Fonts unter Java

Gesetzte Typen

Ein genauer Blick auf das Thema Fonts in Java-Programmen lohnt sich, denn ein umfangreiches API kann Schriften auf viele Arten verändern. Grundlagenwissen über Schriftfamilien und -technologien ist hilfreich, um dieses Potenzial besser auszuschöpfen. Bernhard Bablok

Ein beliebtes Thema in Java-Mailinglisten war früher das „Zapf Dingbats"-Problem: Jede GUI-Anwendung spuckte beim Laden eine Reihe von Meldungen über nicht gefundene Fonts aus. Die Anwendung lief dann zwar ohne sichtbare Schwierigkeiten, aber die Sache war unschön und erinnerte daran, dass Linux eben keine von Sun aktiv unterstützte Plattform war.

Startprobleme

Die Zeiten haben sich zum Glück geändert und die meisten Anwendungen laufen, ohne dass man sich bei der Programmierung Gedanken über Fonts machen müsste. Java verwendet andere Standardfonts und bringt die wichtigen auch gleich selbst mit. Zudem benutzt die Java-Runtime seit dem JDK 1.3 das Font-Rendering des Betriebssystems für AWT und Swing. Es gibt aber immer wieder Fälle, bei denen man nicht darum herum kommt, sich mit dem Thema Fonts zu befassen. Internationalisierte Anwendungen oder die Einführung des Euro sind dafür Beispiele. Dieser Coffee-Shop erklärt die Philosophie hinter dem Java-Fontsystem und beschreibt die wichtigen APIs. Vor den technischen Details steht aber noch die Klärung wichtiger Begriffe.

Bei Schriften unterscheidet man Schriftarten und Schriftfamilien. Die Schriften einer Familie besitzen ein einheitliches typografisches Design. Eine bekannte Schriftfamilie umfasst beispielsweise die „Computer Modern"-Schriften des TeX-Systems von Donald Knuth, beschrieben in [1]. Das Java-SDK liefert Schriften der Lucida-Familie mit.

Eine Schriftart besitzt charakteristische Merkmale über unterschiedliche Schriftfamilien hinweg, zum Beispiel die Größenverhältnisse von Ober- und Unterlängen oder die Ausprägung der Serifen (das sind die Häkchen an den Buchstaben). Schriftschnitte nennt man spezielle Ausprägungen einer Schrift, etwa normale (regular), fette (bold), kursive (italic), schräg gestellte (slanted) oder Kapitälchen (Caps).

Begriffsverwirrung

Leider werden diese Begriffe nicht immer einheitlich verwendet – auch nicht in der JDK-Dokumentation. Außerdem gibt es Unterschiede bei den Benennungen, zum Beispiel bei Typewriter. Bei Donald Knuth heißt so eine Schriftart, im JDK dagegen eine Schriftfamilie (Lucida Sans Typewriter).

Die Schriftgröße wird in Punkt (pt) gemessen. Die tatsächlichen Größen sind selbst im amerikanischem Maßsystem krumm (eine Folge des 1959 veränderten Umrechnungsfaktors zwischen Inch und Zentimeter, siehe [1]): 72,27 pt kommen auf 1 Zoll (Inch), 1 pt hat also ungefähr 0,351 mm. Schriftgrößen ab 10 pt sind

normal lesbar, kleinere sind nützlich für Indizes oder Exponenten.

Vereinfacht ausgedrückt geben Fonts an, wie ein Buchstabe zu malen ist. Dabei gibt es eine n:m-Beziehung zwischen Buchstabe (n) und Zeichen (Glyph): Der Buchstabe ä besteht aus drei Zeichen (dem a und zwei Punkten). Die beiden Buchstaben fi werden bei manchen Schriftarten zu einer Ligatur, also zu einem Zeichen zusammengezogen: fi. Je nach Kontext kann ein und derselbe Buchstabe durch verschiedene Zeichen repräsentiert werden. Das betrifft besonders nicht-europäische Schriften.

Bevor ein Text am Bildschirm erscheint, sind also mehrere Schritte notwendig: von der Unicode-Repräsentation über den Font selbst bis zum Layout der Zeichen. Mit Java kann der Programmierer alle Schritte selber kontrollieren, muss sich aber im Allgemeinen darüber kaum Gedanken machen, da die Standardvorgaben des JDK normale Anwendungsfälle abdecken.

