergeten ook nog de overblijvende muisgebeurtenismethoden te implementeren — met een blanco definitie.

```
public void mouseClicked (MouseEvent e) {}
public void mouseEntered (MouseEvent e) {}
public void mouseExited (MouseEvent e) {}
```

5.3.3. De methode *repaint*

Zoals we reeds eerder hebben benadrukt, moeten alle tekenopdrachten op een component worden uitgevoerd vanuit de methode *paintComponent*. Deze methode wordt automatisch opgeroepen telkens wanneer Swing het nodig vindt om de component te hertekenen, bijvoorbeeld wanneer de component voor het eerst op het scherm komt of nadat een gedeelte van de component terug zichtbaar wordt doordat een venster erboven werd verschoven.

Het is echter onmogelijk om *paintComponent* rechtstreeks vanuit het programma te activeren — er is immers geen *Graphics*-object dat je als argument kan meegeven. In plaats daarvan gebruik je de methode *repaint* van de component. Deze methode verplicht het systeem om op een later tijdstip *paintComponent* op te roepen (met een gepast *Graphics*-object als argument), meestal onmiddellijk nadat alle 'hangende' gebeurtenissen zijn verwerkt.

We hebben deze techniek gebruikt in de muisgebeurtenisroutines uit het laatste voorbeeld en ook in de implementatie van *RijKolomView*, alhoewel we dit voorlopig hadden verzwegen. Als view moet *RijKolomView* immers reageren op elke wijziging in het model door zichzelf te hertekenen:

```
public class RijKolomView extends JPanel implements ChangeListener {
    ...
    public void stateChanged (ChangeEvent e) {
        repaint ();
    }
}
```

De methode *repaint* zonder argumenten doet steeds de volledige component hertekenen. Er bestaan echter ook varianten waarmee je zelf het rechthoekig deelgebied kan opgeven dat moet worden hertekend.

```
public void repaint (int x, int y, int width, int height);

public void repaint (Rectangle r);
```

Wanneer meerdere dergelijke *repaint*s elkaar te snel opvolgen, zal Swing ze eventueel tot één enkele *repaint* combineren die een groter rechthoekig gebied bestrijkt.

Na een *repaint* wordt *paintComponent* uiteindelijk aangeroepen met een *Graphics*-argument waarvan de zogenaamde *clip*-rechthoek (van het Engels voor 'afknippen') het opgegeven deelgebied bestrijkt.

De clip-rechthoek van een grafische context beperkt de zichtbare uitwerking van elke grafische bewerking die erop wordt uitgevoerd: enkel die punten worden getekend die ook binnen de huidige clip-rechthoek vallen. Swing is slim genoeg om te weten wanneer een figuur bijvoorbeeld volledig buiten de clip-rechthoek valt, waardoor dergelijke tekenopdrachten sneller kunnen verlopen. Wanneer je een rechthoek opgeeft bij *repaint* betekent dit dus vaak een snelheidswinst.

Je kan nog efficiënter te werk gaan door zelf in je eigen *paintComponent* met de clip-rechthoek rekening te houden. Je kan de clip-rechthoek opvragen met de methode *getClipBounds* van *Graphics*.

```
public Rectangle getClipBounds();
```

In het volgende voorbeeld tekenen we een raster van witte en zwarte vierkantjes van elk 10 op 10 pixels (eigenlijk 8 op 8 met een rand van 1 pixel). We zorgen ervoor dat we enkel die rechthoekjes opnieuw tekenen die met de clip-rechthoek overlappen.

```
protected void paintComponent (Graphics g) {
    super.paintComponent (g);
    Rectangle clip = g.getClipBounds ();

    // afronden
    int left  = clip.x / 10;
    int top   = clip.y / 10;
    int right  = (clip.x + clip.width + 9) / 10;
    int bottom = (clip.y + clip.height + 9) / 10;

    for (int i=left; i < right; i++)
        for (int j=top; j < bottom; j++)
            if (selected[i][j])
                g.fillRect (10*i+1, 10*j+1, 8, 8);
            else
                g.drawRect (10*i+1, 10*j+1, 7, 7);
}
```

De logische tabel *selected* geeft aan of het vierkantje moet worden opgevuld of dat

enkel de omtrek moet worden getekend. Door met de muis op een vierkantje te klikken 'schakelen' we het 'aan of uit'.

```
public void mouseClicked (MouseEvent e) {
    int i = e.getX() / 10;
    int j = e.getY() / 10;
    selected[i][j] = ! selected[i][j];
    repaint (10*i, 10*j, 10, 10);
}
```

Merk op dat we in de muisgebeurtenisroutine enkel een *repaint* aanvragen voor een rechthoek die het aangeklikte vierkantje omsluit.

6. Layout

Zoals we in hoofdstuk 1 hebben aangehaald, voeg je een component aan een container toe met behulp van de methode *add*. De plaats in de container waar de component uiteindelijk zal terecht komen, wordt echter niet door de component zelf bepaald, maar door de *layout-manager* die aan deze container is verbonden.

Een *JPanel* heeft per default een layout-manager van het type *FlowLayout* en het inhoudspaneel van een venster is normaal geassocieerd met een *BorderLayout* manager (zie §6.3). Je kan echter steeds een andere layout-manager instellen met de methode *setLayout* van de container.

6.1. Inleiding

Een layout-manager bepaalt niet alleen de positie van de componenten uit de container maar ook de grootte waarmee die componenten worden afgebeeld. Bovendien berekent hij ook de grootte van de container zelf.

Voor al deze berekeningen gebruikt de layout-manager een aantal eigenschappen van deze componenten: een *minimumgrootte*, een *voorkeursgrootte* en een *maximumgrootte*, die je kan instellen met de methoden *setMinimumSize*, *setPreferredSize* en *setMaximumSize* van de component. Niet elke layout-manager houdt met elk van deze eigenschappen rekening. Het kan dus best zijn dat je component uiteindelijk kleiner wordt afgebeeld dan de minimumgrootte die je hem hebt toebedeeld.

Elk van de *set...Size*-methoden neemt een object van het type *Dimension* als parameter. Bijvoorbeeld:

button.setPreferredSize (**new** *Dimension* (100, 200));

Als je **null** opgeeft als argument van een *set...Size*-methode, zal Swing zelf de overeenkomstige afmeting van de component proberen te bepalen.

Elke container in een toepassing bezit zijn eigen layout-manager. Wanneer we een knop *button* plaatsen op een paneel *panel* in een venster *window*, dan is de layout-manager van *panel* verantwoordelijk voor de grootte en de positie van *button* — waarvoor hij eventueel de minimum-, maximum- of voorkeursgrootte van *button* raadpleegt.

Het is de taak van diezelfde layout-manager om de minimum-, maximum- en voorkeursgrootte van *panel* zelf te bepalen (opnieuw eventueel door op de afmetingen van *button* te steunen) maar het is de layout-manager van het inhoudspaneel van *window* die de *positie* van *panel* in het venster bepaalt. Merk trouwens op dat *button* een eigen (interne) layout-manager heeft die voor de tekst en de afbeeldingen op de knop zorgt.

Samengevat heeft een layout-manager twee belangrijke taken: hij bepaalt de grootte en positie van de deelcomponenten binnen de container en hij berekent de minimum- en voorkeursgrootte van de container zelf.

We bespreken in deze tekst slechts drie layout-managers van de vele die Swing rijk is. We verwijzen naar de elektronische documentatie van bijvoorbeeld *FlowLayout*, *BoxLayout* en *SpringLayout* voor bijkomende informatie.

6.2. Zonder layout-manager

Vooraleer we een aantal standaard layout-managers bespreken, is het nuttig om na te gaan wat er gebeurt wanneer er *geen* layout-manager met de container is geassocieerd. Dit gebeurt wanneer je **null** als argument van *setLayout* hebt opgegeven — of bij *JPanel*, wanneer je een constructor gebruikt met een **null**-argument.

Een layout-manager is niet nodig als je zelf vanuit je programma de plaats van de componenten wil bepalen — iets wat in de praktijk weinig voorkomt, tenzij wanneer de container geen enkele component bevat, als hij bijvoorbeeld enkel als 'doek' dient waarop getekend wordt.

In dit laatste geval kan je zelfs beter geen layout-manager gebruiken, aangezien de meeste layout-managers een container zonder componenten reduceren tot triviale afmetingen (0×0 pixels).

Als je geen layout-manager gebruikt, moet je zelf de plaats en de grootte van de componenten op je container vastleggen. Je doet dit met *setLocation* en *setSize*.

```
JButton button = new JButton ("OK");
button.setLocation (10, 10);
button.setSize     (60, 60);
panel.add (button);
```

In tegenstelling tot *setPreferredSize* neemt *setSize* geen *Dimension* als parameter, maar twee afzonderlijke gehele getallen.

Je kan beide opdrachten ook combineren tot één enkele oproep van *setBounds*.

```
JButton button = new JButton ("OK");
button.setBounds (10, 10, 60, 60);
panel.add (button);
```

Bovendien moet je ook aan Swing vertellen hoe groot je container zelf is. Dit doe je met *setPreferredSize*, als tip voor de bovenliggende layout-manager die de container beheert waarin jouw container zich bevindt. (In principe moet je ook een minimum- en maximumgrootte instellen, maar in de praktijk doet men dit niet vaak.)

```
JPanel panel = new JPanel (null);
panel.setPreferredSize (new Dimension (80, 800));
```

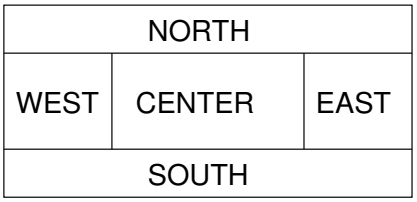
De methoden *setSize* en *setPreferredSize* worden vaak met elkaar verward. Het onderscheid is nochtans eenvoudig: *setPreferredSize* doet een suggestie aan de layout-manager en mag dikwijls worden weggelaten, terwijl *setSize* enkel gebruikt wordt wanneer er geen layout-manager is (of in een klasse die zelf een layout-manager implementeert). Is er toch een layout-manager en geef je toch een *setSize* op, dan zal die worden tenietgedaan door een *setSize* van de layout-manager zelf.

In ons voorbeeld gebruiken we *geen* layout-manager voor de layout van *button*, terwijl er wellicht *wel* een layout-manager wordt gebruikt door de component die het paneel uiteindelijk zal bevatten, vandaar dat we een *setSize* oproepen voor de knop en een *setPreferredSize* voor *panel*.

6.3. *BorderLayout*

Als eerste voorbeeld van een layout-manager bespreken we de klasse *BorderLayout*. Een dergelijke *BorderLayout*-manager verdeelt de container die hij beheert

in vijf gebieden, zoals geschetst in de figuur. Elk gebied kan hoogstens één component bevatten.



Wanneer je een component aan de container toevoegt, moet je het corresponderende gebied aanduiden met een gepaste constante in het tweede argument van *add* — zoals in het onderstaande voorbeeldfragment.

```
JPanel panel = new JPanel ();
panel.setLayout (new BorderLayout ());
 ...
panel.add (buttonE, BorderLayout.EAST);
panel.add (buttonW, BorderLayout.WEST);
panel.add (buttonN, BorderLayout.NORTH);
panel.add (buttonS, BorderLayout.SOUTH);
panel.add (buttonC, BorderLayout.CENTER);
```

Een *add* zonder tweede parameter plaatst de opgegeven component in het middengebied. Dit hebben we eerder al toegepast bij het inhoudspaneel van een venster dat slechts één component bevat.

De voorkeursgrootte van een container met een *BorderLayout*-manager is net ruim genoeg om al zijn componenten met hun voorkeursgrootte te bevatten. Dit betekent bijvoorbeeld dat de voorkeursbreedte gelijk is aan het maximum van de voorkeursbreedte van de bovenste component, de voorkeursbreedte van de onderste component en de som van de voorkeursbreedten van de drie andere.

Bij het opvullen van de container neemt de layout-manager de voorkeurshoogte van de noord- en zuidcomponent en de voorkeursbreedte van de west- en oostcomponent over als werkelijke afmeting. Alle andere afmetingen worden aangepast zodat de componenten de vijf gebieden volledig opvullen. Wellicht zullen de noordelijke en zuidelijke componenten dus breder uitvallen dan hun voorkeursbreedte en zal de centrale component totaal andere afmetingen hebben dan oorspronkelijk ingesteld.

Omdat een container niet noodzakelijk met zijn voorkeursgrootte wordt voorgesteld — dit hangt af van de bovenliggende layout-manager en dus onrechtstreeks van de gebruiker die een venster kan verkleinen of vergroten — kan een *BorderLayout* soms ongewenste effecten hebben: de centrale component kan volledig verdwijnen en de westelijke en oostelijke componenten — of in het ergste geval, de zuidelijke en noordelijke componenten — kunnen elkaar overlappen. Om die

reden wordt *BorderLayout* liefst enkel gebruikt wanneer er slechts één of twee componenten tot de container behoren.

6.4. *GridLayout*

Een container met een layout-manager van het type *GridLayout* schikt zijn componenten in een rooster met een vast aantal rijen en kolommen. Als voorbeeld verwijzen we naar het paneel met dobbelsteenknoppen in de figuur op pagina 22.

De standaardconstructor van *GridLayout* heeft twee argumenten: een aantal rijen en een aantal kolommen. Hiervan hoef je er slechts één op te geven — voor de andere vul je nul in. Zijn zowel rij- als kolomaantal verschillend van nul, dan bekijkt Swing enkel het aantal rijen.

```
panel1.setLayout (new GridLayout (3, 0)); // 3 rijen
panel2.setLayout (new GridLayout (0, 2)); // 2 kolommen
```

In beide gevallen wordt het rooster rij per rij opgevuld, van links naar rechts en van boven naar onder. De eerste component die we met *add* toevoegen komt linksboven terecht, de volgende ernaast, enz. Bij *GridLayout* heeft *add* geen extra parameter.

Merk op dat de laatste rij niet noodzakelijk volledig hoeft te zijn gevuld.

De *GridLayout*-manager bepaalt de voorkeurshoogte van de container door het aantal rijen te vermenigvuldigen met de voorkeursgrootte van de hoogste component — en analoog voor de voorkeursbreedte. De effectieve positie en grootte van de verschillende cellen worden berekend uit de werkelijke grootte en breedte van de container en het aantal rijen en kolommen.

Alle componenten in het rooster worden even groot gemaakt, m.a.w., alle (niet-lege) cellen worden volledig opgevuld. Een *GridLayout* wordt daarom bij voorkeur gebruikt wanneer alle componenten van dezelfde soort zijn.

Met de opdracht '*window.setResizable*(**false**)' kan je verhinderen dat de gebruiker een venster met de muis kan groter of kleiner maken.

6.5. *GridBagLayout*

De klasse *GridBagLayout* is de meest ingewikkelde maar ook de meest flexibele layout-manager uit de Swing-bibliotheek.

Net zoals bij *GridLayout* schikt *GridBagLayout* zijn componenten volgens rijen en kolommen, maar nu zijn niet noodzakelijk alle rijen even hoog, of alle kolommen even breed. Bovendien kan een component tegelijkertijd meerdere cellen van het rooster beslaan en hoeft een component de hem toegewezen cel niet volledig op te vullen.

Om behoorlijk zijn werk te kunnen doen, slaat een *GridBagLayout*-manager een aantal bijzondere gegevens op voor elke component die in de container moet worden geplaatst. Deze gegevens worden opgeslagen als objecten van het type *GridBagConstraints*. Ze beschrijven een aantal randvoorwaarden (*constraints*) waaraan de component moet voldoen, zoals zijn rij- en kolomnummer en of de component de volledige cel in beslag neemt of niet.

Het is de taak van de programmeur om voor elke component deze randvoorwaarden op te stellen en door te geven aan de layout-manager. Het volgende fragment toont hoe we te werk gaan voor de 'Zoek'-knop uit bovenstaande figuur.

```
JPanel panel = new JPanel (new GridBagLayout ());
JButton btnZoek = new JButton ("Zoek");

GridBagConstraints gbc = new GridBagConstraints ();
gbc.insets  = new Insets (3, 5,  3,  5);
gbc.anchor = GridBagConstraints.WEST;
gbc.gridx = 4;
gbc.gridy = 0;
gbc.gridwidth = 1;
```

Het vergt enige ervaring om een visueel aangename layout te creëren met behulp van een layout-manager. Daarom werken beginners vaak zonder layout-manager en geven ze alle posities van componenten expliciet op.

We raden deze werkwijze echter ten zeerste af. Layout-managers houden immers automatisch rekening met verschillen van bedrijfssysteem tot bedrijfssysteem, machine tot machine en gebruiker tot gebruiker en dit is niet eenvoudig in je eigen programma na te bootsen. Een *GridBagLayout* is krachtig genoeg voor 99% van alle toepassingen.

```
gbc.gridheight = 2;
gbc. fill      = GridBagConstraints.BOTH;

panel.add (btnZoek, gbc);
```

Eerst maken we een object *gbc* aan van het type *GridBagConstraints* en vullen we verschillende van zijn velden in (we komen dadelijk terug op hun betekenis). Bij de finale *add* geven we deze randvoorwaarden dan door als tweede argument. De component stuurt ze dan zelf door naar zijn layout-manager die ze kopieert naar een interne voorstelling. Dit betekent dat je hetzelfde object *gbc* mag gebruiken voor verschillende componenten met gelijkaardige randvoorwaarden.

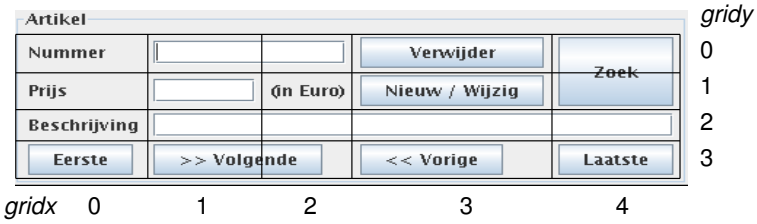
```
gbc.gridheight = 1;
gbc.gridwidth = 1;
gbc.gridx = 0;

JLabel lblNummer = new JLabel ("Nummer");
gbc.gridy = 0;
panel.add (lblNummer, gbc);

JLabel lblPrijs = new JLabel ("Prijs");
gbc.gridy = 1;
panel.add (lblPrijs, gbc);

JLabel lblBeschrijving = new JLabel ("Beschrijving");
gbc.gridy = 2;
panel.add (lblBeschrijving, gbc);
```

Merk op dat we telkens slechts die velden van *gbc* wijzigen die verschillen van component tot component — in dit geval de Y-positie van de labels binnen het rooster.

Zoals je reeds uit bovenstaande voorbeelden hebt kunnen opmaken, bevat de klasse *GridBagConstraints* heel wat attributen waarmee we het gedrag van de layoutmanager kunnen sturen. Hieronder volgt een kort overzicht.

gridx, *gridy*

Geven de kolom- en rijpositie aan van de component in het rooster. De 'Laatste'-knop heeft bijvoorbeeld *gridx*=4 en *gridy*=3 (zie figuur).



Bij componenten die meer dan één cel beslaan (zoals de 'Volgende' en de 'Zoek'-knop), geef je de coördinaten op van de linkerbovenhoek.

gridwidth, *gridheight*

Bevatten het totaal aantal roostercellen dat een component in beslag neemt — in de breedte en in de hoogte. Bij een nieuwgecreëerd *GridBagConstraint*-object zijn deze waarden beide gelijk aan 1.

In ons voorbeeld geldt *gridwidth*=4 voor het onderste tekstveld en *gridheight*=2 voor de 'Zoek'-knop.

fill

Beschrijft of de component moet worden groter gemaakt om het gebied dat eraan is toegewezen volledig te vullen. Dit veld kan worden ingevuld met vier verschillende constanten:

NONE	De component behoudt zijn voorkeursafmetingen.
VERTICAL	De component krijgt de volledige hoogte toegewezen, maar behoudt zijn voorkeursbreedte.
HORIZONTAL	De component krijgt de volledige breedte toegewezen, maar behoudt zijn voorkeurshoogte.
BOTH	De component wordt vergroot totdat hij het volledige toegewezen gebied bestrijkt.

(Elk van deze constanten hoort bij de klasse *GridBagConstraints*.)