Logische und physikalische Schriften

Bei Java gibt es zusätzlich die Unterscheidung zwischen logischen und physikalischen Fonts. Letztere sind die tatsächlichen Schriftbibliotheken mit ihren Zuordnungstabellen zwischen Buchstaben und Zeichen. Auch hier gibt es Unterschiede in den Font-Technologien, verbreitet sind Truetype und Postscript. Alle Java-2-Implementationen müssen Truetype unterstützen, alle weiteren Fonttechnologien sind optional. Physikalische Fonts gibt es auf einem System meist nur für eine begrenzte Anzahl von Schriftsystemen, etwa für die europäischen Sprachen oder Japanisch.

Java kennt fünf logische Schriftfamilien: Serif, Sans Serif, Monospaced, Dialog und Dialog Input. Die Java-Runtime ordnet logische Fonts physikalischen Fonts in einer 1:n-Beziehung zu. Das erlaubt die Unterstützung verschiedenster Zeichensysteme, die Java-Runtime durchsucht für die Darstellung eines Unicode-Zeichens der Reihe nach die definierten physikalischen Fonts. Wie sie die einzelnen Schriften zuordnet, hängt dabei von der Java-Implementation sowie von der Lokalisierung ab.

Ein Programmierer hat die Wahl zwischen logischen und physikalischen Fonts. Beide Varianten besitzen Vor- und Nachteile. Logische Fonts existieren immer, mit ihnen dargestellter Text kann zumindest in der Sprache des zugrunde liegenden Betriebssystems ausgegeben werden. Da die Zuordnung zu dem physikalischen Font aber je nach Implementation unterschiedlich ist, sieht das Ergebnis nicht notwendigerweise gleich aus. Ein Text könnte also unter Linux anders umbrochen werden als unter Windows. Oder das IBM-JDK zeigt einen Text anders an als das Sun-JDK, obwohl das Programm auf demselben Rechner und Betriebssystem läuft.

Physikalische Fonts vermeiden diese Probleme, sind aber schwieriger zu programmieren. Die mitgelieferten Lucida-Fonts unterstützen nicht alle Sprachen (genauer: Unicode-Zeichen), es fehlen insbesondere Chinesisch, Japanisch und Koreanisch. Eine Alternative sind eigene physikalische Fonts, die mit der Anwendung ausgeliefert werden müssen. Hier sind Lizenzfragen zu beachten.

Listing 1a: »font.properties« des Sun-JDK in Linux

```
01 serif.0=-b&h-lucidabright-medium-r-normal--*-%d-*-*-p-*-iso8859-1
02 serif.1=--standard symbols l-medium-r-normal--*-%d-*-*-p-*-urw-fontspecific
03
04 serif.italic.0=-b&h-lucidabright-medium-i-normal--*-%d-*-*-p-*-iso8859-1
05 serif.italic.1=--standard symbols l-medium-r-normal--*-%d-*-*-p-*-urw-fontspecific
06
07 serif.bold.0=-b&h-lucidabright-demibold-r-normal--*-%d-*-*-p-*-iso8859-1
08 serif.bold.1=--standard symbols l-medium-r-normal--*-%d-*-*-p-*-urw-fontspecific
```

Listing 1b: »font.properties« des IBM-JDK in Linux

```
01 serif.0=-jdk-lucidabright-medium-r-normal--*-%d-75-75-p-*-iso8859-1
02 serif.1=-jdk-lucidabright-medium-r-normal--*-%d-75-75-p-*-iso8859-15
03 serif.2=-monotype-timesnewromanwt-medium-r-normal--*-%d-75-75-p-*-microsoft-symbol
04
05 serif.italic.0=-jdk-lucidabright-medium-i-normal--*-%d-75-75-p-*-iso8859-1
06 serif.italic.1=-jdk-lucidabright-medium-i-normal--*-%d-75-75-p-*-iso8859-15
07 serif.italic.2=-monotype-timesnewromanwt-medium-r-normal--*-%d-75-75-p-*-microsoft-symbol
08
09 serif.bold.0=-jdk-lucidabright-medium-r-normal--*-%d-75-75-p-*-iso8859-1
10 serif.bold.1=-jdk-lucidabright-bold-r-normal--*-%d-75-75-p-*-iso8859-15
11 serif.bold.2=-monotype-timesnewromanwt-medium-r-normal--*-%d-75-75-p-*-microsoft-symbol
```

Listing 1c: »font.properties« des Sun-JDK in Windows

```
01 serif.0=Times New Roman,ANSI_CHARSET
02 serif.1=WingDings,SYMBOL_CHARSET
03 serif.2=Symbol,SYMBOL_CHARSET
04
05 serif.italic.0=Times New Roman Italic,ANSI_CHARSET
06 serif.italic.1=WingDings,SYMBOL_CHARSET
07 serif.italic.2=Symbol,SYMBOL_CHARSET
08
09 serif.bold.0=Times New Roman Bold,ANSI_CHARSET
10 serif.bold.1=WingDings,SYMBOL_CHARSET
11 serif.bold.2=Symbol,SYMBOL_CHARSET
```

Fontsuche

Die wichtige Frage ist letztlich: Wo und wie findet die Java-Runtime ihre Fonts? Unter Java 1.1 war diese Frage einfach zu beantworten. Die Datei »$JRE_HOME /lib/font.properties« (und ihre lokalisierten Varianten) definierte das Mapping von Fontnamen, wie man sie innerhalb von Java verwendete, zu X-Window-Systemfonts. Wer zusätzliche Fonts wollte, musste diese Datei editieren. Verständlich, dass dies eine Fülle von Problemen verursachte, denn kaum ein normaler ▶

Listing 2: Ausschnitt aus »DisplayFonts.java«

```
01 import java.awt.*;
02 import java.awt.event.*;
03 import javax.swing.*;
04
05 ...
06
07    public void createGUI() {
08       JPanel panel = new JPanel();
09       panel.setLayout(new GridLayout(0,4));
10       panel.setBackground(Color.white);
11
12       String fontFamilies[] = (new
             ShowFonts()).getFonts12();
13       for (int i=0;i<fontFamilies.length;++i) {
14          panel.add(getLabel(fontFamilies[i],
               "plain",false));
15          panel.add(getLabel(fontFamilies[i],
               "bold",false));
16          panel.add(getLabel(fontFamilies[i],
               "italic",false));
17          panel.add(getLabel(fontFamilies[i],
               "italic",true));
18       }
19       getContentPane().add(new
             JScrollPane(panel));
20    }
21
22
23    //   JLabel mit Attributen erzeugen.
24
25    private JLabel getLabel(String fontFamily,
          String face, boolean antiAlias) {
26       String fontName = fontFamily + "-" +
             face + "-" + FONT_SIZE;
27       JLabel label;
28       if (antiAlias)
29         label = new AntiAliasLabel(fontName);
30       else
31         label = new JLabel(fontName);
32       label.setFont(Font.decode(fontName));
33       label.setForeground(FONT_COLOR);
34       return label;
35    }
36
37    public static void main(String[] args) {
38       DisplayFonts df = new DisplayFonts();
39
40       df.addWindowListener(new
            WindowAdapter() {
41          public void
              windowClosing(WindowEvent event) {
42             System.exit(0);
43          }
44       });
45
46       df.createGUI();
47       df.setSize(new Dimension(800,600));
48       df.setTitle("DisplayFonts");
49       df.show();
50    }
51
52    private static class AntiAliasLabel
          extends JLabel {
53       AntiAliasLabel(String fontName) {
54          super(fontName);
55       }
56       public void paintComponent
           (Graphics g) {
57          Graphics2D g2 = (Graphics2D) g;
58          ((Graphics2D) g).addRenderingHints(
59             new RenderingHints
          (RenderingHints.KEY_TEXT_ANTIALIASING,
60                RenderingHints.
                  VALUE_TEXT_ANTIALIAS_ON));
61          super.paintComponent(g);
62       }
63    }
64 }
```