Bijna alle componenten in het voorbeeld hebben *fill=HORIZONTAL*, behalve de 'Zoek'-knop die *BOTH* gebruikt en de onderste rij knoppen waarvoor *fill=NONE*.

anchor

Wanneer de component het toegewezen gebied niet volledig opvult, bepaalt dit attribuut waar ergens in dit gebied de component zal worden geplaatst. Je kan voor dit veld de volgende constanten gebruiken. Hun betekenis spreekt voor zich. De defaultwaarde is *GridBagConstraints.CENTER*.

NORTHWEST	*NORTH*	*NORTHEAST*
WEST	*CENTER*	*EAST*
SOUTHWEST	*SOUTH*	*SOUTHEAST*

Zoals je aan de 'Volgende'- en 'Vorige'-knop kan zien, is *anchor* in ons voorbeeld gelijk aan *GridBagConstraints.WEST*.

ipadx*, *ipady

Extra *interne* marge die rond de component wordt voorzien. Bij het berekenen van de grootte van het toegewezen gebied wordt twee keer *ipadx* bij de voorkeursbreedte opgeteld en twee keer *ipady* bij de voorkeurshoogte.

Wanneer *fill=NONE* heeft dit als effect dat er een extra lege ruimte rond de component wordt vrijgehouden van minstens *ipadx* breed (zowel links als rechts) en minstens *ipady* hoog (zowel boven als onder). In ons voorbeeld hebben we deze randvoorwaarden niet toegepast.

insets

Extra *externe* marge rond het gebied dat aan de component wordt toegewezen, uitgedrukt als een object van het type *Insets* (zie ook §6.6). In tegenstelling tot *ipadx* en *ipady* is een lege rand rond de component ook zichtbaar wanneer *fill* niet gelijk is aan *NONE*.

In het voorbeeld hebben we op die manier voor elke component een linker- en rechtermarge van 5 pixels en een boven- en ondermarge van 3 pixels voorzien.

Tot nog toe hebben we steeds stilzwijgend verondersteld dat de container wordt afgebeeld met zijn voorkeursgrootte. Wat gebeurt er wanneer de container zelf groter of kleiner wordt gemaakt?

Wanneer je geen bijzondere voorzorgen neemt, verandert de layout-manager niets aan de grootte of de onderlinge positie van de cellen en houdt hij het geheel gecentreerd in de container. Is dit niet het gedrag dat je wenst, dan moet je de velden *weightx* en *weighty* van het *GridBagConstraint*-object aanpassen (twee reële variabelen met waarden tussen 0.0 en 1.0).

De extra horizontale ruimte die vrijkomt wordt verdeeld over de kolommen volgens de opgegeven *weightx*-gewichten, de extra vertikale ruimte wordt verdeeld over de rijen volgens hun *weighty*-gewichten. Je moet meestal wel wat experimenteren vooraleer je het gewenste effect bereikt.

6.6. Schrijf je eigen layout-manager

In plaats van één van de standaardlayout-managers te gebruiken, kan je ook zelf een layout-manager ontwerpen, hoewel er daar zelden nood aan is. Hiertoe schrijf je een klasse die één van de interfaces *LayoutManager* of *LayoutManager2* implementeert.

In het voorbeeld dat we hieronder bespreken, gebruiken we de eerste interface. Ook *GridLayout* en *FlowLayout* zijn op deze interface gebaseerd en hun broncode vormt trouwens een goed voorbeeld van hoe je een layout-manager moet bouwen. De tweede interface, zelf een uitbreiding van de eerste, dient voor meer ingewikkelde layout-managers, zoals *GridBagLayout*, die zogenaamde 'beperkingsobjecten' (*constraint objects*) nodig hebben.

Elke layout-manager voorziet vijf methoden die worden opgeroepen door de container waarbij de layout-manager is geïnstalleerd. We geven een kort overzicht:

void *addLayoutComponent* (*String*, *Component*)

Deze methode wordt opgeroepen wanneer je bij *add* een extra stringargument opgeeft als je een component op de container plaatst. Op deze manier wordt de layout-manager op de hoogte gesteld van welke string er bij welke component hoort. Dit komt bijvoorbeeld voor bij een *BorderLayout*-manager: de constanten *NORTH* en *SOUTH* zijn namelijk strings.

void *removeLayoutComponent* (*Component*)

Deze tegenhanger van *addLayoutComponent* wordt opgeroepen wanneer een component uit de container wordt verwijderd.

***Dimension preferredLayoutSize* (*Container*)**

Wanneer Swing om de één of andere reden wenst te weten wat de voorkeursgrootte is van een container, roept hij de methode *getPreferredSize* van deze container op. Als de gebruiker niet zelf een voorkeursgrootte voor de container heeft ingesteld (met *setPreferredSize*) wordt deze bepaald door de layout-manager met behulp van deze methode.

***Dimension minimumLayoutSize* (*Container*)**

Heeft dezelfde functie als *preferredLayoutSize*, maar dan voor de minimumgrootte in plaats van de voorkeursgrootte

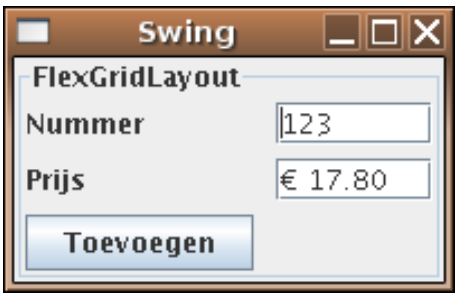
void *layoutContainer* (*Container*)

Deze methode wordt opgeroepen wanneer de container voor het eerst wordt afgebeeld of wanneer hij van grootte verandert. Hier bepaal je de grootte en de positie van de componenten die op de container werden geplaatst door voor elke component *setSize*, *setBounds* of *setLocation* uit te voeren met de juiste argumenten.

Let op! Je mag er niet zomaar van uitgaan dat *preferredLayoutSize* of *minimumLayoutSize* reeds zijn opgeroepen vooraleer deze methode wordt aangesproken.

Merk op dat de interface *LayoutManager* geen methode *maximumLayoutSize* bevat, maar *LayoutManager2* wel.

We bespreken kort een zelfgeschreven layout-managerklasse met de naam *FlexGridLayout* die een beetje het midden houdt tussen een *GridLayout*- en een *GridBagLayout*-manager. Zoals je kan zien in de afbeelding hiernaast, wordt de container in cellen onderverdeeld die in een rooster worden geschikt.



In tegenstelling tot bij *GridLayout* zijn hier niet alle cellen even groot, maar heeft elke kolom van het rooster een eigen breedte en elke rij een eigen hoogte. We laten een kleine tussenruimte tussen de cellen, maar voor de rest vult elke component zijn cel volledig op.

De constructor van *FlexGridLayout* heeft drie parameters: het aantal kolommen *kol* en de horizontale en vertikale tussenruimte tussen de cellen (*horspacing* en *verspacing*). Het aantal rijen wordt onrechtstreeks bepaald door het aantal componenten in de container. De container wordt rij per rij opgevuld.

public *FlexGridLayout* (**int** *kol*, **int** *horspacing*, **int** *verspacing*)

De constructor slaat deze drie waarden op in gelijknamige attributen van de layout-manager.

Omdat onze layout-manager geen gebruik maakt van strings om de verschillende posities van de componenten te benoemen, kunnen we voor *addLayoutComponent* en voor *removeLayoutComponent* volstaan met een triviale implementatie. (We kunnen deze methoden niet zomaar weglaten omdat we anders de interface *LayoutManager* niet implementeren.)

De voorkeursgrootte van de container bepalen we op de volgende manier: de voorkeursbreedte is de som van de voorkeursbreedtes van de kolommen en de voorkeurshoogte is de som van de voorkeurshoogten van de rijen. De voorkeursbreedte van een kolom is dan het maximum van de voorkeursbreedten van de verschillende componenten in deze kolom en de voorkeurshoogte van een rij vinden we op een analoge manier.

Dit alles kunnen we niet berekenen zonder informatie over de componenten die aan onze container zijn toegevoegd. Hiervoor gebruik je de volgende methoden van de klasse *Container*:

```
public int getComponentCount ();
// aantal componenten in de container

public Component getComponent (int n);
// component met het gegeven volgnummer
```

Om de voorkeursgrootte van een component terug te vinden, roep je gewoon *getPreferredSize* op voor deze component. Bij de voorkeursbreedte of -hoogte moeten we ook nog de spatiëring tussen de cellen optellen en de interne marge die door een eventuele rand (*border*) wordt ingenomen (zoals de titelrand in de figuur). Randen worden aan de *binnenkant* van de component aangebracht (en worden dus getekend *bovenop* de aanwezige inhoud). De afmetingen van de huidige rand van een component bekom je met de volgende methode van *JComponent*:

```
public Insets getInsets();
```

Het object van het type *Insets* heeft velden *bottom*, *left*, *right* en *top* die de onder-, linker-, rechter- en bovenafmetingen van de huidige rand bevatten.

Uiteindelijk ziet de implementatie van *preferredLayoutSize* er zo uit:

```
public Dimension preferredLayoutSize (Container c) {
    int aantal = c.getComponentCount();
    int rij  = (aantal + kol −1) / kol;
```

```
        Insets insets = c.getInsets();

        int hoogte = insets.top + insets.bottom;
        for (int i=0; i < aantal; i+=kol) {
            int maxh = 0;
            for (int nr=i; nr < i+kol && nr < aantal; nr++) {
                Component comp = c.getComponent(nr);
                if (comp.getPreferredSize().height > maxh)
                    maxh = comp.getPreferredSize().height;
            }
            hoogte += maxh;
        }

        // en analoog voor de breedte
        ...

        return new Dimension
            (breedte + (kol-1)*horspacing, hoogte + (rij-1)*verspacing);
    }
```

De implementatie van *minimumLayoutSize* verloopt volgens hetzelfde stramien.

De methode *layoutContainer* is de meest ingewikkelde van de vijf. Eerst en vooral moet er een aantal berekeningen uit *preferredLayoutSize* worden overgedaan. We kunnen die namelijk niet door *preferredLayoutSize* laten opslaan en nu opnieuw gebruiken, omdat we niet zeker zijn in welke volgorde *preferredLayoutSize* en *layoutContainer* zullen worden opgeroepen.

Daarnaast moeten we ook rekening houden met het feit dat de grootte van de container waarvan we de layout moeten doen vaak niet de voorkeursgrootte is. Bij onze *FlexGridLayout* kiezen we in dit geval een layout in roostervorm waarbij we de relatieve verhoudingen tussen de kolombreedtes (of rijhoogtes) zo goed mogelijk laten overeenkomen met de relatieve verhoudingen wanneer de container met zijn voorkeursgroote wordt afgebeeld (zie figuur).

We berekenen in de methode *layoutContainer* eerst wat de coördinaten van alle componenten zouden zijn als de container zijn voorkeursgrootte had, en als er geen marges en tussenruimten zouden zijn. De tabellen *kolx* en *rijy* bevatten de X- en Y-coördinaten van de kolommen en de rijen (met één extra index voor de totale breedte en hoogte).

```
public void layoutContainer (Container c) {
    int aantal = c.getComponentCount();
    if (aantal == 0)
        return;
    int rij = (aantal + kol − 1)/kol;
    int[] kolx = new int[kol + 1];
    int[] rijy = new int[rij + 1];

    int x = 0;
    for (int i=0; i < kol; i++) {
        kolx[i] = x;
        int maxb = 0;
        for (int nr=i; nr < aantal; nr+=kol) {
            Component comp = c.getComponent (nr);
            if (comp.getPreferredSize ().width > maxb)
                maxb = comp.getPreferredSize ().width;
            }
        x += maxb;
    }
    kolx[kol] = x; // x bevat de totale breedte

    // en op analoge wijze voor rijy en y (de totale hoogte)
```

In een volgende stap herschalen we alle coördinaten naar de werkelijke hoogte en breedte van de container die we opvragen met *getWidth* en *getHeight* en waarvan we de marges en tussenruimtes aftrekken.

```
    Insets insets = c.getInsets ();
    int w = c.getWidth()−insets.left−insets.right−(kol−1)∗horspacing;
    int h = c.getHeight()−insets.top−insets.bottom−(rij−1)∗verspacing;

    for (int i=0; i <= kol; i++)
        kolx[i] = kolx[i]∗w / x;
    for (int i=0; i <= rij; i++)
        rijy[i] = rijy[i]∗h / y;
```

En tot slot plaatsen we alle componenten op hun juiste plaats met *setLocation* (merk op dat we nu wel met de marges en tussenruimtes moeten rekening houden).

```
        for (int nr=0; nr < aantal; nr++) {
            int k = nr % kol;
            int r = nr / kol;
            c.getComponent (nr)
                .setBounds ( kolx[k] + insets.left  + k*horspacing,
                             rijy[r] + insets.top + r*verspacing,
                             kolx[k+1]-kolx[k],
                             rijy[r+1]-rijy[r] );
        }
    }
```

7. Vensters

Elke GUI-toepassing bezit één of meerdere vensters. Meestal stellen we een venster voor als een object van het type *JFrame*, maar in dit (korte) hoofdstuk bespreken we ook de klasse *JDialog* waarmee je zogenaamde *dialoog*vensters kunt oproepen. Daarnaast behandelen we ook het gebruik van venstergebeurtenissen.

7.1. Dialoogvensters

De klasse *JDialog* verschilt van *JFrame* eigenlijk slechts in twee punten: dialoogvensters hebben een zogenaamde *ouder* en kunnen ook *modaal* zijn.

De ouder van een dialoogvenster kan een gewoon venster zijn, of opnieuw een dialoogvenster. Een dialoogvenster wordt automatisch gesloten (of geminimaliseerd) wanneer zijn ouder wordt gesloten (of geminimaliseerd) en komt terug op het scherm wanneer zijn ouder terug zichtbaar gemaakt wordt. Je geeft dit oudervenster op als eerste argument van de constructor.

JDialog dialog = **new** *JDialog* (*ouder*, *titel*);

In de praktijk zijn dialoogvensters vaak modaal. Dit betekent dat andere vensters (behalve dialoogvensters die dit venster als ouder hebben) geen invoer accepteren zolang het dialoogvenster actief is. Om een venster modaal te maken, gebruik je een extra argument **true** in de constructor.

JDialog dialog = **new** *JDialog* (*ouder*, *titel*, **true**); // modale dialoog

Als programmavoorbeeld bespreken we een modaal dialoogvenster waarmee je een gebruikersnummer en wachtwoord kan opvragen (zie figuur). Het venster wordt gesloten wanneer er op één van de knoppen wordt gedrukt.

We implementeren dit voorbeeld als een klasse *PasswordDialog*, een extensie van *JDialog*. Het is de bedoeling dat de klasse wordt gebruikt op de volgende manier:

```
PasswordDialog dialog = new PasswordDialog (parentWindow);
if (dialog.showDialog ()) {
    // OK-knop werd ingedrukt
    String userId = dialog.getUserId ();
    String password = dialog.getPassword ();
    ...
}
else {
    // Cancel-knop werd ingedrukt of venster werd gesloten
    ...
}
```

De methode *showDialog* opent het (modaal) dialoogvenster, laat de gebruiker een gebruikersnaam en wachtwoord intikken en gaat pas verder wanneer het venster terug sluit. Het geeft de waarde **true** terug als de gebruiker op de OK-knop heeft gedrukt om het venster te sluiten, en **false** in andere gevallen. Na afloop kan je met de methoden *getUserId* en *getPassword* de strings opvragen die door de gebruiker werden ingetikt.

De methode *getUserid* (en analoog *getPassword*) wordt gedelegeerd naar het overeenkomstige tekstveld.

```
public String getUserId () {
    return txtUserId.getText ();   // txtUserId is een tekstveld
}
```

Ook de definitie van *showDialog* is vrij elementair.

```
public boolean showDialog () {
    lastPressed = null;
    setVisible  (true);
    return lastPressed == btnOK;
}
```

Omdat het venster modaal is, blijft de opdracht '*setVisible(***true***)*' wachten totdat het venster opnieuw wordt gesloten (verborgen).

De variabele *lastPressed* stelt de knop voor die het laatst werd ingedrukt en wordt ingevuld door de luisteraar van beide knoppen.

```
private JButton lastPressed;

private class ButtonListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        lastPressed = (JButton)e.getSource ();
        setVisible  (false);  // sluit (verberg) het venster
    }
}
```

De waarde van *lastPressed* blijft **null** wanneer de gebruiker het venster op een andere manier sluit — bijvoorbeeld via de knoppen op de titelbalk.

De rest van de implementatie van *PasswordDialog* vergt nog heel wat lijnen Javacode, ook al is die niet bijzonder ingewikkeld. De grootste inspanning gaat naar de schikking van de verschillende componenten op het venster (bijvoorbeeld met een *GridBagLayout*-manager).

Om het gebruik van dialoogvensters nog te vereenvoudigen, bevat de Swingbibliotheek een aantal standaarddialogen die voor de meeste toepassingen volstaan. Deze dialoogvensters kan je oproepen met een aantal statische methoden uit de klasse *JOptionPane*.

We maken onderscheid tussen vier verschillende soorten dialogen die elk met een afzonderlijke methode worden opgeroepen:

JOptionPane.showMessageDialog

Toont een dialoogvenster met een bericht en daaronder een OK-knop waarmee het venster wordt afgesloten.

JOptionPane.showConfirmDialog

Een dialoogvenster dat een bericht toont met daaronder twee knoppen met opschrift 'Yes' en 'No'. Meestal wordt voor dezelfde functionaliteit echter het volgende type gebruikt.

JOptionPane.showOptionDialog

Een bericht met daaronder een aantal knoppen en een opschrift naar keuze.

JOptionPane.showInputDialog

Een bericht en één enkel tekstveld (of combobox). Dit type venster is voor vele toepassingen niet flexibel genoeg — zo kan je de inhoud van het veld bijvoorbeeld niet op juistheid valideren — en wordt daarom vaak vervangen door een eigen extensie van *JDialog*.

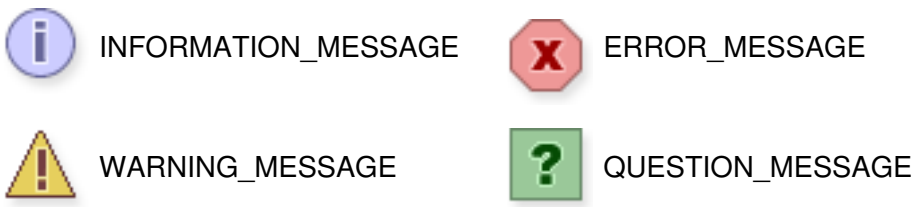
De methode *showMessageDialog* heeft vijf parameters waarvan je er twee, vier of vijf moet opgeven.

```
public static void showMessageDialog
    (Component ouder,
     Object bericht,
     String  titel ,
     int type,
     Icon icon);
```

De parameter *ouder* is het venster dat als ouder zal dienen voor het dialoogvenster. De tweede parameter is het bericht dat je wenst af te beelden. Doorgaans geef je hier een string op. Deze string mag eventueel *linefeed*-lettertekens bevatten wanneer je het bericht over meerdere lijnen wil verdelen.

Het parametertype van de tweede parameter is eigenlijk *Object*: dat betekent dat je behalve strings ook andere soorten 'berichten' in het dialoogvenster kan opnemen: afbeeldingen (type *Icon*), componenten (type *Component*) of tabellen van dergelijke berichten (type *Object[]*) die dan onder elkaar worden afgebeeld.

Het derde argument bevat de tekst die in de titelbalk van het dialoogvenster moet verschijnen. De vierde parameter geeft het *type* aan van het bericht. Naast de vier constanten uit onderstaande figuur, kan je hier ook de constante *JOptionPane.PLAIN_MESSAGE* opgeven voor een 'gewoon' bericht.



Andere standaarddialogen die door Swing worden ondersteund, zijn de zogenaamde *bestandskiezer* (*JFileChooser*) waarmee de gebruiker een bestand of directory kan selecteren, en de *kleurenkiezer* (*JColorChooser*) waarmee je een kleur kan aanduiden.

De vierde parameter bepaalt in de eerste plaats welke afbeelding er in het venster is te zien — of dat er geen afbeelding wordt getoond, zoals bij *PLAIN_MESSAGE*. Het besturingssysteem kan echter ook nog een bijkomende betekenis hechten aan deze parameter.

Tot slot kan je als laatste parameter ook nog een eigen afbeelding meegeven, ter vervanging van de standaardafbeelding die door Swing wordt voorgesteld.

```
JOptionPane.showMessageDialog
    (window,
     "Alles heeft een reden,\nbehalve misschien voetbal.",
     "Belangrijk bericht",
     JOptionPane.INFORMATION_MESSAGE,
     ballIcon);
```

Met de methode *showOptionDialog* kan je de gebruiker laten kiezen tussen verschillende opties. (Voor elke optie komt er een knop onderaan het dialoogvenster.) Dit is de declaratie van deze methode:

```
public static int showOptionDialog
    (Component ouder,
     Object bericht,
     String titel ,
     int optieType,        // niet vergeten!
     int type,
     Icon icon,            // mag ook null zijn
     Object[] opties,
     Object   defaultOptie)
```

De meeste parameters hebben dezelfde betekenis als bij *showMessageDialog*.

De tabel *opties* bevat de opschriften van de knoppen die onderaan het venster worden geplaatst. De laatste parameter definieert de optie waarvan de knop als

De *defaultknop* van een venster wordt automatisch geactiveerd wanneer je op de ENTER- of RETURN-toets drukt terwijl het venster de focus heeft. Je stelt de *defaultknop* in met een opdracht zoals deze:

window.*getRootPane*().*setDefaultButton* (*button*);

Meestal wordt de defaultknop afgebeeld met een dikkere rand.

defaultknop moet worden ingesteld. Net als bij de parameter *bericht*, gebruikt men hier meestal strings, maar ook andere objecttypes zijn toegelaten.

Zodra de parameter *opties* verschillend is van **null**, wordt de parameter *optieType* genegeerd. Hij moet wel een geldige waarde hebben en daarom vullen we hier steeds de constante *JOptionPane*.*DEFAULT_OPTION* in.

```
String[] knoppen = { "Rood", "Groen", "Blauw" };

int answer = JOptionPane.showOptionDialog
    (window,
     "Kies je kleur",
     "Vraagje",
     JOptionPane.DEFAULT_OPTION,
     JOptionPane.QUESTION_MESSAGE,
     null,
     knoppen,
     knoppen[1]);            // 'Groen' is defaultknop
```

De methode *showOptionDialog* geeft de index terug (in de tabel *opties*) van de knop die de gebruiker heeft ingedrukt. Wanneer de gebruiker het venster op een andere manier sluit, is het resultaat negatief.

In deze paragraaf hebben we slechts enkele klassenmethoden van *JOptionPane* besproken. Daarnaast kan je ook objecten van het type *JOptionPane* creëren en manipuleren. Een dergelijke component correspondeert met wat er in het inhoudspaneel van de standaarddialoogvensters te zien is: hij bevat een afbeelding, een tekstbericht en een aantal knoppen. Soms kan je zulk een paneel gebruiken als onderdeel van dialoogvensters die je zelf ontwerpt. We gaan er hier niet dieper op in.

7.2. Venstergebeurtenissen

Een (dialoog)venster genereert zogenaamde *venstergebeurtenissen* wanneer het voor het eerst wordt geopend of het uiteindelijk wordt gesloten, wanneer het wordt geminimaliseerd (geïconiseerd) en daarna terug zichtbaar gemaakt (ge*de*-iconiseerd) en wanneer het de focus krijgt (wanneer het wordt *geactiveerd*) of de focus terug verliest (gedesactiveerd).

In een aantal gevallen is het interessant om deze gebeurtenissen op te vangen: misschien wens je een lange berekening stop te zetten zolang het venster is geïconiseerd, de vensterachtergrond een andere kleur te geven als het de focus krijgt of te vermijden dat de gebruiker per ongeluk een venster sluit waarvan de inhoud nog niet is bewaard.

Venstergebeurtenissen zijn van het type *WindowEvent* en worden opgevangen door een luisteraar van het type *WindowListener*. Deze interface declareert de volgende methoden:

```
public void windowOpened (WindowEvent e);
// Het venster wordt voor het eerst op het scherm getoond

public void windowIconified (WindowEvent e);
public void windowDeiconified (WindowEvent e);
// Het venster wordt ge(de)iconiseerd

public void windowActivated (WindowEvent e);
public void windowDeactivated (WindowEvent e);
// Het venster krijgt (verliest) de focus

public void windowClosing (WindowEvent e);
// De gebruiker probeert het venster te sluiten — zie verder

public void windowClosed (WindowEvent e);
// Het venster wordt effectief gesloten — zie verder
```

Registratie van een *WindowListener* bij een venster gebeurt met *addWindowListener*. Zoals verwacht is er ook een klasse *WindowAdapter* die de interface *WindowListener* op triviale wijze implementeert.

De namen van de gebeurtenismethoden spreken in de meeste gevallen voor zich, enkel het sluiten van een venster verdient iets meer aandacht.

Er zijn drie mogelijke oorzaken voor het sluiten van een venster: het venster kan vanuit het programma worden *verborgen* met *window.setVisible(***false***)*, het kan worden *verwijderd* met *window.dispose()* of het kan worden *gesloten* door de gebruiker (bijvoorbeeld door te klikken op de sluitknop in de titelbalk).

Wanneer het programma een venster verbergt, kan het later opnieuw zichtbaar worden gemaakt met *window.setVisible(***true***)* en vertoont het dan opnieuw zijn oorspronkelijke inhoud. In dit geval wordt er *geen* gebeurtenis gegenereerd.

Een venster dat wordt verwijderd kan echter niet meer zichtbaar worden gemaakt, *setVisible(***true***)* heeft dan geen enkel effect. Het systeem genereert in dit geval een *windowClosed*-gebeurtenis.

Is het de gebruiker die een venster sluit, dan veroorzaakt dit eerst een *windowClosing*-gebeurtenis. Wat er daarna gebeurt hangt af van de instelling van de standaardsluitbewerking van het venster. Er zijn vier mogelijkheden, elk aangeduid door een constante:

WindowConstants.DO_NOTHING_ON_CLOSE

Behalve het genereren van de *windowClosing*-gebeurtenis gebeurt er niets. De programmeur moet het venster zelf verbergen of verwijderen, wellicht als onderdeel van de implementatie van de vensterluisteraar. Dit is een handige instelling wanneer je van plan bent om zelf venstergebeurtenissen op te vangen.

WindowConstants.HIDE_ON_CLOSE

Het venster wordt automatisch verborgen nadat de *windowClosing*-gebeurtenis is verwerkt. Dit is de standaardinstelling.

WindowConstants.DISPOSE_ON_CLOSE

Het venster wordt automatisch verwijderd nadat de *windowClosing*-gebeurtenis is verwerkt. Als gevolg hiervan wordt er dus ook nog een *windowClosed*-gebeurtenis gegenereerd.

JFrame.EXIT_ON_CLOSE

Het programma wordt automatisch beëindigd nadat de *windowClosing*-gebeurtenis is verwerkt.

De standaardsluitbewerking stel je in met de methode *setDefaultCloseOperation* van het venster. Je geeft één van bovenstaande constanten mee als parameter.

Het volgende fragment vraagt de gebruiker om een bevestiging wanneer hij het venster wil sluiten. Het venster zelf krijgt de volgende instellingen mee:

```
window.addWindowListener (new Listener ());
window.setDefaultCloseOperation
    (WindowConstants.DO_NOTHING_ON_CLOSE);
```

en de volgende klasse dient als vensterluisteraar:

```
public class Listener extends WindowAdapter {

    private static final String[] OPTIONS = {"Ja", "Nee" };

    public void windowClosing (WindowEvent e) {
        Window window = e.getWindow ();
        if (JOptionPane.showOptionDialog
            (window,
             "Wens je het venster werkelijk te sluiten ?",
             "Opgelet !",
             JOptionPane.DEFAULT_OPTION,
             JOptionPane.QUESTION_MESSAGE,
             null,
             OPTIONS,
             OPTIONS[1]) == 0)
            window.dispose ();
    }
}
```

Componenten van het type *JInternalFrame* zijn vensters die er uitzien en zich gedragen als gewone vensters, maar zich binnen een container bevinden in plaats van op het scherm. De container die je hiervoor gebruikt, heet een *bureaubladpaneel* en is van het type *JDesktopPane*.

Zoals gewone vensters genereren interne vensters ook venstergebeurtenissen, maar van een ander type. De relevante klassen en interfaces heten nu *InternalFrameEvent*, *InternalFrameListener* en *InternalFrameAdapter*.

8. Invoer en uitvoer in Java (deel 2)

8.1. Eigenschapsbestanden

In enkele bijzondere gevallen is het mogelijk bestandsgegevens in te lezen en uit te schrijven zonder daarvoor het volledige *java.io*-instrumentarium nodig te hebben. Een voorbeeld hiervan zijn de zogenaamde *eigenschapsbestanden* (Engels: *property files*).

8.1.1. De klasse *java.util.Properties*

Een eigenschapsbestand is een tekstbestand dat een aantal zogenaamde *sleutels* met hun *waarden* verbindt (beiden zijn strings). Zo bevat het volgende eigenschapsbestand bijvoorbeeld vier dergelijke sleutel/waarde-paren:

```
# connection.properties
# =====================
server = database.ugent.be
database = finance
userid = kcoolsaet
password = sesamopenu
```

Lege lijnen en lijnen die beginnen met `#` worden genegeerd, andere lijnen bevatten een sleutel/waarde-combinatie met een gelijkheidsteken als scheidingsteken. Traditioneel eindigt de naam van een eigenschapsbestand op de extensie *.properties*, maar dit is niet verplicht.

De inhoud van een eigenschapsbestand kan worden ingeladen als object van het type *Properties* (uit het pakket *java.util*). En aan dit object kan je gemakkelijk de waarde van een bepaalde sleutel opvragen. Het inladen gebeurt met de methode *load*, het opvragen met *getProperty*:

Properties props = **new** *Properties* ();
props.load (**new** *FileInputStream* ("`connection.properties`"));
System.out.println ("`Server: `" + *props.getProperty* ("`server`"));

(Dit schrijft dus '`Server:  database.ugent.be`' op het scherm.)

De methode *load* neemt een *InputStream* als parameter. Hier gebruikten we een gewone *FileInputStream*, maar net zoals eerder kan je hier ook een inputstream ophalen uit het class path met *getResourceAsStream*.

Je kan een *Property*-object ook wijzigen — nieuwe waarden geven aan bestaande sleutels of nieuwe sleutels met nieuwe waarden introduceren. Hievoor gebruik je de volgende methode:

public *String setProperty* (*String sleutel*, *String waarde*);

(Dit geeft de vorige waarde van de sleutel terug, of anders **null**.)

En tot slot kan je het *Property*-object ook uitschrijven naar een *OutputStream* met behulp van

public void *store* (*OutputStream out*, *String commentaar*);

Eerst wordt de commentaarstring uitgeschreven, samen met de datum (en voorafgegaan door een #) en dan volgen alle sleutel/waarde-paren. Merk op dat zowel *load* als *store* een *IOException* kunnen opgooien.

8.1.2. Internationalisatie

Eigenschapsbestanden worden ook gebruikt voor *internationalisatie*. Een goede (GUI-)toepassing is op een zodanige manier ontworpen dat ze gemakkelijk kan worden aangepast aan verschillende talen en regio's. Een geïnternationaliseerde toepassing kan op een eenvoudige manier worden geconfigureerd voor een nieuwe taal of een nieuw land zonder dat de gecompileerde code moet gewijzigd worden.

Meer uitleg over het exacte formaat van een eigenschapsbestand vind je bij de specificatie van de klasse *java.util.Properties*, methode *load*, in de elektronische documentatie.

Dit betekent onder andere dat berichten en opschriften van knoppen en menus niet in het programma zijn gecodeerd maar zijn opgeslagen in afzonderlijke (eigenschaps)bestanden en hierbij biedt Java heel wat ondersteuning.

Als voorbeeld bespreken we een programma met één menu ('Spel') met één enkele menu-optie ('Nieuw'). We willen deze opschriften en de bijbehorende afkortingstoetsen tweetalig maken (Nederlands/Engels).

Het volgende fragment bevat de *niet*-geïnternationaliseerde code die dit menu opbouwt:

```
JMenu menu = new JMenu ("Spel");
menu.setMnemonic (KeyEvent.VK_S);
JMenuItem mnNieuw = new JMenuItem ("Nieuw");
mnNieuw.setMnemonic (KeyEvent.VK_N);
 ...
menu.add (mnNieuw);
```

De eerste internationalisatiestap bestaat erin om een eigenschapsbestand op te stellen met daarin alle taalafhankelijke elementen:

```
# Opschriften en afkortingstoetsen (Nederlandse versie)
game     = Spel
game_afk = S
new      = Nieuw
new_afk  = N
```

(We noemen dit bestand *game.properties*.)

De sleutels zijn taal-onafhankelijk, de waarden verschillen van taal tot taal. De Engelse versie ziet er zo uit:

```
# Opschriften en afkortingstoetsen (Engelse versie)
game     = Game
game_afk = G
new      = New
new_afk  = N
```

en is dan bijvoorbeeld opgeslagen in een bestand met naam *game_en.properties*.

Eigenschapsbestanden voor internationalisatie gebruik je met behulp van de klasse *ResourceBundle* uit het pakket *java.util*. Je maakt een dergelijke bundel aan met de klassenmethode *getBundle*:

```
ResourceBundle bundle = ResourceBundle.getBundle ("prog2/io/game");
```

Als parameter geef je de naam op van het eigenschapsbestand, zonder extensie (en zonder taal-aanduiding *_en*). Het bestand wordt opgezocht door de class loader van de klasse waarin deze methode wordt opgeroepen, m.a.w., in het class path. (Er wordt steeds gezocht vanaf de wortel van het class path, en niet vanuit de directory waarin de 'huidige' klasse zich bevindt!)

Om later een string in deze bundel terug te vinden, gebruik je de methode *getString* met de sleutel als argument. (Merk dus op dat we geen expliciet gebruik maken van *Property*-objecten, alhoewel die achter de schermen wel bestaan, en door *ResourceBundle* worden beheerd.)

In ons voorbeeld wordt dit dus

```
ResourceBundle bundle = ResourceBundle.getBundle ("prog2/io/game");

setText  (bundle.getString ("game"));
setMnemonic (bundle.getString ("game_afk").charAt(0)
                - 'A' + KeyEvent.VK_A);

JMenuItem mnNieuw = new JMenuItem (bundle.getString ("new"));
mnNieuw.setMnemonic (bundle.getString ("new_afk").charAt(0)
                - 'A' + KeyEvent.VK_A);
 ...
add (mnNieuw);
```

Merk op dat we hier toch enige moeite moeten doen om de afkortingstoets vast te leggen: we zetten de string die uit de bundel komt eerst om naar het type **char** (door er de eerste letter van te nemen) en zetten dit letterteken daarna om naar de overeenkomstige gehele code voor *setMnemonic*.

De methode *getBundle* doet haar best om een property-bestand te vinden dat zo goed mogelijk met de taal en het land van de gebruiker overeenkomt. Taal en land worden door het besturingssysteem bijgehouden als drie korte codes:

- Een internationaal overeengekomen code die de taal aangeeft: `en` voor Engels, `nl` voor Nederlands, ...

De Engelse term *internationalization* wordt vaak afgekort tot 'i18n' (omdat er 18 letters staan tussen de begin-i en de eind-n). Men schrijft ook 'l10n' voor het verwante begrip *localization*.

- Een internationale code die het land aangeeft: `FR` voor Frankrijk, `BE` voor België, ...
- Een code die een eventuele variant opgeeft (zoals `UNIX` of `WIN98`).

Voor een UNIX-belg die Nederlands spreekt, zoekt de methode *getBundle* in ons voorbeeld achtereenvolgens naar bestanden met de namen:

```
prog2/io/game_nl_BE_UNIX.properties
prog2/io/game_nl_BE.properties
prog2/io/game_nl.properties
prog2/io/game.properties
```

en gebruikt daarbij het eerste bestand in deze lijst dat bestaat. Het bestand *game.properties* bevat met andere woorden de defaultwaarden voor onze opschriften, terwijl *game_en.properties* enkel gebruikt wordt wanneer we met een Engelstalig gebruiker te doen hebben.

Wanneer je de keuze niet aan het besturingssysteem wil overlaten, kan je ze ook zelf instellen. Dit doe je met een object van het type *Locale*. Deze klasse (uit het pakket *java.util*) heeft drie verschillende constructoren waarvan de parameters overeenkomen met de codes die we hierboven hebben opgesomd:

public *Locale* (*String taal*);
public *Locale* (*String taal*, *String land*);
public *Locale* (*String taal*, *String land*, *String variant*);

Je kan een dergelijk object dan doorgeven als tweede parameter aan *getBundle*:

```
Locale locale = new Locale ("nl", "BE");
ResourceBundle bundle
    = ResourceBundle.getBundle ("prog2/io/game", locale);
```

Om JDOM te kunnen gebruiken, moet je een JAR-archief voor deze bibliotheek in het class path plaatsen – zowel tijdens compilatie als tijdens de uitvoering van je programma. Deze JAR kan je van het Internet ophalen bij `jdom.org`. Je vindt daar ook de elektronische documentatie.

8.2. XML met behulp van JDOM

Tegenwoordig zal men configuratiegegevens vaak opslaan in bestanden waarvan de inhoud voldoet aan de zogenaamde *XML-standaard*. Java kent een heel uitgebreide ondersteuning voor het verwerken van dergelijke bestanden.

De Java-bibliotheek biedt twee verschillende raamwerken voor het gebruik van XML, genaamd *SAX* en *DOM*. We hebben er echter voor gekozen om in deze tekst nog een derde systeem te gebruiken, namelijk de 'open source'-bibliotheek met de naam *JDOM*, omdat zij veruit de gemakkelijkste interface biedt. We zullen ons in deze tekst beperken tot het inlezen van XML-bestanden. Aanmaken en uitschrijven van XML is ook zeer eenvoudig en gebeurt op een analoge manier.

We keren even terug naar het voorbeeld uit paragraaf §2.4 met de prijslijst van goederen. Een XML-bestand met gegevens voor deze prijslijst zou er bijvoorbeeld zo kunnen uitzien:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- goederen.xml / maart 2006 -->
<goederen>
   <detail prijs="160">Extra HD 100GB</detail>
   <detail prijs="135">512 MB extra RAM</detail>
   <detail prijs="175">DVD drive R/W</detail>
   <detail prijs="26" >Gigabit netwerkkaart</detail>
   <detail prijs="161">Externe ADSL-modem</detail>
   <detail prijs="90" >19" TFT ipv 17"</detail>
   <detail prijs="12" >Bluetooth interface</detail>
</goederen>
```

De bovenste lijn geeft aan dat dit om een XML-bestand gaat dat met behulp van UTF-8 is gecodeerd. De tweede lijn is commentaar ten behoeve van de menselijke lezer. De rest van het bestand bevat de feitelijke gegevens.

De volledige inhoud van dit bestand, ook wel het XML-*document* genoemd, wordt in JDOM voorgesteld als één enkel object van de klasse *Document* (om precies te zijn, *org.jdom.Document*) en kan op de volgende manier worden ingelezen:

Document document = **new** *SAXBuilder*().*build*(*input*);

Merk op dat de *build*-methode een *IOException* kan opgooien of een *JDOMException*, die je dus beide ergens moet opvangen.

Er bestaan heel wat methodes waarmee je allerhande informatie over dit *Document*-object kunt opvragen, maar doorgaans gebruik je enkel *getRootElement*, die het *element* teruggeeft dat alle gegevens bevat. In bovenstaand voorbeeld is dit het `goederen`-element dat lijnen 3 tot en met 11 beslaat. Een dergelijk element wordt bijgehouden als een object van het type *Element* (of dus eigenlijk *org.jdom.Element*).

Element root = *document.getRootElement* ();

Een element is een gestructureerd object dat bestaat uit de volgende onderdelen:

- Een *naam*. De naam van het wortelelement is de string "`goederen`". Het element op de vierde lijn van het voorbeeldbestand heeft als naam "`detail`".
- Andere elementen — die *kinderen* van dit element worden genoemd — zoals de zeven `detail`-elementen die in ons voorbeeld de kinderen vormen van het `goederen`-element.
- Zogenaamde *attributen*. De zeven `detail`-elementen hebben elk een *prijs*-attribuut. Elk attribuut heeft een bijbehorende *waarde* (de prijs in EURO).
- Tekstuele inhoud. De `detail`-elementen in ons voorbeeld bevatten tekstuele inhoud (de beschrijving van het voorwerp), bij het `goederen`-element is die leeg.

Elk element heeft en naam, maar niet elk element hoeft tegelijkertijd kindelementen, attributen en tekst te bevatten.

Je gebruikt de methode *getChildren()* om alle kindelementen van een element op te vragen. Deze methode geeft een lijst terug met objecten van de klasse *Element*. JDOM houdt nog geen rekening met de vernieuwingen uit Java 5.0, dus in tegenstelling van wat je zou verwachten, is deze lijst *niet* van het type *List*<*Element*>, maar gewoon *List* en moet je de elementen van de lijst telkens expliciet omzetten van type *Object* naar *Element* met een cast:

```
for (Object obj: element.getChildren()) {
    Element child = (Element)obj;
    // doe iets met child
}
```

Er is een variant van *getChildren* die een naamstring als argument neemt en enkel een lijst van kinderen teruggeeft met deze opgegeven naam. In ons voorbeeld gebruiken we dit om alle goederen één voor één te configureren:

```
List list = root.getChildren ("detail");
int aantal = list.size ();
items = new String[aantal];
prices = new int[aantal];
for (int i=0; i < aantal; i++) {
    Element detail = (Element)list.get (i);
    ...
    prices[i] = ...
    items[i] = ...    // zie verder
}
```

De waarde van een attribuut vraag je op met *getAttributeValue*(*naam*). Deze methode geeft een string terug of **null** wanneer het element geen attribuut heeft met de opgegeven naam.