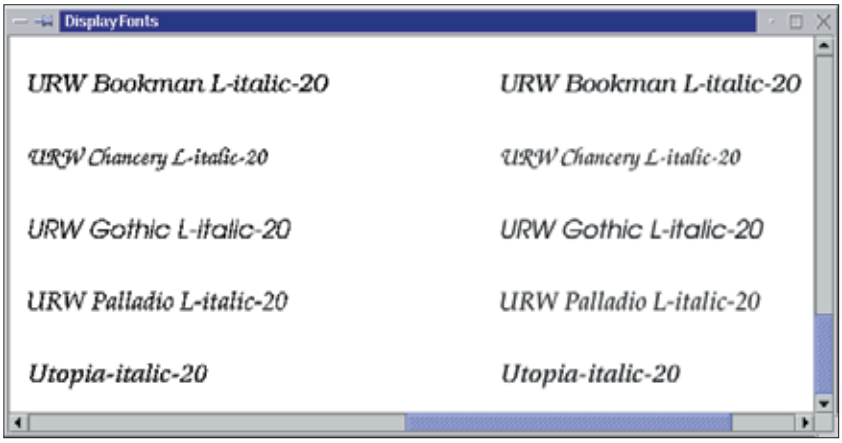

Abbildung 1: In der linken Hälfte des Fensters sind Fonts mit, rechts ohne Antialiasing zu sehen.

Anwender war wirklich in der Lage (oder willig), Fonts unter X zu installieren und dann in einer Java-Systemdatei die richtigen Pfade einzutragen. Unter Java 2 ist die Sache einfacher und gleichzeitig komplizierter. Einfacher, weil man normalerweise die »font.properties« in Ruhe lassen kann, und komplizierter, weil es jetzt mehrere Mechanismen gibt, um Fonts zu verwenden.

Font-Pfade

Die Datei »font.properties« ist erhalten geblieben. Ihre Rolle beschränkt sich im Wesentlichen auf die Zuordnung zwischen logischen und physikalischen Fonts. Die Listings 1a bis 1c zeigen Ausschnitte aus drei »font.properties«-Dateien (Sun-JDK, IBM-JDK, Sun-JDK unter Windows). Sie demonstrieren, dass jede Java-Implementation die Schriftarten anders zuordnet. Beim Sun-JDK sucht die Java-Runtime einen Buchstaben der logischen Serifenschrift an zwei Stellen, beim IBM-JDK und unter Windows an drei Stellen.

Java 2 kann direkt mit den Fontdateien umgehen, muss diese aber erst mal finden. Hierzu gibt es zwei Möglichkeiten. Der eingebaute Suchpfad findet Fonts in »$JRE_HOME/lib/font« sowie in »/usr/X11R6/lib/X11/fonts/Type1« und »/usr/X11R6/lib/X11/fonts/TrueType«. Alternativ setzt man den Suchpfad explizit mittels »JAVA_FONTS = ...« – dies überschreibt den eingebauten Pfad.

Das Programm in Listing 3 gibt alle gefundenen Font-Familien aus. Die Methode »getFonts11()« (Zeilen 7 bis 9) verwendet dafür ein deprecated API (»getFontList()«), das auch nur die in der »font.properties« definierten logischen Fontfamilien findet – außer, wenn es sich noch um eine »font.properties« von Java 1.1, wie oben beschrieben, handelt. Mit Java 2 zeigt die Methode »getAvailableFontFamilyNames()« die installierten Fontfamilien an.

Zum eigenen Experimentieren bietet sich die Umgebungsvariable »JAVA_FONTS« an. Potenzielle Kandidaten dafür sind alle Verzeichnisse mit einer Datei namens »fonts.dir«, die das Programm »locate« auf der Festplatte findet.