```
String prijs = detail.getAttributeValue ("prijs");
prices[i] = Integer.parseInt (prijs);
```

Om de tekstuele inhoud van een element op te vragen, gebruik je ofwel *getText*, *getTextTrim* of *getTextNormalize*. De eerste methode geeft de tekst letterlijk terug, in de tweede wordt witte ruimte vóór- en achteraan de tekst automatisch verwijderd en bij de derde wordt bovendien alle tussenliggende witte ruimte genormaliseerd: opeenvolgende spaties en linefeeds worden tot één enkele spatie herleid.

```
items[i] = detail.getTextTrim () + "  (\u20ac " + prijs + ")";
```

8.3. Werken met databanken

Wanneer persistente gegevens heel veel aan verandering onderhevig zijn, is het gebruik van gewone bestanden hiervoor vaak niet de meest geschikte technologie, maar gebruik je liever een zogenaamde *relationele gegevensbank* (of *databank*). Een dergelijke databank wordt meestal bestuurd met behulp van een programmeertaal die *SQL* heet. (Appendix B bevat een hele korte inleiding tot SQL voor de lezer die er niet mee vertrouwd is.)

Java biedt je de mogelijkheid om SQL-opdrachten vanuit je programma te lanceren met behulp van de zogenaamde *JDBC*-API (*Java Database Connectivity*). JDBC bestaat in drie verschillende versies, waarbij elke versie meer mogelijkheden biedt dan de vorige. In deze tekst beperken we ons tot enkele basisfunctionaliteiten uit versie JDBC 1.0. Deze bevinden zich allemaal in het pakket *java.sql*.

8.3.1. Drivers

De JDBC-API zelf is in principe onafhankelijk van de gekozen databanksoftware (Oracle, MySQL, PostgreSQL, ...). Om echter in de praktijk met JDBC te kunnen werken, heb je steeds een zogenaamde *driver* nodig die specifiek is voor de databank die je bij je toepassing wenst te gebruiken. Een dergelijke driver wordt meestal ter beschikking gesteld door de producent van de gegevensbank, maar soms ook door een onafhankelijke firma.

Een drivers moeten worden ingeladen als onderdeel van de toepassing. Dit gebeurt nogal vaak als onderdeel van een **static** blok, op de volgende manier:

```
private final static String PAR_DRIVER = "org.postgresql.Driver";
    ...

static {
    try {
        Class.forName (PAR_DRIVER);
    } catch (ClassNotFoundException e) {
        ... // driver niet gevonden
    }
}
```

De constante *PAR_DRIVER* stelt de volledige klassenaam van de JDBC-driver

Java levert een standaarddriver met de naam *sun.jdbc.odbc.JdbcOdbcDriver*. Dit is een JDBC/ODBC-brug die je in principe enkel voor testdoeleinden hoort te gebruiken. Ze laat toe om onder Microsoft Windows elke databank te gebruiken waarvoor er een ODBC-driver is geïnstalleerd (zoals bijvoorbeeld Microsoft Access). Vergeet niet ook de ODBC-instellingen te configureren als je van deze driver gebruik maakt. Dit gebeurt buiten het Java-programma.

voor. In het voorbeeld wordt een PostgreSQL-gegevensbank gebruikt. De driverklasse wordt geleverd door de gegevensbankproducent en maakt uiteraard geen deel uit van de standaard Java API. Om deze klassen te kunnen gebruiken moet het jar-bestand dat de driverklasse bevat zich in het class path bevinden.

De methode *forName* laadt en initialiseert de driverklasse in de virtuele machine. Tijdens het initialiseren van de driverklasse wordt een instantie aangemaakt en geregistreerd bij de *DriverManager*. De klasse *DriverManager* vormt een gemeenschappelijke toegangslaag tot de verschillende JDBC-drivers van een applicatie. Merk op dat het perfect mogelijk is om tegelijkertijd verschillende drivers actief te hebben. Je toepassing kan op die manier bijvoorbeeld tegelijkertijd gebruik maken van verschillende gegevensbanken van verschillende leveranciers.

8.3.2. Verbindingen

Eenmaal de gepaste JDBC-driver is ingeladen, kan je een verbinding met een gegevensbank opzetten. Een dergelijke verbinding wordt voorgesteld door een object dat tot de interface *Connection* behoort. De eigenlijke implementatie van een verbinding is afhankelijk van de gebruikte gegevensbank.

Om de verbinding aan te maken, gebruik je de methode *getConnection* van *DriverManager*. Deze methode heeft drie argumenten: een JDBC-URL, een gebruikersnaam en een wachtwoord. De gebruikersnaam en het wachtwoord zijn de gebruikersnaam en het wachtwoord die je gebruikt om toegang te krijgen tot de gegevensbank (niet het UNIX-wachtwoord).

De JDBC-URL bestaat uit drie stukken: het protocol, het subprotocol en de 'subname'. Het protocol is `jdbc:` en het subprotocol identificeert de gegevensbankdriver en wordt bepaald door de gegevensbankproducent. In onderstaand voorbeeld is het subprotocol `postgresql:`, bij een JDBC/ODBC-brug is dit bijvoorbeeld `odbc:`.

```
private final static String
    PAR_JDBC_URL = "jdbc:postgresql:finance";
private final static String PAR_LOGIN = "kcoolsaet";
private final static String PAR_PASWOORD = "sesamopenu";

...

try {
    Connection conn =
        DriverManager.getConnection
            (PAR_JDBC_URL, PAR_LOGIN, PAR_PASWOORD);
    try {
        ... // gegevens ophalen, toevoegen, veranderen, ...
    } finally {
        conn.close();
    }
} catch (SQLException e) {
    ... // gegevensbankproblemen
}
```

Het laatste gedeelte van de URL (de subname) beschrijft de gebruikte gegevensbank op een manier die kan verschillen van driver tot driver. Bij een PostgreSQL-gegevensbank kan dit één van de volgende vormen aannemen:

```
naamdatabank
//host/naamdatabank
//host:poort/naamdatabank
```

Hierbij is `naamdatabank` de naam van de gegevensbank, `host` de naam van de server waarop de gegevensbank draait en `poort` de (netwerk)poort waarop de gegevensbank luistert naar aanvragen.

Merk op dat de meeste methoden uit de JDBC API uitzonderingen kunnen genereren van het type *SQLException*. In bovenstaand fragment worden die opgevangen door de buitenste **try**/**catch**-blok. De **try**/**finally**-blok binnenin zorgt ervoor dat de verbinding met de databank zeker wordt afgesloten, zelfs wanneer er een fout optreedt. We hebben hier twee vernestelde **try**-opdrachten nodig omdat ook de *close* een uitzondering kan opgooien.

8.3.3. Opdrachten

Eenmaal de verbinding is gemaakt, kan je de gegevensbank vanuit een Java-programma een aantal SQL-opdrachten laten uitvoeren. Een dergelijke opdracht wordt voorgesteld als een object van het type *Statement*. *Statement* is een interface die SQL-opdrachten naar de gegevensbank stuurt en de resultaten ophaalt.

Een *Statement*-object wordt op de volgende wijze aangemaakt en terug afgesloten:

```
try {
    Statement stmt = conn.createStatement ();
    try {
        ... // SQL-opdrachten uitvoeren
    } finally {
        stmt.close ();
    }
} catch (SQLException e) {
    ...
}
```

Het object *conn* is van het type *Connection* en stelt een verbinding met de gegevensbank voor. De methode *close* zorgt ervoor dat de gereserveerde geheugenruimte in de gegevensbank en in de virtuele machine terug wordt vrijgegeven.

Bij het gebruik van het *Statement*-object wordt een onderscheid gemaakt tussen SQL-opdrachten die rijen of records van de databank als resultaat hebben (zoekopdrachten) en andere opdrachten.

Dit laatste soort opdrachten gebruik je om tabellen (of gebruikers, ed.) aan te maken of om rijen aan een tabel toe te voegen, te wijzigen of te verwijderen. In deze gevallen roep je de methode *executeUpdate* van *Statement* op om de opdracht uit te voeren.

```
public int executeUpdate (String sqlOpdracht) throws SQLException;
```

Het argument van deze methode is een *String* die de SQL-opdracht voorstelt. De waarde is het aantal aangepaste rijen.

In het volgende fragment gebruiken we een gegevensbanktabel *temp* die uit één enkele kolom van getallen bestaat (met naam *nummer*). We voegen een rij aan deze tabel toe die enkel het getal 1 bevat, en veranderen het daarna door 2.

```
// SQL-opdrachten
String voegRijToe = "insert into temp values (1)";
String pasRijenAan = "update temp set nummer=2 where nummer=1";

// uitvoeren SQL-opdrachten
stmt.executeUpdate (maakTabel);
stmt.executeUpdate (voegRijToe);
int updateCnt = stmt.executeUpdate (pasRijenAan);
```

Merk op dat de SQL-opdrachten niet op een puntkomma eindigen. SQL is ook niet hoofdlettergevoelig.

Zoekopdrachten, t.t.z., `SELECT`-opdrachten die een aantal rijen uit de gegevensbank als resultaat hebben, worden uitgevoerd met behulp van de methode *executeQuery* van *Statement*. Het resultaat van de zoekactie is een object van het type *ResultSet*. Je kan dan later de inhoud van dit object (de waarden van de verschillende kolommen) opvragen met behulp van gepaste methoden.

```
public ResultSet executeQuery (String sqlOpdracht) throws SQLException;
```

Elke *ResultSet*-object bevat een wijzer (ook *cursor* genoemd). Deze wijzer staat na het aanmaken van het object vóór de eerste rij. Met deze wijzer wordt het *ResultSet*-object rij na rij overlopen. Je kan enkel gegevens opvragen uit de rij waarnaar de cursor op dat moment wijst. De interface *ResultSet* biedt hiervoor verschillende methoden.

```
public boolean getBoolean (int columnIndex);
public double getDouble (int columnIndex);
public int    getInt    (int columnIndex);
public String getString (int columnIndex);
```

De methode *getXxx* geeft de waarde van de kolom met volgnummer *columnIndex* terug als een element van het type *xxx*. (Merk op dat de eerste kolom volgnummer 1 heeft.) Men raadt aan om de gegevens van een bepaalde rij zoveel mogelijk op te vragen in stijgende volgorde van kolomnummers.

Daarnaast bestaan er ook gelijknamige methoden waarbij je de gewenste kolom mag aanduiden met behulp van zijn naam.

```
public boolean getBoolean (String columnName);
public double getDouble (String columnName);
```

```
public int    getInt    (String columnName);
public String getString (String columnName);
```

De implementatie van *ResultSet* kan verschillen van driver tot driver: in sommige gevallen wordt de inhoud van dit set meteen ingevuld wanneer *executeQuery* wordt opgeroepen, in andere gevallen wordt de gegevensbank slechts gecontacteerd op het moment dat je de gegevens met *getXxx*-methoden opvraagt. Dit verklaart waarom elk van deze methoden een *SQLException* kan veroorzaken.

Om de cursor één rij naar beneden te schuiven, gebruik je de methode *next* (zonder parameters). Deze functie geeft **true** terug als er nog rijen zijn en **false** in het andere geval. Ze wordt vaak gebruikt in een lus van de volgende vorm:

```
ResultSet res = stmt.executeQuery (...);
while (res.next ()) {
    // haal de gegevens op van de huidige rij
}
```

In het volgende fragment maken we een lijst aan van alle boeken die zich in een gegevensbank bevinden. Een boek wordt gekenmerkt door een unieke identificatie, een titel en een prijs. In de gegevensbank worden de boeken bewaard in een tabel *boeken* met kolommen *id*, *titel* en *prijs*. In het geheugen stellen we de boekenlijst voor als een lijst van *Boek*-objecten.

```
private static final String QUERY_BEPAAL_BOEKEN
                                = "select * from boeken";
private static final String PARAM_ID = "id";
private static final String PARAM_TITEL = "titel";
private static final String PARAM_PRIJS = "prijs";

List<Boek> boeken = new ArrayList<Boek> ();
try {
    Statement stmt = conn.createStatement ();
    try {
        ResultSet rs = stmt.executeQuery (QUERY_BEPAAL_BOEKEN);
        while (rs.next ()) {
            Boek boek = new Boek (rs.getString (PARAM_ID),
                                  rs.getString (PARAM_TITEL),
                                  rs.getDouble (PARAM_PRIJS) );
            boeken.add (boek);
        }
```

```
    } finally {
        stmt.close ();
    }
} catch (SQLException e) {
    ...
}
```

We nemen aan dat *conn* een verbinding met de gegevensbank voorstelt die in een ander gedeelte van het programma wordt geopend en gesloten. De constructor van *Boek* neemt drie parameters: de unieke identificatie, de titel (beide strings) en de prijs (een reëel getal) van het boek.

Als een SQL-opdracht meermaals uitgevoerd moet worden of verschillende keren met andere parameters, dan is het efficiënter om een *PreparedStatement*-object te gebruiken in plaats van een gewoon *Statement*-object zoals hierboven. *PreparedStatement* is een interface afgeleid van *Statement*. Tijdens het aanmaken van een prepared statement wordt reeds een SQL-opdracht meegegeven. Wanneer de driver dit ondersteunt wordt deze opdracht onmiddellijk naar de gegevensbank doorgestuurd en daar gecompileerd. Dit zorgt ervoor dat de SQL-opdracht sneller kan worden uitgevoerd.

Het aanmaken van een prepared statement verloopt als volgt. Het object *conn* is een *Connection*-object.

```
private static final String BEWAAR_BESTELLING_OPDRACHT
    = "insert into bestelling (klant, boek, aantal)"
      + " values (?,?,?)";

try {
    PreparedStatement bewaarBestellingStmt =
        conn.prepareStatement (BEWAAR_BESTELLING_OPDRACHT);
    try {
        ... // bestelling bewaren (zie verder)
    } finally {
        bewaarBestellingStmt.close();
    }
} catch (SQLException e) {
    ...
}
```

In dit voorbeeld werd een *PreparedStatement*-object aangemaakt dat gebruikt wordt om rijen toe te voegen aan een tabel `bestelling`. De vraagtekens in de

SQL-opdracht van *bewaarBestellingStmt* stellen parameters voor die later een waarde krijgen. Hetzelfde *PreparedStatement*-object kan opnieuw gebruikt worden met verschillende waarden voor elke parameter.

De volgende lijnen Java-code werken het bovenstaande voorbeeld verder uit. Voor elke klant wordt een lijst *artikels* overlopen. Deze lijst bevat een reeks *Artikel*-objecten. Een *Artikel* bestaat uit een boek en het aantal aangekochte exemplaren van dit boek. Deze gegevens verkrijg je via de methoden *getBoek* en *getHoeveelheid*.

```
// bestelling bewaren
bewaarBestellingStmt.setString (1, klantcode);
for (Artikel artikel : artikels) {
    bewaarBestellingStmt.setString (2, artikel.getBoek().getId());
    bewaarBestellingStmt.setInt   (3, artikel .getHoeveelheid());
    bewaarBestellingStmt.executeUpdate();
}
```

De parameters van een prepared statement worden ingevuld met methoden van de volgende vorm:

```
public void setXxx (int indexParameter, xxx waardeParameter)
```

Het eerste argument bepaalt het volgnummer van de parameter die de meegegeven waarde *waardeParameter* moet krijgen (parameters worden genummerd vanaf 1). Alle parameters moeten op deze manier een waarde krijgen vooral de SQL-opdracht uitgevoerd kan worden. De methoden *executeUpdate* en *executeQuery* voeren dan uiteindelijk de opdracht uit. Net zoals bij *Statement* gebruik je voor zoekopdrachten de methode *executeQuery* en in het andere geval de methode *executeUpdate*.

Hieronder illustreren we nog een andere reden om *PreparedStatement* boven *Statement* te verkiezen. Veronderstel dat we in een tabel met studentengegevens alle studenten met een gegeven familienaam wensen op te zoeken. Deze familienaam wordt gegeven in de vorm van een parameter of attribuut *naam*. We zouden dit kunnen doen op de volgende manier:

```
Statement stmt = conn.createStatement ();
ResultSet rs   = stmt.executeQuery
    ("select * from studenten where name = \'" + naam + "\'");
...
```

of anders, met behulp van een prepared statement:

```
PreparedStatement stmt = conn.prepareStatement
          ("select * from studenten where name = ?");
stmt.setString (1, naam);
ResultSet rs = stmt.executeQuery ();
 ...
```

Het nut van een prepared statement lijkt bij dit voorbeeld op het eerste zicht niet groot. We merken echter op dat wanneer *naam* een aanhalingsteken bevat, enkel het tweede fragment zal werken.

8.3.4. Transacties

Sommige applicaties vereisen dat een reeks SQL-opdrachten ofwel allemaal succesvol zijn ofwel allemaal niet uitgevoerd worden. Het is niet toegelaten dat een deel van de opdrachten wel en een ander deel niet uitgevoerd worden. Als alle opdrachten uitgevoerd konden worden, dan mogen ze permanent gemaakt worden in de databank (*commit* in het Engels), in het andere geval moeten alle reeds uitgevoerde opdrachten ongedaan gemaakt worden (*rollback* in het Engels). Een *transactie* is een dergelijke reeks opdrachten die samen uitgevoerd moeten worden.

De interface *Connection* voorziet de volgende methoden om transactie te implementeren.

```
public void setAutoCommit(boolean autoCommit) throws SQLException;
public void commit() throws SQLException;
public void rollback() throws SQLException;
```

Een *Connection*-object wordt standaard aangemaakt in de mode 'auto-commit'. Dit betekent dat de uitvoering van elke SQL-opdracht onmiddellijk permanent is. Om meerdere opdrachten samen te kunnen uitvoeren moet de auto-commit-modus uitgeschakeld worden. De methode *setAutoCommit* schakelt de auto-commit-modus aan of uit afhankelijk van het argument (respectievelijk **true** of **false**). Indien de modus is uitgeschakeld, wordt het resultaat van een transactie pas permanent na het oproepen van de methode *commit*. Om het resultaat van een aantal SQL-opdrachten ongedaan te maken roep je de methode *rollback* aan.

In het voorbeeld uit de vorige paragraaf kunnen we eisen dat de bestelling van één klant ofwel volledig moet ingevoerd worden ofwel helemaal niet. De code wordt dan als volgt aangepast.

```
try {
    // bestelling bewaren
    conn.setAutoCommit (false);
    bewaarBestellingStmt.setString (1, klantcode);
    for (Artikel artikel : artikels) {
        bewaarBestellingStmt.setString (2, artikel.getBoek ().getId ());
        bewaarBestellingStmt.setInt (3, artikel.getHoeveelheid ());
        bewaarBestellingStmt.executeUpdate ();
    }
    conn.commit ();
} catch (SQLException e) {
    conn.rollback ();
} finally {
    conn.setAutoCommit (true);
}
```

Merk op dat bij gelijktijdige transacties elke transactie een ander *Connection*-object moet hebben. Verschillende transactie kunnen niet gelijktijdig over hetzelfde *Connection*-object verlopen.

9. Modellen (deel 2)

Een aantal belangrijke componenten — lijsten, tabellen en boomdiagrammen — kwamen tot nog toe niet aan bod hoewel ze in professionele toepassingen vaak worden gebruikt. Het zal je niet verrassen dat deze componenten gebruik maken van het MVC-patroon.

9.1. Lijsten

We hebben reeds met de klasse *JList* te maken gehad bij de bespreking van keuzelijsten (zie 2.4). Dergelijke lijsten laten je echter enkel toe om een keuze te maken uit een aantal strings die op voorhand zijn opgegeven en tijdens de loop van de toepassing niet meer veranderen. De klasse *JList* biedt immers niet direct een mogelijkheid om elementen aan de lijst toe te voegen of eruit te verwijderen. Om dit mogelijk te maken, moet je gebruik maken van een model.

9.1.1. Lijstmodellen

Swing stelt een lijst intern voor als een combinatie van verschillende objecten: een visuele component (van het type *JList*) en twee verschillende modellen, een *gegevensmodel* (van het type *ListModel*) en een *selectiemodel* (van het type *ListSelectionModel*).

Het gegevensmodel bepaalt de inhoud van de lijst. Het selectiemodel houdt bij welke van de lijstelementen door de gebruiker zijn geselecteerd. Door bijvoorbeeld twee verschillende lijsten te gebruiken met een gemeenschappelijk selectiemodel, zorg je ervoor dat beide selecties zijn gesynchroniseerd: wat er in de ene lijst wordt ge(de)selecteerd wordt ook automatisch in de andere ge(de)selecteerd. We zullen ons hier echter beperken tot de bespreking van het gegevensmodel.

Lijst en gegevensmodel zijn zeer nauw met elkaar verbonden: vooraleer een lijst-

element kan worden afgebeeld moet het worden opgehaald bij het model en wanneer de inhoud van het model verandert, wordt de visuele component hiervan op de hoogte gesteld zodat de afbeelding van de lijst op het scherm kan worden aangepast.

De interface *ListModel* waartoe het gegevensmodel van een lijst behoort, bezit de volgende methoden:

```
int getSize();      // aantal elementen

Object getElementAt (int index); // element met gegeven index

void addListDataListener (ListDataListener l);
// voeg een luisteraar toe
void removeListDataListener (ListDataListener l);
// verwijder een luisteraar
```

Een gegevensmodel moet dus kunnen vertellen hoeveel elementen de lijst bevat en wat het element met een gegeven index is. Communicatie tussen lijst en model verloopt via gebeurtenissen van het type *ListDataEvent* die door *ListDataListeners* worden opgevangen. Een dergelijk gebeurtenisobject bevat informatie over welk gedeelte van de lijst werd aangepast en of het gaat over toevoegen, verwijderen of aanpassen van elementen.

9.1.2. De klasse *AbstractListModel*

Het is echter weinig of nooit nodig om de interface *ListModel* rechtstreeks te gebruiken. Voor het gemak biedt Swing immers reeds een aantal implementaties van deze interface aan: hieronder bespreken we de abstracte klasse *AbstractListModel* en de klasse *DefaultListModel* komt later nog aan bod (zie §9.1.3).

De klasse *AbstractListModel* zorgt reeds zelf voor de registratie van luisteraars en biedt drie methoden waarmee de juiste *ListDataEvent*-gebeurtenissen op een gemakkelijke manier kunnen worden gegenereerd:

```
protected void fireContentsChanged
                    (Object source, int index0, int index1);

protected void fireIntervalAdded
                    (Object source, int index0, int index1);
```

```
protected void fireIntervalRemoved
                    (Object source, int index0, int index1);
```

Net zoals bij de implementatie van *IntegerModel* (zie §4.1.2) moeten de *fire*-methoden worden opgeroepen nadat er iets aan de inhoud van het model is veranderd. Als parameters geef je een interval op dat aangeeft waar in de lijst de veranderingen hebben plaatsgevonden. Het interval loopt van posities *index0* tot en met *index1* in de lijst. De eerste parameter geeft de gewenste bron van de gebeurtenis aan. In de praktijk is dit meestal **this**, het gegevensmodel zelf.

Als voorbeeld definiëren we een klasse *MultiList* als uitbreiding van *AbstractListModel*. Deze klasse beschrijft een gegevensmodel dat je kan interpreteren als een tweedimensionale tabel waarvan telkens slechts één rij wordt uitgelijst. Met de methode *setRowNr* kan je vanuit het programma bepalen uit welke rij de lijstcomponent zijn gegevens moet halen. De lijst beeldt zichzelf automatisch opnieuw af wanneer met *setRowNr* een nieuwe waarde wordt ingesteld.

De gegevens voor het model worden opgeslagen als een tweedimensionale tabel *tab* van strings.

```
public class MultiList extends AbstractListModel {
    private String[][] tab;

    public MultiList (String[][] tab) {
        this.tab = tab;
    }
    ...
}
```

Het model houdt ook het huidige rijnummer bij.

```
    private int rowNr = 0;

    public void setRowNr (int newRowNr) {
        if (rowNr != newRowNr) {
            rowNr = newRowNr;
            fireContentsChanged (this, 0, tab[0].length-1);
        }
    }
```

Merk op dat het model al zijn luisteraars op de hoogte brengt van elke inhoudsverandering.

Tot slot dienen we ook nog de twee methoden uit *ListModel* te implementeren die nog niet door *AbstractListModel* waren ingevuld.

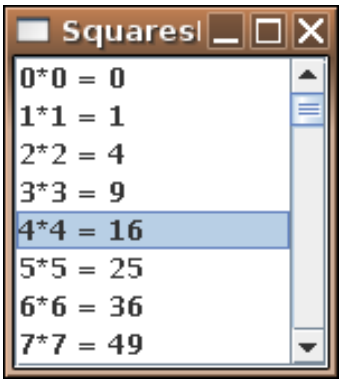
```
public int getSize () {
    return tab[rowNr].length;
}

public Object getElementAt (int index) {
    return tab[rowNr][index];
}
```

Om nu een dergelijk model te gebruiken, maak je gewoon een lijstcomponent aan met dit model als enige argument in een *JList*-constructor:

```
String[][] names = ... ;

ListModel model = new MultiList (names);
JList list  = new JList (model);
 ...
model.setRowNr (3);
```

Een gegevensmodel hoeft niet noodzakelijk alle lijstgegevens in tabelvorm op te slaan. Je kan de lijstinhoud ook pas bepalen op het moment dat ernaar wordt gevraagd (in *getElementAt*).

De klasse *SquaresModel*, waarvan we de code op de volgende bladzijde afdrukken, beschrijft bijvoorbeeld een model van een lijst van de eerste 1000 gehele kwadraten.



Omdat de inhoud van het model niet verandert, is het bijzonder eenvoudig om deze klasse als extensie van *AbstractListModel* te implementeren. De methode *getSize* geeft gewoon de constante 1000 terug en *getElementAt* berekent het kwadraat van de gegeven parameter en zet die op een gepaste manier om naar een string.

De *JList*-constructor (met een tabel van objecten als argument) waarmee je een keuzelijst aanmaakt, bouwt zelf achter de schermen een (anoniem) gegevensmodel op als extensie van *AbstractListModel*. Dit is de definitie:

```
public JList (final Object[] listData) {
    this ( new AbstractListModel() {
              public int getSize() { return listData.length; }
              public Object getElementAt(int i) { return listData[i]; }
            } );
}
```

Het is een goeie oefening om eens een analoog model te ontwerpen die van *listData* een *List* maakt in plaats van een tabel.

```
public class SquaresModel extends AbstractListModel {
    public int getSize () {
        return 1000;
    }

    public Object getElementAt (int index) {
        return index + "*" + index + "  =  " + (index*index);
    }
}
```

Bij lijsten met zeer veel elementen treedt er een kleine complicatie op: om zijn eigen voorkeursgrootte te kunnen bepalen, moet de lijstcomponent alle lijstelementen op voorhand aan het model opvragen — ook wanneer de lijst een onderdeel vormt van een scrollpaneel.

Om dit te verhelpen kan je bij de lijstcomponent een vaste celgrootte instellen met behulp van de methoden *setFixedCellHeight* en *setFixedCellWidth*. Als alternatief kan je ook een prototype voor een cel opgeven waaruit de lijstcomponent dan zelf de vaste celafmetingen berekent. Dit doe je met *setPrototypeCellValue*, zoals in onderstaand fragment.

```
JList list = new JList (new SquaresModel ());
list.setPrototypeCellValue("999*999 = 998001");
...
window.getContentPane ().add (new JScrollPane (list));
```

9.1.3. De klasse *DefaultListModel*

In de praktijk gebruikt men vaak een lijstmodel van het type *DefaultListModel*. Dit model gedraagt zich een beetje zoals een *ArrayList* of een *Vector*: het bevat een tabel van objecten waaraan je elementen kunt toevoegen en waaruit je elementen kunt verwijderen. Deze veranderingen worden meteen zichtbaar gemaakt in de corresponderende lijstcomponent.

Als voorbeeld van het gebruik van een dergelijk model implementeren we een eenvoudige toepassing waarmee je namen aan een lijst kunt toevoegen of geselecteerde namen uit een lijst verwijderen. (Om deze toepassing echt bruikbaar te maken, zou je de namen eigenlijk in een bestand, of beter nog, in een databank, moeten bewaren.)

Het toepassingsvenster bevat een lijst *list* in een scrollpaneel, en onderaan een tekstveld *field* en twee knoppen *addButton* en *deleteButton* (zie figuur).

Wanneer je een naam invoert in het tekstveld en je drukt op de 'Toevoegen'-knop (of op de ENTER-toets) dan wordt hij ingevoegd vóór de geselecteerde positie in de lijst. Is er geen enkele naam geselecteerd, dan komt de nieuwe naam achteraan de lijst.

Door op de 'Verwijder'-knop te drukken, verwijder je de geselecteerde naam terug uit de lijst.

Wanneer een naam in de lijst is geselecteerd, dan wordt dezelfde naam in het tekstvak afgebeeld. Is er niets geselecteerd, dan is het tekstvak leeg en is de 'Verwijder'-knop niet actief.

Het aanmaken van een dergelijke lijst is eenvoudig: je creëert een nieuw (leeg) model, je voegt er eventueel reeds een aantal namen aan toe en dan geef je het model door als parameter aan een *JList*-constructor:

```
model = new DefaultListModel();
if (tab != null)
    for (String naam : tab)
        model.addElement (naam);
list  = new JList (model);
```

De tabel *tab* bevat de strings die we in het begin in de lijst moeten opnemen.

De toepassing gebruikt twee hulp(binnen)klassen, één voor de 'Verwijder'-knop en één voor het tekstveld. De 'Toevoegen'-knop is een gewone *JButton* die het tekstveld als *ActionListener* gebruikt.

De klasse *DeleteButton* is een uitbreiding van *JButton* die tegelijkertijd dient als controller voor het lijstmodel en als luisteraar voor het selectiemodel: de knop moet namelijk worden geactiveerd of gedesactiveerd al naargelang er een lijstelement is geselecteerd of niet.

```
private class DeleteButton extends JButton
    implements ListSelectionListener, ActionListener {

    public DeleteButton (String opschrift) {
        super(opschrift);
        list .addListSelectionListener (this);
        addActionListener (this);
    }

    public void valueChanged (ListSelectionEvent e) {
        if (!e.getValueIsAdjusting ())
            setEnabled (!list .isSelectionEmpty ());
    }

    public void actionPerformed (ActionEvent e) {
        int index = list .getSelectedIndex ();
        if (index >= 0) {
            Namenlijst.this.model.remove (index); // zie opmerking
            int size = Namenlijst.this.model.getSize ();
            if (size != 0) {
                if (index == size)
                    index--;
                list .setSelectedIndex (index);
            }
        }
    }
}
```

De uitdrukking *Namenlijst*.**this**.*model* verwijst naar het veld *model* uit de klasse *Namenlijst* waarvan dit een binnenklasse is. De klasse *JButton* (en dus ook *DeleteButton*) bezit namelijk zelf een veld met de naam *model*.

Merk op dat je het element uit het model verwijdert en niet uit de lijst. Ook de grootte van de lijst kom je via het model te weten. Anderzijds vragen we aan de lijst welke index er is geselecteerd — of dus eigenlijk onrechtstreeks aan het selectiemodel van de lijst.

De klasse *Field* ziet er als volgt uit:

```
private class Field extends JTextField
    implements ListSelectionListener, ActionListener {

    public Field (int breedte) {
        super(breedte);
        list .addListSelectionListener (this);
        addActionListener (this);
    }

    public void valueChanged (ListSelectionEvent e) {
        if (!e.getValueIsAdjusting ()) {
            int index = list .getSelectedIndex ();
            if ( list .isSelectionEmpty ())
                field .setText (null);
            else
                field .setText ((String)list .getSelectedValue ());
        }
    }

    public void actionPerformed (ActionEvent e) {
        String str = field .getText ();
        if (str.length () != 0) {
            int index = list .getSelectedIndex ();
            if (index < 0)
                Namenlijst.this.model.addElement (str); // achteraan
            else {
                Namenlijst.this.model.insertElementAt (str, index);
                list .setSelectedIndex (index + 1);
            }
        }
    }
}
```

Merk opnieuw op hoe we elementen toevoegen aan het model en niet aan de lijst.

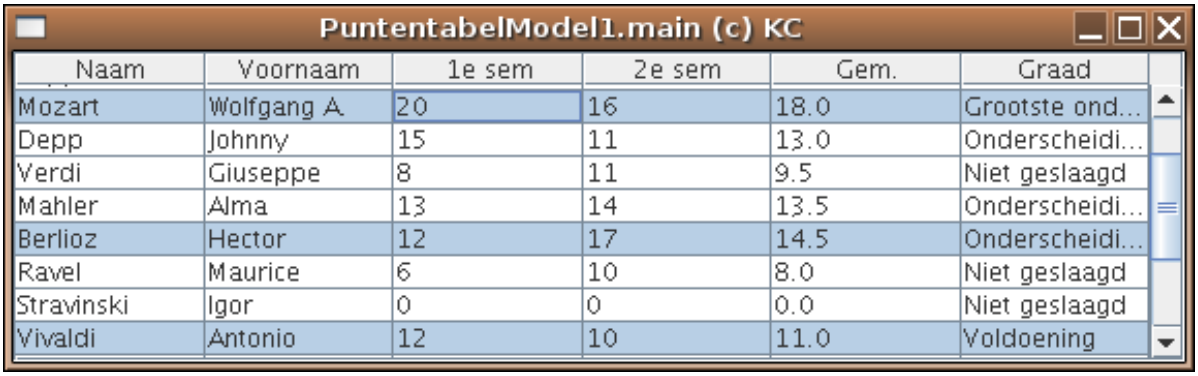
We gebruiken de term *tabel* in deze paragraaf als naam voor een GUI-component (Engels: *table*). Verwar niet met de gelijknamige basisdatastructuur uit Java (Engels: *array*).

9.2. Tabellen

Een tabel (van het type *JTable*) is een component waarmee gegevens in een tweedimensionaal rooster kunnen worden voorgesteld — anders gezegd, een lijst met meerdere kolommen.

Tabellen hebben een *hoofding* die voor elke kolom een titel bevat en je kan een tabel zo instellen dat de inhoud van sommige cellen door de gebruiker kan worden gewijzigd. De gebruiker kan rijen uit een tabel selecteren op dezelfde manier als bij een lijst. Het is ook mogelijk om kolommen te selecteren in plaats van rijen, of (groepen van) afzonderlijke cellen.

Het zou ons te ver leiden om deze klasse tot in detail te bespreken, voor meer informatie verwijzen we dan ook naar de documentatie en de uitgebreide Java-literatuur. We beperken ons hier tot een eenvoudig voorbeeld.



Bovenstaande momentopname toont een tabel met een studentenpuntenlijst. De tabel bevat naam en voornaam van de student, punten behaald in de twee afzonderlijke semesters, bijbehorend jaargemiddelde en de corresponderende graad. De gegevens in deze tabel zijn afkomstig uit een databank.

Bij *JTable* moeten we rekening houden met drie verschillende modellen: een gegevensmodel, een kolommodel en een selectiemodel. Het selectiemodel is identiek aan dat van *JList*. Het kolommodel houdt voornamelijk informatie bij over de breedte van de kolommen en we zullen het hier ook enkel in die context gebruiken. Je hoeft geen eigen kolommodel aan te maken — het standaardkolommodel

dat automatisch bij elke tabel wordt gecreëerd, volstaat.

Het gegevensmodel bepaalt niet alleen de inhoud van de tabel maar ook de kolomtitels. Doorgaans leid je een eigen model af van *AbstractTableModel*. Een minimaal model hoeft dan enkel de methoden *getRowCount*, *getColumnCount* en *getValueAt* te implementeren.

We illustreren dit aan de hand van de puntenlijsttabel:

```
public class PuntentabelModel1 extends AbstractTableModel {

    private List<Punten> data;

    public int getRowCount () {
        return data.length;
    }

    public int getColumnCount () {
        return 6;
    }
     ...
}
```

De variabele *date* bevat een lijst met puntengegevens, één voor elke student. (Deze lijst wordt bij het aanmaken van het model opgevuld met gegevens uit een databank. We houden met andere woorden een interne kopie bij van de databankgegevens — om redenen van efficiëntie. Dit is enkel mogelijk wanneer de databank niet al te groot is.)

De klasse *Punten* is een eenvoudige klasse waarvan de objecten (publieke) velden bevatten met de gegevens van één student:

```
public class Punten {

    public String familienaam;
    public String voornaam;

    public int[] punten; // één puntental per semester
     ...
}
```

Deze klasse bevat ook een methode *getGemiddelde()* waarmee de gemiddelde jaarpunten worden berekend (het resultaat is een **double**) en *getGraad()* die de graad teruggeeft dat met dit jaargemiddelde correspondeert. De graad is van hetzelfde type *Graad* dat we eerder bij het spinnermodel hebben gebruikt (zie §4.3.2). Merk dus op dat een aantal gegevens uit de tabel niet rechtstreeks worden opgeslagen als onderdeel van het model, maar worden berekend telkens we ze nodig hebben.

De methode *getValueAt* bepaalt het object dat op de gegeven rij en kolom van de tabel moet worden afgebeeld.

```
public Object getValueAt (int row, int column) {
    Punten p = data.get (row);
    switch (column) {
        case 0:
            return p.familienaam;
        case 1:
            return p.voornaam;
        case 2:
        case 3:
            return "" + p.punten[column − 2];
        case 4:
            return "" + p.getGemiddelde ();
        case 5:
            return p.getGraad ();
        default:
            return null; // zou niet mogen gebeuren
    }
}
```

Het bleek in ons voorbeeld ook nodig om een aantal andere methoden van *AbstractTableModel* opnieuw te definiëren. We wensen bijvoorbeeld zelf kolomtitels op te geven, in plaats van de standaardstrings ‘A’, ‘B’, ‘C’, ... Hiertoe overschrijven we *getColumnName*:

```
private static final String[] COLUMN_NAMES
    = { "Naam", "Voornaam", "1e sem", "2e sem", "Gem.", "Graad" };

public String getColumnName (int column) {
    return COLUMN_NAMES[column];
}
```

Plaats een tabel steeds in een scrollpaneel, anders moet je bijzondere voorzorgen nemen opdat de hoofding van de tabel niet zou verdwijnen.

Normaal mag de gebruiker *alle* cellen van een tabel van waarde veranderen, terwijl we dit hier enkel willen toelaten voor de semesterpunten. Het model geeft aan of een cel editeerbaar is of niet met behulp van *isCellEditable*.