Listing 3: Ausschnitt aus »ShowFonts.java«

```
01 import java.awt.*;
02
03 ...
04
05    //   Dump fonts using JDK-1.1 API
06
07    public String[] getFonts11() {
08       return Toolkit.getDefaultToolkit()
            .getFontList();
09    }
10
11    //   Dump fonts using JDK-2 API
12
13
14    public String[] getFonts12() {
15       return GraphicsEnvironment.
16         getLocalGraphicsEnvironment()
              .getAvailableFontFamilyNames();
17    }
18
19    public static main(String[] args) {
20       int i;
21       ShowFonts sf = new ShowFonts();
22       System.out.println("------ 1.1 method:
            getFontList() (deprecated) --------");
23       String fontNames[] = sf.getFonts11();
24       for (i=0; i<fontNames.length; ++i)
25          System.out.println(fontNames[i]);
26
27       System.out.println("------ 1.2 method:
            getAvailableFontFamilyNames() -----");
28       String fontFamilies[] = sf.getFonts12();
29       for (i=0; i<fontFamilies.length; ++i)
30          System.out.println(fontFamilies[i]);
31
32       System.exit(0);
33    }
```

Java-Klassen

Wichtigste Fontklasse im JDK ist die »Font«-Klasse im Package »java.awt«. Verwirrenderweise gibt es seit Java 1.2 aber noch ein Package »java.awt.font«. Die Fontklasse kapselt eine physikalische Schriftart. Fontobjekte erzeugt man über Konstruktoren, die entsprechende Attribute erhalten (Name, Stil, Größe), oder über eine Reihe statischer Factory-Methoden wie »getFont()« oder »createFont()« (Zeile 32 in Listing 2).

Die Fontklasse bietet auch Methoden, um die Eigenschaften eines Fonts abzufragen, beispielsweise »getSize(), getFamiliy()« oder »getFontName()«. Nützlich ist es auch, zu einem gegebenen Font einen neuen Font mit etwas anderen Eigenschaften zu erzeugen. So gibt »deriveFont(12.0)« den 12 Punkt großen Verwandten des aktuellen Fonts zurück, genauso wie »deriveFont (Font.BOLD)« die fette Variante liefert.

Die Javadoc-Dokumentation zur Fontklasse ist sehr lesenswert, sie enthält eine Reihe von Links zu weiteren nützlichen Dokumenten. Die Klassen des »java.awt.font«-Packages sind für komplexere Aufgaben gedacht. Ohne hier auf Einzelheiten einzugehen, sei nur ein Beispiel genannt: Es gibt die Klasse »LineBreakMeasurer«, mit deren Hilfe man einen Text sauber umbrechen kann.

Schriften im GUI

Generell lautet die Empfehlung hierzu: Zu viele manuelle Feineinstellungen schaden der Portabilität der Java-Programme. Optimierungen von Menüs, Labels und so weiter sind immer nur auf einem einzigen Rechner mit immer der-

selben Auflösung optimal, deshalb lohnt sich die Mühe der Handarbeit nicht. Interessant werden Fonts jedoch dann, wenn es nicht um Standard-Widgets geht, sondern um einen eigenen Canvas. Beispiele dafür sind: das Anlegen einer Legende für eine Grafik oder ein eigener Wysiwyg-Editor.

Eine solche Anwendung zeigt Listing 2: Ein Programm, das alle Fontfamilien in den Standardausführungen Plain, Bold und Italic darstellt (siehe Abbildung 1). Das Wichtige passiert in der »getLabel()«-Methode, sie erzeugt ein JLabel und setzt den Font mittels »Font.decode()«. Der Fontname ergibt sich aus Familienname, Art und Größe.

Das Programm erzeugt die Italic-Ausführung in zwei Varianten: einmal normal, einmal mit Antialiasing. Der Trick besteht in einer Subklasse von JLabel, die die »paintComponent()«-Methode überschreibt und entsprechende Hinweise für den Renderer setzt, so genannte »RenderingHints«, (siehe Zeilen 54 bis 62 im Listing 2). In Abbildung 1 ist der Effekt deutlich zu sehen.

Jenseits von Widgets

Das obige Beispiel verwendet Standard-Widgets der GUI-Klassen Swing (beispielsweise JPanel oder JLabel). Für alle diese Komponenten setzt die »setFont()«-Methode den Font. Interessanter und anspruchsvoller ist aber der Fall, dass eine Grafik beschriftet werden soll. Denn hier muss man sich selbst darum kümmern, wo und wie die Java-Runtime den Text darstellt.