```
public boolean isCellEditable (int row, int column) {
    return column == 2 || column == 3;
}
```

Wanneer cellen kunnen van waarde veranderen, moet de nieuwe waarde ook worden opgenomen in het model. De tabelcomponent signaleert een celverandering aan het model via de methode *setValueAt*.

```
public void setValueAt (Object value, int row, int column) {
    if (column != 2 && column != 3)
        return; // zou niet mogen gebeuren
    try {
        Punten p = data.get (row);
        int mark = Integer.parseInt ((String)value);
        if (p.punten[column−2] != mark) {
            p.punten[column−2] = mark;

            ...      // pas de punten ook aan in de databank

            fireTableRowsUpdated (row, row); // zie tekst !
        }
    }
    catch (NumberFormatException ex) {}
}
```

(De parameter *value* is van het type *Object* maar behoort tot de klasse *String*.)

Zoals steeds moet het model zijn views op de hoogte brengen van alle inhoudsveranderingen. *AbstractTableModel* voorziet hiervoor een aantal *fire*-methoden.

```
public void fireTableCellUpdated (int row, int column);
// De inhoud van één enkele cel werd gewijzigd
```

public void *fireTableRowsUpdated* (**int** *firstRow*, **int** *lastRow*);
// Er werden veranderingen aangebracht aan de opgegeven rijen

public void *fireTableRowsInserted* (**int** *firstRow*, **int** *lastRow*);
// Er werden nieuwe rijen ingevoegd

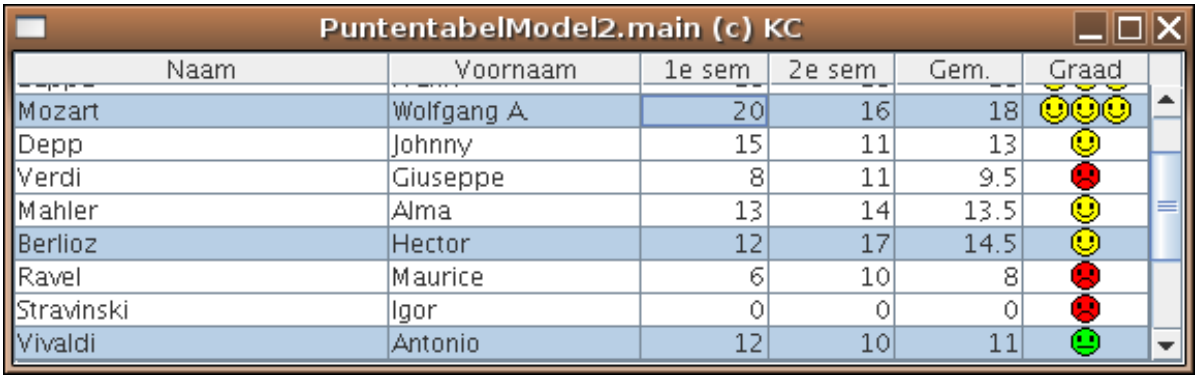
public void *fireTableRowsDeleted* (**int** *firstRow*, **int** *lastRow*);
// Er werden rijen verwijderd

public void *fireTableDataChanged* ()
// De tabel moet volledig opnieuw worden afgebeeld

public void *fireTableStructureChanged* ()
// De structuur van de tabel is gewijzigd (bijv. het aantal kolommen)

Hoewel de gebruiker — in ons voorbeeld — slechts één cel editeert en er in ons model slechts één element van de twee-dimensionale tabel *marks* wordt aangepast, roepen we in *setValueAt* niet gewoon *fireTableCellUpdated* op. Door in de plaats *fireTableRowsUpdated* te gebruiken, zijn we er zeker van dat de elementen in de laatste twee kolommen (waarvan de waarde afhangt van de cel die de gebruiker zojuist heeft geëditeerd) ook opnieuw worden afgebeeld (en berekend).

Tot slot bespreken we nog een aantal kleine 'cosmetische' aanpassingen die de tabel er doen uitzien zoals in onderstaande figuur.



Merk op dat de kolommen niet allemaal even breed zijn, dat de numerieke elementen rechts zijn gealigneerd en dat we de graad aanduiden met *smileys* in plaats van met strings.

De klasse *JTable* kan immers ook werken met cellen die niet tot de klasse *String*

behoren, maar tot een andere klasse zoals *Integer*, *Double*, *Boolean* en *Icon*. Getallen worden rechts gealigneerd, afbeeldingen worden gecentreerd in hun cel en logische waarden worden als checkboxen voorgesteld.

Het vraagt enkele aanpassingen om van deze mogelijkheid gebruik te maken. Eerst en vooral moet de methode *getColumnClass* van het model worden ingevuld zodat ze de juiste klasse teruggeeft.

```
private static final Class COLUMN_CLASSES []
        = { String.class, String.class, Integer.class,
            Integer.class, Double.class, Icon.class };

public Class getColumnClass (int column) {
    return COLUMN_CLASSES [column];
}
```

Vervolgens moet je ervoor zorgen dat *getValueAt* objecten van de juiste klassen genereert.

```
public Object getValueAt (int row, int column) {
    Punten p = data.get (row);
    switch (column) {
        case 0:
            return p.familienaam;
        case 1:
            return p.voornaam;
        case 2:
        case 3:
            return new Integer (p.punten[column − 2]);
        case 4:
            return new Double (p.getGemiddelde ());
        case 5:
            return p.getGraad ().icon;
        default:
            return null; // zou niet mogen gebeuren
    }
}
```

(Merk op dat we voor de numerieke kolommen niet de primitieve types **int** en **double** gebruiken, maar de overeenkomstige *wrapper*-klassen *Integer* en *Double*.)

Je kan de elementtypes die *JTable* ondersteunt zelf uitbreiden door een zogenaamde *cell-renderer* en *cell-editor* te implementeren en te registreren. Dit is analoog met wat we bij de spinner hebben gedaan, of wat we verder in deze tekst nog zullen bespreken voor boomdiagrammen — zie §9.3.2.

Tenslotte mag je niet vergeten dat de eerste parameter van *setValueAt* nu niet meer noodzakelijk een string hoeft te zijn.

```
public void setValueAt (Object value, int row, int column) {
    ...
    try {
        Punten p = data.get (row);
        int mark = ((Integer)value).intValue ();   // aangepast
        if (p.punten[column−2] != mark) {
            p.punten[column−2] = mark;
            ...
        }
    }
    catch (NumberFormatException ex) {}
}
```

De breedte van de kolommen stel je in met behulp van het kolommodel.

```
private static final int[] COLUMN_WIDTHS={180, 120, 60, 60, 60, 60};
...

JTable table = new JTable (model);

TableColumnModel tcm = table.getColumnModel ();
for (int i = 0; i < 6; i++)
    tcm.getColumn (i).setPreferredWidth (COLUMN_WIDTHS[i]);
```

De methode *getColumn* van het kolommodel geeft een kolombeschrijving terug (van het type *TableColumn*) waarvan je de voorkeursbreedte kunt aanpassen met *setPreferredWidth*.

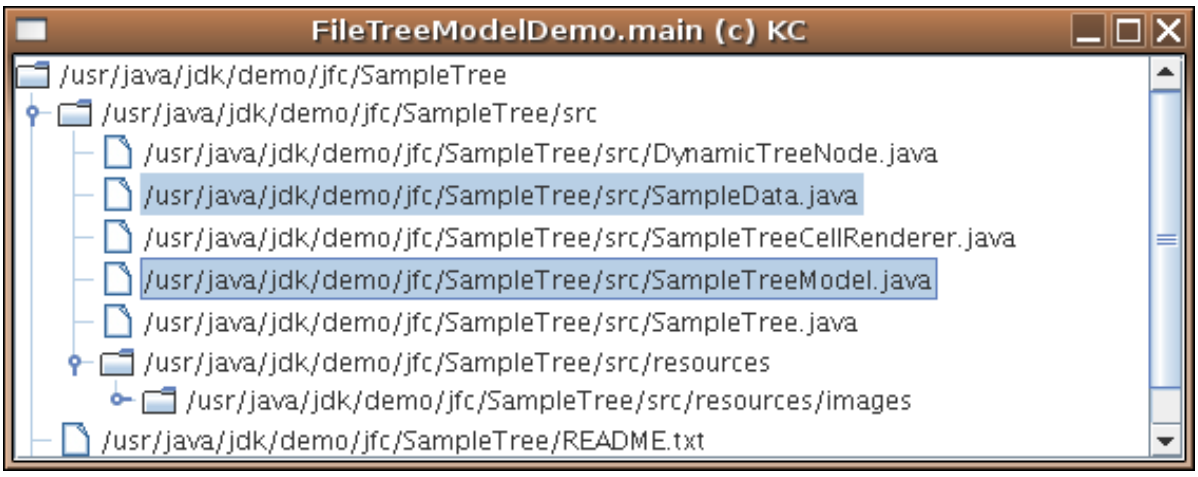
9.3. Boomdiagrammen

Net zoals bij lijsten en tabellen, past Swing het MVC-principe toe voor het voorstellen van boomdiagrammen. De view is in dit geval een object van het type *JTree*, het model is van het type *TreeModel*. We zullen dit concept illustreren aan de hand van twee uitgewerkte voorbeelden : in het eerste voorbeeld creëren we zelf een klasse die aan de interface *TreeModel* voldoet, in het tweede voorbeeld zullen we gebruik maken van de 'standaard'modelklasse *DefaultTreeModel* die onderdeel is van Swing.

9.3.1. Het model

Het gegevensmodel van een boom gedraagt zich zoals je zou verwachten na kennis te hebben gemaakt met lijst- en tabelmodellen: het model bepaalt wat de wortel is van de boom, en wat de kinderen zijn van een gegeven knoop. Wanneer er wijzigingen in het model optreden, genereert het model een overeenkomstige gebeurtenis (van het type *TreeModelEvent*) en stuurt dit door aan alle luisteraars (van het type *TreeModelListener*). Wanneer de gebruiker de inhoud van een editeerbare knoop verandert, signaleert de component dit aan het model door een gepaste methode op te roepen.

In tegenstelling tot bij lijsten en tabellen bestaat er bij boomdiagrammen geen klasse *AbstractTreeModel* die je op een eenvoudige manier tot je eigen gegevensmodel kan uitbreiden. Je hebt dus twee keuzen: ofwel implementeer je je model vanaf nul, ofwel breid je de klasse *DefaultTreeModel* uit. In deze paragraaf kiezen we voor de eerste techniek, de andere mogelijkheid komt later aan bod (zie §9.3.3).



De boom die we in dit eerste voorbeeld zullen uitwerken, is een grafische voorstelling van (een deel van) de bestandsstructuur van een computer, zoals afgebeeld in bovenstaande figuur. (Een latere versie van het programma zal zorgen voor een diagram dat iets 'leesbaarder' is — zie §9.3.2.)

Swing gebruikt verschillende afbeeldingen voor de *knopen* van een boom (een map) als voor de *blaadjes* (een blad). Knopen kunnen door de gebruiker naar believen worden geëxpandeerd of terug gereduceerd.

Voor het model van dit diagram hebben we een nieuwe klasse *FileTreeModel* geïmplementeerd. We bespreken de verschillende methoden van *TreeModel* aan de hand van dit model.

De klasse *FileTreeModel* gebruikt de klasse *File* als achterliggende boomstructuur (zie §3.4.2). Zoals je weet kan een object van de klasse *File* zowel een bestand als een directory (folder) voorstellen. In dit laatste geval is het gemakkelijk om met behulp van een methode *listFiles* de inhoud van de directory op te vragen als een tabel van *File*-objecten.

Zoals bij alle modellen uit Swing, bestaat er ook bij *TreeModel* een methode om een luisteraar toe te voegen en een methode om een luisteraar te verwijderen. We kiezen bij dit eenvoudige voorbeeld voor een onveranderlijk model — we veronderstellen dat de inhoud van een *File*-object niet verandert tijdens de loop van het programma. We hebben in dit geval dus geen luisteraars nodig.

```
public class FileTreeModel implements TreeModel {
    public void addTreeModelListener (TreeModelListener l) {}

    public void removeTreeModelListener (TreeModelListener l) {}
     ...
}
```

Elk boommodel bevat een methode *valueForPathChanged* die van buitenaf kan worden opgeroepen (bijvoorbeeld door de boomdiagramcomponent) wanneer er iets aan de boom verandert. Het model wordt verondersteld hiervan al zijn luisteraars op de hoogte te brengen.

In de praktijk gebeurt dit meestal door *fire*-methoden, die we dit keer, door het ontbreken van een *AbstractTreeModel*, zelf zouden moeten schrijven. Omdat ons model onveranderlijk is, zijn we hier gelukkig van verlost.

```
    public void valueForPathChanged (TreePath path, Object newValue) {}
```

(De betekenis van de klasse *TreePath* komt later aan bod — zie §9.3.4).

Alle andere methoden uit *TreeModel* dienen om informatie over het model op te vragen. Zo geeft de volgende methode de *wortel* van de boom terug.

```
public Object getRoot () {
    return root;
}
```

In dit geval is de wortel het *File*-object waarvan het boomdiagram moet worden opgesteld. Dit object is opgeslagen in een variabele *root* die we door de *FileTreeModel*-constructor laten invullen.

```
private File root;

public class FileTreeModel (File root) {
    this.root = root;
}
```

De methode *getChildCount* vertelt hoeveel kinderen een bepaalde knoop heeft. In ons voorbeeld kunnen we de kinderen bepalen met behulp van *listFiles*.

```
public int getChildCount (Object parent) {
    File f = (File)parent;
    return f.listFiles ().length;
}
```

De methode *getChild* bepaalt het kind met een gegeven index bij een gegeven ouder:

```
public Object getChild (Object parent, int index) {
    File f = (File)parent;
    return f.listFiles ()[index];
}
```

De methode *getIndexOfChild* doet het omgekeerde: hij berekent de index van een gegeven kind bij een gegeven ouder. Opnieuw zoeken we dit op met behulp van *listFiles*.

```
public int getIndexOfChild (Object parent, Object child) {
    File f = (File)parent;
    File[] list = f.listFiles ();
    int i=list.length-1;
    while (i >= 0 && list[i] != child)
        i--;
    return i;
}
```

Tot slot moet je ook een logische functie schrijven die aangeeft of een bepaalde knoop een blad van de boom is of niet.

```
public boolean isLeaf (Object node) {
    File f = (File)node;
    return !f.isDirectory ();
}
```

Om nu aan de hand van bovenstaand model een boomdiagram op het scherm af te beelden, volstaan enkele eenvoudige opdrachten:

```
File root = new File ("/usr/java/jdk/demo/jfc/SampleTree");
FileTreeModel model = new FileTreeModel (root);
JTree tree = new JTree (model);
 ...
window.setContentPane(new JScrollPane (tree));
```

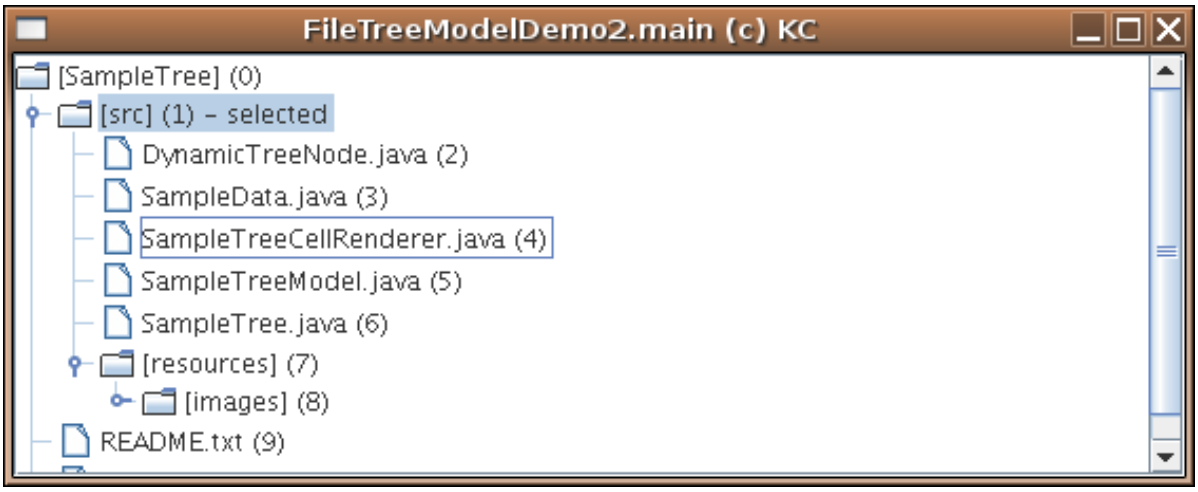
9.3.2. Gebruik van een cell-renderer

De afbeelding op de volgende bladzijde illustreert een eerste verandering die we kunnen aanbrengen in het programma.

In plaats van de volledige padnaam van een bestand af te beelden, wensen we een kortere naam af te drukken met bijkomende informatie. Foldernamen worden tussen vierkante haakjes geplaatst, na elke naam plaatsen we het rijnummer tussen ronde haakjes en bovendien voegen we de tekst 'selected' toe achter elke knoop die is geselecteerd.

Dergelijke aanpassingen aan de tekst van de knopen in het diagram kunnen op een eenvoudige manier worden bekomen door de klasse *JTree* uit te breiden tot

een eigen versie en daarin de methode *convertValueToText* te overschrijven. Deze methode wordt door het boomdiagram (of meer precies: door de cell-renderer van dit diagram — zie verder) automatisch opgeroepen telkens wanneer een knoop in dit diagram moet worden afgebeeld.



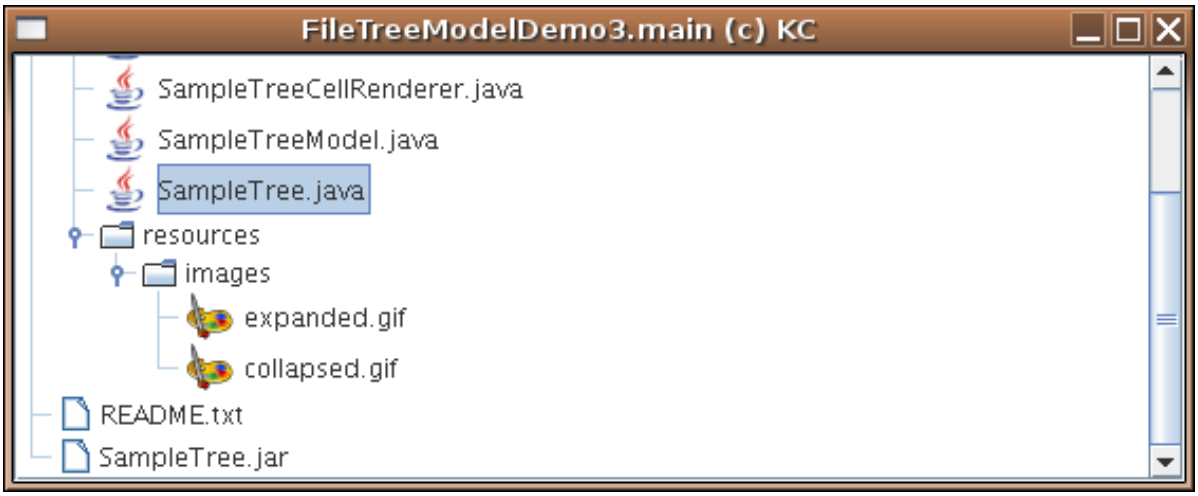

```
public class MyJTree extends JTree {
    ...
    public String convertValueToText
        (Object value,           // waarde van de knoop (in het model)
         boolean selected,       // is de knoop geselecteerd ?
         boolean expanded,       // werd de knoop geëxpandeerd ?
         boolean leaf,           // gaat het om een bladknoop ?
         int row,                // rijnummer in het diagram
         boolean hasFocus) {     // heeft de knoop de focus ?
        File file = (File) value;
        String text = file.getName();
        if (! leaf) {
            text = "[" + text + "]";
        }
        text += "  (" + row + ")";
        if (selected)
            text += " - selected";
        return text;
    }
}
```

Zoals je ziet, bieden de vele parameters van deze methode heel wat informatie over de context waarin een bepaalde knoop moet worden afgebeeld. De standaard-

versie van deze methode, zoals geïmplementeerd door *JTree* zelf, roept gewoon de *toString*-methode op van *value*. Dit verklaart waarom we in het eerdere voorbeeld de volledige padnaam te zien kregen.

Zoals reeds vermeld, gebruikt de *JTree*-component een zogenaamde *cell-renderer* om de inhoud van het boomdiagram op het scherm af te beelden. Dit is een bijzonder soort component die zorgt voor de uiteindelijke grafische voorstelling van de knopen. De cell-renderer die standaard wordt gebruikt is een soort *JLabel*. De afbeelding van dit label hangt enkel af van of de knoop al dan niet een blad is. Het opschrift van dit label is de stringvoorstelling die de boom teruggeeft als waarde van *convertValueToText*.

Als laatste aanpassing zullen we nu de afbeelding ook doen afhangen van de bestandsnaam, meer bepaald van de gebruikte extensie (*.java*, *.gif*, ...). We moeten dus een nieuwe cell-renderer ontwerpen en bij de boom registeren.



Om een nieuwe cell-renderer te definiëren vertrek je van de klasse *DefaultTreeCellRenderer*. Dit is een extensie van *JLabel* die is geoptimaliseerd om met een boomdiagram te worden gebruikt.

De enige methode die je zelf moet herschrijven is de volgende:

```
public Component getTreeCellRendererComponent
    (JTree tree,  // boom waarin de renderer wordt gebruikt
     Object value,
     boolean selected, boolean expanded,
     boolean leaf, int row, boolean hasFocus);
```

Deze routine wordt opgeroepen voor elke knoop die het diagram moet afbeelden.

De parameters hebben dezelfde betekenis als bij *convertValueToText*. Het is de bedoeling dat de methode een component teruggeeft die de knoop kan afbeelden.

Het is leerzaam om eens de implementatie van deze methode te bekijken in de klasse *DefaultTreeCellRenderer*.

Eerst wordt de tekst bepaald die moet worden afgedrukt en wordt deze tekst ingesteld als opschrift van het label. (Omdat *DefaultTreeCellRenderer* een extensie is van *JLabel* mag je **this** als een label behandelen.)