Listing 4 zeigt das Prinzip. Jede grafische Komponente besitzt eine »paint()«-Methode mit einem »Graphics«-Objekt als Argument. Texte lassen sich dann mit der »Graphics.drawString()«-Methode ausgeben, den Font setzt die Methode »setFont()«. Über affine Transformationen, entweder direkt auf den Font oder auf das Graphics-Objekt angewendet, sind verschiedenste Darstellungen möglich (siehe Abbildung 2). Fonts lassen sich damit also vergrößern, verkleinern, drehen, scheren oder spiegeln.

Auch Listing 4 ist noch vergleichsweise simpel. Die erweiterten Möglichkeiten der 2D-Engine erlauben aber die Textpositionierung bis hinunter zur Ebene der

Abbildung 2: Auf einem »Canvas« ausgegebene Fonts lassen sich drehen oder anders affin transformieren.

Zeichen (Glyphs). Für gewöhnliche Anwendungen in unserem Sprachraum ist dies wohl nur selten erforderlich. In komplizierteren Fällen lassen sich aber damit Buchstaben beispielsweise mit diakritischen Zeichen wie Akzenten kombinieren, falls kein Unicode-Font dafür existieren sollte.

finally{}

Unterschiedliche Fonts mit Java nutzen ist inzwischen auch unter Linux einfach. Ob es sinnvoll ist, vieles beim Font-Handling selbst zu programmieren, sollte man sich aber gut überlegen. Eigenartigerweise gibt es innerhalb des GUI-Toolkits Swing keinen Standarddialog für die Fontauswahl, aber mit den hier vorgestellten Codeschnipseln wäre dies nicht schwer. Im Internet existieren schon entsprechende Beispiele – eine Suche bei Google fördert gleich mehrere Treffer zu Tage.

Die kompletten Listings der Beispielprogramme sind wie üblich auf dem Listings-Server des Linux-Magazins [2] abgelegt. Wer mehr zum Thema Fonts und Textsatz wissen will, sollte sich die entsprechenden Teile der JDK-Dokumentation (API-Docs und die verschiedenen Guides) ansehen.

Ein Bereich, der durchaus für das Thema Fonts interessant wäre, wurde explizit ausgeklammert: das Drucken. Java bietet zwar ein umfangreiches und leistungsstarkes Druck-API, es greift aber erst, wenn das Dokument erstellt ist (Drucker finden, Druckjob erzeugen, Status verfolgen und so weiter). Ein Textdokument mit verschiedenen Fonts aufs Papier bringen ist mit reinen Java-Mitteln mühselig und wird vom JDK kaum unterstützt. *(ofr)* ■

Infos

[1] Donald E. Knuth, Computer Modern Typefaces, Volume E, „Computers and Typesetting“: Reading, Massachusetts, Addison-Wesley 1986

[2] Listings zu diesem Artikel: [http://www.linux-magazin.de/Service/Listings/2004/02/Coffeeshop]

Der Autor

Bernhard Bablok arbeitet bei der Allianz Versicherungs AG im Bereich Data-Warehouse-Systeme. Wenn er nicht Musik hört, mit dem Radl oder zu Fuß unterwegs ist, beschäftigt er sich mit Themen rund um Objektorientierung. Er ist unter [coffee-shop@bablokb.de] zu erreichen.

Listing 4: »DrawString.java«

```
01 import java.awt.*;
02 import java.awt.event.*;
03 import javax.swing.*;
04
05 ...
06
07    public void createGUI() {
08       DrawCanvas canvas = new DrawCanvas();
09       getContentPane().add(new JScrollPane
            (canvas));
10    }
11
12    public static void main(String[] args) {
13
14       DrawString ds = new DrawString();
15
16       ds.addWindowListener(
17          new WindowAdapter() {
18             public void windowClosing
                  (WindowEvent event) {
19                System.exit(0);
20             }
21          }
22       );
23
24       ds.createGUI();
25       ds.setSize(new Dimension(200,200));
26       ds.setTitle("DrawString");
27       ds.show();
28
29    }
30
31    private static class DrawCanvas extends
       Canvas {
32       public void paint(Graphics g) {
33          g.setFont(Font.decode("SansSerif-
               bold-20"));
34          g.drawString("Coffee-Shop",50,50);
35          ((Graphics2D)
               g).rotate(Math.PI/2.0,50,50);
36          g.drawString("Coffee-Shop",50,70);
37       }
38    }
39 }
```