```
String stringValue =
    tree.convertValueToText (value, sel, expanded, leaf, row, hasFocus);
setText(stringValue);
```

Merk op dat dit gebeurt met behulp van de methode *convertValueToText* uit *JTree*.

In een volgende stap initialiseert de cell-renderer allerhande eigenschappen van het label, afhankelijk van de doorgegeven parameters.

```
this.hasFocus = hasFocus;
if(sel)
    setForeground (getTextSelectionColor());
else
    setForeground (getTextNonSelectionColor());
selected = sel;
```

Daarna worden de afbeeldingen gekozen en ingesteld met *setIcon*

```
if (leaf)
    setIcon(getLeafIcon());
else if (expanded)
    setIcon(getOpenIcon());
else
    setIcon(getClosedIcon());
```

En tenslotte geeft de cell-renderer zichzelf terug als gevraagde component.

```
return this;
```

Dit is wellicht één van de weinige situaties waarin hetzelfde component-object tegelijkertijd op meerdere plaatsen kan (en mag) worden afgebeeld.

In ons voorbeeldprogramma wensen we dit standaardgedrag slechts gedeeltelijk aan te passen: we willen een andere tekst en andere afbeeldingen. (En als extraatje zullen we voor 'verborgen' bestanden een grijze tekstkleur gebruiken.) We doen dit met behulp van overerving.

```
public class FileTreeCellRenderer extends DefaultTreeCellRenderer {
    ...
    public Component getTreeCellRendererComponent
            (JTree tree, Object value, boolean selected,
             boolean epanded, boolean leaf, int row,
             boolean hasFocus) {

        super.getTreeCellRendererComponent
            (tree, value, selected, expanded, leaf, row, hasFocus);
        File file = (File)value;
        String naam = file.getName ();

        if ( file .isHidden())
            setForeground (Color.LIGHT_GRAY);
        else {
            int pos = naam.lastIndexOf ('.');
            if (pos >= 0) {
                Icon icon = map.get (naam.substring (pos+1));
                if (icon != null)
                    setIcon (icon);
            }
        }
        setText (naam);
        return this;
    }
}
```

We maken gebruik van een (hash) map om de afbeelding op te zoeken die bij een bepaalde extensie hoort. Vinden we een extensie niet terug, dan wordt de standaardafbeelding voor een blad gebruikt. De map wordt bijgehouden in een attribuut van onze cell-renderer. We voorzien ook een methode *registreerExtensie* waarmee de eindgebruiker afbeeldingen met extensies kan verbinden.

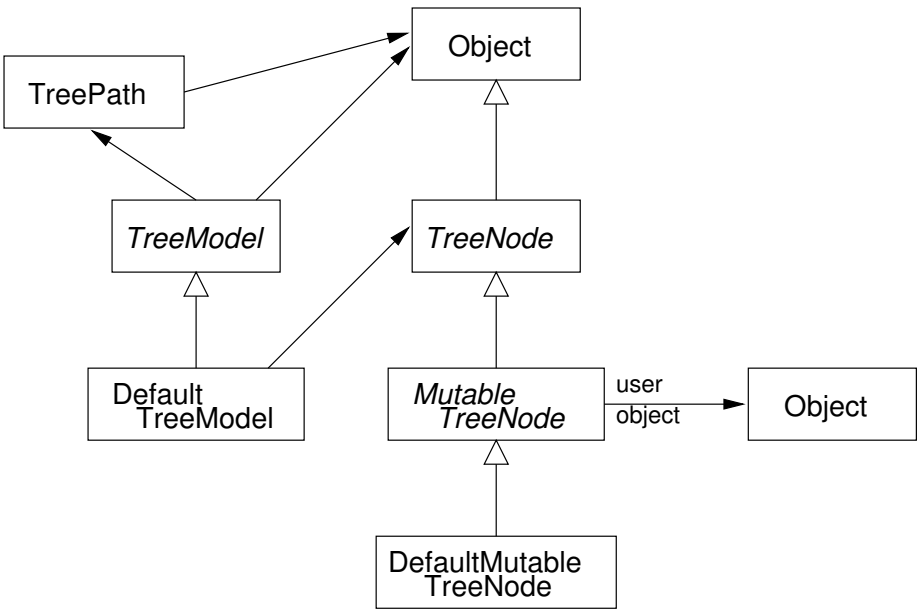
```
    public void registreerExtensie (String ext, Icon icon) {
        map.put (ext, icon);
    }
```

Er rest ons nu nog enkel deze nieuwe cell-renderer aan te maken, de juiste afbeeldingen te kiezen en de cell-renderer bij de boom te registreren:

```
FileTreeCellRenderer renderer = new FileTreeCellRenderer ();
renderer.registreerExtensie ("java", new ImageIcon (...));
Icon imIcon = new ImageIcon (...);
renderer.registreerExtensie ("png", imIcon);
renderer.registreerExtensie ("gif", imIcon);
renderer.registreerExtensie ("jpg", imIcon);

JTree tree = new JTree (new FileTreeModel (file));
tree .setCellRenderer (renderer);
```

9.3.3. Het standaardmodel voor boomdiagrammen

Het is niet altijd nodig om een eigen gegevensmodel te ontwikkelen voor een boomdiagram. Een eenvoudige implementatie zoals *FileTreeModel* is bijvoorbeeld enkel mogelijk met boommodellen waarvan de inhoud nooit mag veranderen. Het *FileTreeModel* blijkt trouwens ook niet echt praktisch te zijn, aangezien de bestanden in een directory er niet noodzakelijk in alfabetische volgorde worden opgesomd. (De oorzaak hiervan ligt bij de methode *listFiles* van *File* waarvan het resultaat ook niet alfabetisch is gerangschikt, en daar is weinig aan te doen.)



Met andere woorden, in de praktijk komt het weinig voor dat men zelf een boommodelklasse ontwikkelt, maar kiest men meestal voor de bestaande implementatieklasse *DefaultTreeModel*. Om deze klasse op de juiste manier toe te passen is het echter belangrijk ook enkele andere klassen en interfaces uit de Swingbibliotheek van dichtbij te bekijken. Het schema op de vorige bladzijde geeft een overzicht van deze klassen en hun samenhang.

DefaultTreeModel is vanzelfsprekend een implementatie van *TreeModel*, maar in tegenstelling tot *TreeModel* die geen veronderstellingen maakt over de specifieke interne voorstelling van een boom en zijn knopen, eist *DefaultTreeModel* dat knopen van de boom die hij beheert, voldoen aan de interface *TreeNode*.

De interface *TreeNode* heeft een extensie *MutableTreeNode* voor knopen waarvan de inhoud en structuur kunnen veranderen, en in de praktijk gebruikt men zelfs meestal de bibliotheekklasse *DefaultMutableTreeNode* die deze interface implementeert.

Een knoop van dit laatste type bevat een zogenaamd *user object* dat de inhoud van de knoop representeert. Het is dit object dat in het boomdiagram wordt afgebeeld. Wanneer *convertValueToText* namelijk de methode *toString* oproept voor een dergelijke knoop, dan geeft de knoop de *toString*-waarde van het user object terug.

Je geeft het user object mee als argument van de constructor van de knoop.

public *DefaultMutableTreeNode*
(*Object userObject*, **boolean** *allowsChildren*);

De parameter *allowsChildren* bepaalt of deze knoop kinderen mag hebben (indien **false** is het een blad van de boom). Je mag deze tweede parameter weglaten als hij **true** is.

Bij wijze van voorbeeld bespreken we een toepassing waarmee een boom *at run time* door de gebruiker kan worden opgebouwd.

De gebruiker kan dubbelklikken op een knoop om de tekst van die knoop te veranderen (hiervoor hoeven we niets bijzonders te doen, deze functionaliteit wordt door *JTree* vanzelf ondersteund), hij kan een geselecteerde knoop uitvegen met behulp van de knop `Verwijderen` onderaan op het paneel, hij kan een kind aan een geselecteerde knoop toevoegen of hij kan een volledig nieuwe boom aanmaken (met enkel een wortelknoop).

De centrale klasse voor deze toepassing is *DynamicTree*, een extensie van *JTree* met een model van het type *DefaultTreeModel*.


```
public class DynamicTree extends JTree {

    private DefaultTreeModel treeModel;

    public DynamicTree () {
        super (new DefaultTreeModel (null));
        this.treeModel = (DefaultTreeModel)getModel ();

        setEditable (true); // maakt knopen editeerbaar
        ...
        clear  ();
    }

    ...
}
```

In dit geval geven we het model door als argument van de constructor *JTree* en een *TreeNode* (hier toevallig **null**) als argument van de constructor van *DefaultTreeModel*. Er bestaat ook een constructor van *JTree* die meteen een *TreeNode* als parameter neemt en er automatisch een model rond bouwt.

De methode *clear* van *DynamicTree* (die onder andere wordt opgeroepen wanneer je 'Nieuw' kiest in het menu) zorgt voor een nieuwe wortelknoop:

```
public void clear () {
    TreeNode rootNode = new DefaultMutableTreeNode ("Wortel");
    treeModel.setRoot (rootNode);
    setSelectionPath(new TreePath(rootNode)); // zie §9.3.4
}
```

de methode *removeCurrent* veegt de huidige geselecteerde knoop uit

```
public void removeCurrentNode () {
    MutableTreeNode node = ...   // geselecteerde knoop
    treeModel.removeNodeFromParent (node);
}
```

en *addObject* voegt een knoop toe met een gegeven user object *child* als kind van de knoop die op dit moment is geselecteerd.

```
public void addObject (Object child) {
    MutableTreeNode parentNode = ...  // geselecteerde knoop
    MutableTreeNode childNode = new DefaultMutableTreeNode (child);

    treeModel.insertNodeInto
        (childNode, parentNode, parentNode.getChildCount ());
}
```

De methode *insertNodeInto* neemt als derde parameter de index van de plaats waar het nieuwe kind in de lijst van kinderen moet worden tussengevoegd. In dit geval voegen we het kind achteraan toe. We bekomen de index hier met andere woorden door aan de ouderknoop te vragen hoeveel kinderen hij heeft.

Als je de elektronische documentatie wat beter bekijkt, zal je merken dat *MutableTreeNode* zelf enkele methoden bevat om kinderknopen toe te voegen of te verwijderen. Het is echter beter om te vermijden deze methoden rechtstreeks te gebruiken en dergelijke aanpassingen in de plaats via het model te laten verlopen. Dit zorgt er immers voor dat alle luisteraars — de *JTree* in ons voorbeeld — automatisch van deze veranderingen op de hoogte worden gebracht.

9.3.4. Selecties

Selecties worden bij boomdiagrammen op een gelijkaardige manier verwerkt als bij lijsten en tabellen — door gebruik van een luisteraar, in dit geval van het type *TreeSelectionListener*.

In onze toepassing gebruiken we een dergelijk luisteraar om er automatisch voor te zorgen dat de juiste menukeuzes (acties) geactiveerd worden afhankelijk van wat er in de boom is geselecteerd. De volgende actie correspondeert met de menukeuze 'Toevoegen' en is actief als en slechts als er een knoop is geselecteerd.

```
public class DynamicTreeAddAction extends AbstractAction
    implements TreeSelectionListener {

    ...
    public void valueChanged(TreeSelectionEvent e) {
        setEnabled (tree.getSelectionPath() != null);
    }
}
```

De methode *getSelectionPath* van *JTree* geeft de huidige selectie terug in de vorm van een object van het type *TreePath*, of **null** wanneer er niets is geselecteerd. Bij meervoudige selectie kan je gebruik maken van *getSelectionPaths* (meervoud) die een tabel van dergelijke objecten teruggeeft.

Het *TreePath*-object correspondeert met het volledige selectie*pad*. Dit pad bevat niet alleen het geselecteerde element zelf, maar ook al zijn ouders en voorouders. Het laatste element van dit pad (het element dat het verst van de wortel ligt) bekom je dan met *getLastPathComponent*. Deze componenten zijn van het type *Object* en moeten in ons voorbeeld dus op het gepaste moment worden ge'cast' naar *TreeNode*s.

De actie die overeenkomt met de menukeuze 'Verwijderen' kijkt niet alleen of de huidige selectie bestaat, maar kan ook slechts worden geactiveerd als deze selectie niet de wortelknoop is van de boom. Dit controleren we op de volgende manier:

```
public void valueChanged(TreeSelectionEvent e) {
    TreePath path = tree.getSelectionPath();
    setEnabled(path != null && path.getPathCount() > 1);
}
```

We tellen het aantal elementen in het selectiepad. Alleen bij de wortelknoop is dit gelijk aan 1.

Tot slot drukken we nog de volledige definitie af van *removeCurrentNode*:

```
public void removeCurrentNode () {
    TreePath selection = getSelectionPath ();
    if (selection != null && selection.getPathCount () > 1)
        treeModel.removeNodeFromParent
            ((MutableTreeNode)selection.getLastPathComponent ());
}
```

De methode *addObject* gebruikt het selectiepad op een analoge manier.

10. Draden in Swing

Java is één van de weinige programmeertalen waarin je op een eenvoudige manier twee verschillende delen van je programma *tegelijkertijd* kan laten draaien. In dit hoofdstuk herhalen we kort welke faciliteiten de programmeertaal hiervoor biedt, en bespreken we een aantal extra voorzorgen die je moet nemen om er onder Swing gebruik van te maken.

10.1. Draden in Java

Om een nieuw gedeelte van een programma op te starten naast het reeds lopende programma, moet je een zogenaamde *draad* (*thread*) creëren en die lanceren. Een draad wordt voorgesteld als een object van het type *Thread*.

Het aangeven welk gedeelte van het programma er door een nieuwe draad moet worden uitgevoerd, gebeurt op het moment dat je die draad creëert. Er zijn verschillende manieren om dit te doen, waarvan we er hier slechts één bespreken. Het programmagedeelte dat als draad dient, moet steeds in één of andere klasse worden geplaatst als methode met de volgende signatuur:

public void *run*();

Bovendien moet deze klasse de interface *Runnable* implementeren, een interface die precies deze ene methode definieert.

Om dan een nieuwe draad aan te maken, roept men de constructor van *Thread* op met een object van die klasse als enige parameter. De draad wordt dan later opgestart met behulp van de methode *start*.

Thread thread = **new** *Thread* (*someRunnableObject*);
thread.start ();

Het opstarten van de draad betekent dat de corresponderende methode *run* van het object wordt uitgevoerd, tegelijkertijd met de andere draden uit de toepassing. De draad stopt vanzelf wanneer de uitvoering van *run* is afgelopen, of wanneer tijdens *run* een uitzondering wordt gegenereerd die niet wordt opgevangen. Dezelfde draad kan slechts één keer worden opgestart.

Het is niet altijd nodig om het draadobject zelf bij te houden. Het vorige fragment wordt dan ook vaak afgekort tot één enkele lijn.

```
new Thread (someRunnableObject).start ();
```

Vaak bestaat de implementatie van *run* uit slechts één of twee opdrachten. In dat geval creëert men soms een anonieme klasse als implementatie van *Runnable*:

```
new Thread (new Runnable () {
        public void run () {
            sendMailInBackground (message, receiver);
        }
}).start  ();
```

Om de leesbaarheid van je programma niet al te veel geweld aan te doen, gebruik je deze techniek best niet wanneer *run* meer dan slechts enkele opdrachten bevat.

Eenmaal een draad loopt, kan je hem in principe niet meer stoppen vanuit een andere draad. Er zit dus niets anders op dan de draad zichzelf te laten stoppen, bijvoorbeeld door hem regelmatig een bepaalde logische variabele te laten controleren die vanuit een andere draad op **false** wordt gezet.

De klasse *Thread* bevat een methode *stop* waarmee je de ene draad vanuit een andere kan laten stoppen. Men heeft echter ingezien dat deze methode een fout bevat — erger nog, dat het in principe niet mogelijk is om een *stop*-methode te schrijven die deze fout niet bevat.

Vandaar dat alle Java-documentatie nu zegt dat je deze methode niet meer mag gebruiken. Hetzelfde geldt voor de methoden *suspend* en *resume*.

10.2. Draden en Swing

Zelfs al ben je niet van plan toepassingen te schrijven die gebruik maken van afzonderlijke draden, toch dien je op de hoogte te zijn van een aantal details over de interactie tussen draden en Swing. Elke Swing-toepassing bevat immers reeds minstens twee draden: de hoofdprogrammadraad (waarin de methode *main* wordt uitgevoerd) en de gebeurtenisverwerkingsdraad (*event management thread*) die, zoals de naam aangeeft, de verschillende gebeurtenisroutines uitvoert — methoden zoals *actionPerformed* en *paintComponent*.

Het is steeds gevaarlijk om twee verschillende draden dezelfde gegevens te laten wijzigen zonder hiervoor speciale voorzieningen te treffen. Om redenen van efficiëntie neemt Swing echter geen dergelijke voorzorgen — men zegt dat Swing niet *thread safe* is — dus moet je zelf bijzonder goed oppassen. De belangrijkste regel die je hierbij in acht moet nemen is de volgende:

> *Eenmaal een component is gerealiseerd, moet alle programmacode die de component op één of andere manier wijzigt of die van bepaalde eigenschappen van die component afhankelijk is, worden uitgevoerd vanuit de gebeurtenisverwerkingsdraad.*

Een venster heet *gerealiseerd* zodra *pack* of *setVisible(***true***)* voor dat venster werd opgeroepen. Een component heet gerealiseerd wanneer hij zich op een gerealiseerd venster bevindt.

Er zijn een aantal uitzonderingen op deze hoofdregel:

- Een klein aantal methoden is toch *thread safe*. In dat geval is dit duidelijk aangegeven in de documentatie.
- Je mag in het hoofdprogramma toch nog een venster zichtbaar maken vlak nadat je *pack* voor dat venster hebt opgeroepen. Daarna voer je echter best geen GUI-methoden meer uit als onderdeel van *main*.
- Je kan *repaint* altijd veilig oproepen omdat deze methode enkel een nieuwe gebeurtenis in de wachtrij plaatst die dan op een later moment door de gebeurtenisverwerkingsdraad zal worden uitgevoerd. Hetzelfde geldt voor de methode *revalidate* die we hier in deze tekst niet hebben besproken.
- Je kan veilig nieuwe luisteraars bij een component registreren (of de registratie ongedaan maken) vanuit elke draad.

10.3. De methode *invokeLater*

De gemakkelijkste manier om te voldoen aan de hoofdregel voor draden in Swing, is om alle bewerkingen in gebeurtenisroutines te plaatsen of in methoden die erdoor worden opgeroepen. Dit is echter niet altijd mogelijk.

Omdat de gebeurtenisverwerkingsdraad ook zorgt voor het afbeelden van de vensters en de componenten en voor het verwerken van standaardmuisbewerkingen, mogen gebeurtenisroutines namelijk geen langdurige bewerkingen uitvoeren. Anders zal je toepassing niet vlot op de gebruiker blijven reageren. We bespreken twee courante situaties waarbij dit een rol speelt:

- Soms heeft de toepassing een langdurige initialisatie nodig — bijvoorbeeld wanneer afbeeldingen of andere gegevensbestanden eerst moeten worden ingeladen over het Internet. In dat geval toont de toepassing meestal een voorlopige GUI terwijl de nodige bewerkingen in een andere draad op de achtergrond worden uitgevoerd. Na afloop van de draad moet dan de GUI worden aangepast.

  De initialisatie moet in dit geval buiten de gebeurtenisverwerkingsdraad plaatsvinden, terwijl het aanpassen van de GUI erbinnen moet gebeuren.

- Vaak moet een toepassing wachten op een externe gebeurtenis. Dit kan bijvoorbeeld informatie zijn die via een netwerkverbinding wordt ontvangen, maar ook gegevens die van een bestand worden ingelezen — zeker als dit bestand zich op een andere computer bevindt.

  Ook dergelijke bewerkingen kan je beter niet in de gebeurtenisverwerkingsdraad uitvoeren.

De klasse *EventQueue* biedt een methode *invokeLater* die in bovenstaande gevallen heel nuttig is. Deze methode laat toe om een bepaald gedeelte van je programma uit te voeren als onderdeel van de gebeurtenisverwerkingsdraad.

public static void *invokeLater* (*Runnable runnable*);

Wanneer je *invokeLater* oproept, wordt het opgegeven object *runnable* achteraan de wachtrij met gebeurtenissen geplaatst. Nadat alle gewone gebeurtenissen zijn afgewerkt, wordt dan de methode *run* van *runnable* uitgevoerd in de gebeurtenisverwerkingsdraad.

We passen dit toe in het volgende voorbeeld: een paneel toont enkele afbeeldingen in een rooster van 3 kolommen. Terwijl de afbeeldingen worden ingeladen wordt echter voorlopig enkel de tekst 'Even geduld' afgebeeld. We gebruiken hierbij de volgende strategie: bij creatie van het venster plaatsen we de enkel een label met tekst 'Even geduld' op het paneel.

```
public class LoadImages extends JPanel {
    ...
    public LoadImages () {
        super (new BorderLayout ());
        setPreferredSize (new Dimension (300, 200));
        statusLabel = new JLabel ("Even geduld ...");
        statusLabel.setHorizontalAlignment (SwingConstants.CENTER);
        add (statusLabel);
    }
    ...
}
```

We voorzien echter een methode *installIcons* die een gegeven tabel van afbeeldingen als labels op het paneel installeert.

```
    public void installIcons (Icon[] icons) {
        remove (statusLabel);
        setLayout (new GridLayout (0, 3, 5, 5));
        setBorder (BorderFactory.createEmptyBorder (5,5,5,5));
        for (Icon icon: icons)
            add (new JLabel (icon));
        revalidate ();    // zorgt voor het herindelen van het venster
        repaint ();
    }
```

Het hoofdprogramma creëert eerst het paneel, laadt dan de afbeeldingen in en installeert die tenslotte op het paneel.

```
    public static void main (String[] args) {
        JFrame window = new JFrame ("...");
        final LoadImages panel = new LoadImages ();
        window.setContentPane (panel);
        window.pack ();
        window.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
        window.setVisible (true);
```

```
        // laadt de afbeeldingen in en stop ze in de tabel icons
        Icon[] icons = ...;

        // signaleer panel dat alle afbeeldingen zijn ingeladen
        EventQueue.invokeLater (new Runnable () {
                public void run () {
                    panel.installIcons (icons);
                }
        });
    }
```

We mogen de laatste opdracht niet inkorten tot enkel '*panel.installIcons (icons);*' aangezien de methode *main* niet in de gebeurtenisverwerkingsdraad wordt uitgevoerd.

10.4. De klassen *JProgressBar* en *ProgressMonitor*

Een label met de tekst 'Even geduld' is niet zo veelzeggend. Het is dan ook gebruikelijk om de voortgang van het programma op een meer dynamische manier aan te geven, bijvoorbeeld met behulp van een zogenaamde *progress-bar* (zie figuur).

Een progress-bar is een component van de klasse *JProgressBar*. Een progress-bar heeft een gehele *waarde* (*value*) die zich tussen een bepaald minimum en maximum bevindt. Deze waarde bepaalt de lengte van het gekleurde deel van de progress-bar en wordt vanuit het programma ingesteld met behulp van *setValue*.

De progress-bar uit de figuur werd geïnitialiseerd op de volgende manier.

```
progressBar = new JProgressBar (0, nrIcons);
progressBar.setStringPainted (true);
```

De constructor neemt de minimum- en maximumwaarde van de progress-bar als parameters. De tweede opdracht zorgt ervoor dat ook de percentages worden afge-

beeld. De afgebeelde tekst hoort niet bij de progress-bar, maar bij een afzonderlijk label bovenaan het paneel.

Telkens wanneer er een nieuwe afbeelding ingeladen is, verhogen we de waarde van de progress-bar met 1. Dit gebeurt met behulp van de volgende methode van onze paneelklasse.

```
public void stepProgressBar () {
    progressBar.setValue (progressBar.getValue() + 1);
}
```

Zoals voorheen mogen we deze methode enkel uitvoeren binnen de gebeurtenisverwerkingsdraad. We zullen dus opnieuw gebruik maken van *invokeLater*. Het hoofdprogramma krijgt nu de volgende structuur:

```
public static void main (String[] args) {
    JFrame window = new JFrame ("...");
    final LoadImages panel = new LoadImages ();
    ...
    Icon[] icons = new Icon [nrIcons];
    for (int i=0; i < nrIcons; i++) {
        // laadt afbeelding icons[i]
        ...
        EventQueue.invokeLater (new Runnable () {
                public void run () {
                    panel.stepProgressBar ();
                }
        });
    }

    // signaleer panel dat alle afbeeldingen zijn ingeladen
    EventQueue.invokeLater (new Runnable () {
            public void run () {
                panel.installIcons (icons);
            }
    });
}
```

Als alternatief voor een progress-bar kan je een zogenaamde *progress-monitor* oproepen: een afzonderlijk dialoogvenster met daarin een eigen progress-bar (zie de figuur op de volgende bladzijde). Dit is een object van de klasse *ProgressMonitor*.

Je creëert een progress-monitor met de volgende constructor.

public *ProgressMonitor* (*Component ouder*, *Object message*,
String note, **int** *min*, **int** *max*)

De parameter *ouder* is de oudercomponent van het dialoogvenster. De parameter *message* is het bericht dat bovenaan het dialoogvenster wordt afgebeeld ('De afbeeldingen worden ingeladen') en de gehele getallen *min* en *max* bepalen de grenzen voor de waarde van de progress-bar.

De string *note* bevat een bijkomende tekst die boven de progress-bar in de dialoog wordt afgedrukt. Deze tekst geeft een extra aanduiding van hoever het programma is gevorderd. In ons voorbeeld schrijven we hier de naam van de afbeelding die het laatst werd ingeladen ('broccoli').

Je kan deze tekst tijdens de loop van het programma veranderen met behulp van de methode *setNote* — in de gebeurtenisverwerkingsdraad wel te verstaan. In plaats van *setValue* gebruik je nu *setProgress* om een nieuwe waarde voor de progress-bar in te stellen.

Een progress-monitor verschijnt niet onmiddellijk op het scherm, maar enkel nadat de taak langer dan een zekere tijd heeft geduurd. Na een halve seconde probeert de monitor te voorspellen of de taak nog minstens twee seconden nodig heeft en alleen in dat geval verschijnt de dialoog op het scherm. Je kan beide tijdsintervallen instellen met *setMillisToDecideToPopup* en *setMillisToPopup*.

Het dialoogvenster bevat ook een 'Cancel'-knop. Hiermee geeft de gebruiker aan dat hij de taak wil afbreken. Vanuit het programma ga je na of deze knop werd ingedrukt met de methode *isCanceled*. De gebruiker kan het venster ook gewoon sluiten met de sluitknop op de titelbalk, met hetzelfde effect.

A. Archiveren in JAR-bestanden

De Java-omgeving laat je toe om verschillende bestanden en directories te comprimeren en samen te bundelen tot één zogenaamd JAR-bestand (*Java ARchive*). Dit gebeurt met behulp van het programma **jar**. Een dergelijk archief biedt bepaalde mogelijkheden:

- Je kan de *.class*-bestanden van één of meer pakketten bundelen tot een archief dat je dan aan het *class path* toevoegt bij het uitvoeren van een programma dat dit pakket gebruikt. Dit is vaak eenvoudiger dan de overeenkomstige (geëxpandeerde) directorystructuur aan je class path toe te voegen. Het is ook gemakkelijker om één enkel archief tussen computers te transporteren dan de afzonderlijke bestanden.
- Je kan een archief aanmaken met daarin een programmabestand en alle klassen die nodig zijn om dit programma uit te voeren. Java laat toe om het programma uit te voeren zonder daarom eerst het archief te moeten uitpakken. Het archief kan ook allerhande *bronnen* bevatten die het programma nodig heeft: afbeeldingen, configuratiebestanden enz.

Merk op dat je het programma **jar** in principe voor alle soorten bestanden kan gebruiken, ook al hebben ze niet direct iets met Java te maken. Het JAR-formaat heeft het bijkomende voordeel dat het onafhankelijk is van het besturingssysteem waarop het wordt gebruikt.

Om een JAR-bestand aan te maken heb je drie verschillende gegevens nodig:

- Eventueel een zogenaamd *manifest* (zie verder).
- De naam van het JAR-bestand zelf. Dit draagt meestal de extensie *.jar*, maar dit is niet verplicht.
- De bestanden en directories die je wil archiveren.

Om bijvoorbeeld alle *.class*-bestanden uit de huidige directory in een archief met de naam *archief.jar* te stoppen, schrijf je

```
jar cf archief.jar *.class
```

Om dan later een kopie van de bestanden uit het archief te halen, doe je

```
jar xf archief.jar
```

De letters `c` en `x` zijn optieletters die aangeven wat het **jar**-programma moet doen: een archief aanmaken (`c`) of bestanden uit het archief halen (`x`). De optieletter `f` duidt aan dat de tweede opdrachtlijnparameter de naam is van het archiefbestand. Als deze ontbreekt wordt het archief uitgeschreven naar het standaard uitvoerkanaal.

Je kan ook een extra optie `v` gebruiken als je wil dat **jar** uitvoerig bericht over wat hij aan het doen is. In plaats van `x` gebruik je `t` om een overzicht te krijgen van de inhoud van het archief zonder het uit te pakken.(Voor wie de Unix-opdracht `tar` kent, zijn deze optieletters geen verrassing.)

Elk archief bevat een zogenaamd *manifest*, een bestand met extra informatie over wat het archief bevat. Dit bestand heeft de naam *MANIFEST.MF* en wordt gearchiveerd alsof het zich in de directory *META-INF* bevindt. Wanneer je de bestanden uit een archief haalt, zal je zien dat ook deze directory en dit manifest worden gecreëerd.

Je kan het manifest onder andere gebruiken om een gearchiveerd programma rechtstreeks met de volgende opdracht te kunnen uitvoeren:

```
java –jar archief.jar
```

Hiertoe moet het manifest een lijn bevatten van de volgende vorm:

```
Main-Class: klassennaam
```

waarbij je de naam opgeeft van de naam van de klasse (in het archief) die de *main*-methode van je toepassing bevat.

Om dergelijke lijnen aan het manifest toe te voegen, stop je ze in een afzonderlijk bestand (bijvoorbeeld met de naam *manifest*) en gebruik je de `m`-optie van de opdracht **jar** op de volgende manier:

```
jar cfm archief.jar manifest *.class
```

of ook nog

```
jar cmf manifest archief.jar *.class
```

Merk op dat de volgorde van de opdrachtlijnparameters moet overeenkomen met de volgorde van de `f`- en `m`-opties.

Je kan de manifestlijnen ook achteraf nog toevoegen met behulp van de `u`-optie:

```
jar cf archief.jar *.class
jar ufm archief.jar manifest
```

Neem als voorbeeld een programma 'solitaire' dat uit een aantal klassen bestaat (waarvan de *.java*-bestanden zich in de huidige directory bevinden) en een aantal GIF-afbeeldingen nodig heeft uit de subdirectory *images*. De programmaklasse heet *SolPanel*.

Je kan dan de volgende (UNIX)-opdrachten gebruiken om dit programma in een uitvoerbare vorm te archiveren:

```
javac *.java
echo 'Main-Class: SolPanel' > manifest
jar cfm solitaire.jar manifest *.class images/*.gif
rm manifest *.class    # verwijder overtollige bestanden
```

Merk op dat de GIF-bestanden die in het archief zijn opgeslagen nog steeds door *SolPanel* kunnen worden opgehaald, op dezelfde manier als bij bronnen die zich in een directory bevinden (zie 1.5).

B. Databanken en SQL

Een relationele databank groepeert zijn gegevens in zogenaamde *tabellen*. Elke *rij* uit die tabel bevat een lijst van waarden die bij elkaar horen. De *kolommen* van de tabel geven betekenis aan die elementen.

De tabel *studenten* die we hieronder afbeelden, bevat bijvoorbeeld een aantal gegevens uit een studentendatabank:

nummer	*familienaam*	*voornaam*
20010356	Janssens	Griet
20001324	Peters	Bram
20001425	Vander Ven	Hendrik
19990205	Janssens	Jan
20001222	Huybrechts	Bram

Wanneer hij een tabel in een databank creëert, legt de databankbeheerder vast wat de naam is van die tabel, hoeveel kolommen de tabel bevat, wat de namen zijn van die kolommen (in dit voorbeeld, *nummer*, *familienaam* en *voornaam*) en wat de *types* zijn van die kolommen (getal, tekst, tekst). De rijen zelf worden tijdens de toepassing ingevuld, veranderd of verwijderd.

Merk op dat 'tabel' een abstracte notie is. Ook al drukken we een tabel meestal in tabelvorm af, betekent dit daarom niet noodzakelijk dat ze ook als 2-dimensionale structuur in de databank is opgeslagen. We hoeven ons echter niet druk te maken over hoe een databank er 'van binnen' uitziet.

Moderne databanksoftware laat toe om gegevens op te zoeken, in te voegen, te verwijderen en te veranderen met behulp van de computertaal *SQL*.

De volgende SQL-opdracht vraagt bijvoorbeeld de studentennummers en voornamen van alle studenten die Janssens heten:

```
SELECT nummer, voornaam FROM studenten
  WHERE familienaam='Janssens';
```

Het resultaat van deze zoekopdracht is de volgende lijst van gegevens (opnieuw voorgesteld als een tabel):

nummer	*voornaam*
19990205	Jan
20010356	Griet

Merk op dat de volgorde van de rijen in het resultaat niet a priori vastligt, en wellicht afhangt van de interne structuur van de databank op dat moment. Je kan zelf een volgorde vastleggen met behulp van de `ORDER`-clausule:

```
SELECT nummer, voornaam FROM studenten
  WHERE familienaam='Janssens'
  ORDER BY voornaam;
```

In dit geval zal de uitvoer alfabetisch gerangschikt zijn volgens voornaam.

Om de volledige tabel in één keer op te vragen, gebruik je de volgende eenvoudige opdracht:

```
SELECT * FROM studenten;
```

Een `SELECT`-opdracht kan ook gegevens opvragen die verspreid zijn over verschillende tabellen. Bekijk bijvoorbeeld onderstaande tabellen *studiejaren* en *benamingen*:

nummer	*jaarcode*
19990205	WIJS2MAST
20001324	INFO1BACH
20010356	INFO2BACH
20001425	INFO2BACH

code	*titel*
INFO1BACH	1e jaar, bachelor informatica
INFO2BACH	2e jaar, bachelor informatica
...	...
WIJS2MAST	2e jaar, master wijsbegeerte

Onderstaande opdracht drukt de titel af van het studiejaar waartoe student met studentennummer 20001324 behoort:

```
SELECT titel FROM studiejaren, benamingen
  WHERE nummer=20001324 AND jaarcode=code;
```

De volgende opdracht toont een lijst van alle studenten en de overeenkomstige studiejaartitels

```
SELECT voornaam,familienaam,titel
  FROM studenten,studiejaren,benamingen
  WHERE studenten.nummer=studiejaren.nummer
        AND jaarcode=code;
```

Merk op hoe je onderscheid maakt tussen twee kolommen met dezelfde naam (*nummer*) in twee verschillende tabellen (*studenten* en *studiejaren*).

Je kan met een `SELECT`-opdracht ook aantallen rijen tellen. Onderstaande opdracht vraagt aan de gegevensbank hoeveel studenten er Janssens heten.

```
SELECT count(*)
  FROM studenten
  WHERE familienaam='Janssens';
```

Het resultaat hiervan is een tabel met één rij en één kolom die enkel het gevraagde aantal bevat (in ons voorbeeld: 2).

Tot slot nog, zonder uitleg, een voorbeeld van een meer gesofisticeerd gebruik van `SELECT`. Zo druk je bijvoorbeeld voor elk studiejaar in de opleiding informatica het aantal studenten af:

```
SELECT jaarcode, count(nummer)
  FROM studiejaren
  WHERE jaarcode LIKE 'INFO%'
  GROUP BY jaarcode;
```

De `LIKE`-operator vergelijkt een string met een patroon dat jokertekens kan bevatten: een underscore is een joker voor één letterteken, een percent is een joker voor nul of meer opeenvolgende tekens.

De `INSERT`-opdracht dient voor het toevoegen van nieuwe rijen aan een tabel. Bijvoorbeeld:

```
INSERT INTO studenten
  VALUES (20001555, 'Rogiers', 'Pierre');
```

Bestaande waarden kan je veranderen met `UPDATE`:

```
UPDATE studenten
  SET voornaam='Piet' WHERE nummer=20001555;
```

Bovenstaande opdracht verandert de voornaam van één bepaalde student. Je kan `UPDATE`-opdrachten ook gebruiken om meerdere rijen tegelijkertijd te veranderen. De volgende twee opdrachten veranderen bijvoorbeeld overal de studiejaarcode 'INFO2BACH' in 'INFOBACH2':

```
UPDATE studiejaren
  SET jaarcode='INFOBACH2' WHERE jaarcode='INFO2BACH';
UPDATE benamingen
  SET code='INFOBACH2' WHERE code='INFO2BACH';
```

SQL laat ook allerhande rekenkundige bewerkingen toe. Zo kunnen we bijvoorbeeld de studentennummers van iedereen met 1 verhogen op de volgende manier:

```
UPDATE studenten SET nummer=nummer+1;
```

Omdat de `WHERE` ontbreekt, wordt deze verandering uitgevoerd op elke rij van de tabel.

De `DELETE`-opdracht dient om rijen te verwijderen.

```
DELETE FROM studenten
    WHERE nummer='20001555';
```

Nog een voorbeeld:

```
DELETE FROM studiejaren
    WHERE jaarcode LIKE 'WIJS%';
```

Tot slot merken we nog op dat SQL ook kan gebruikt worden om de tabellen zelf te manipuleren — tabellen aan te maken of te vernietigen, kolommen bij te voegen of te hernoemen — maar dit valt buiten het bestek van deze korte inleiding.